Proportional Rate Reduction for TCP

Nandita Dukkipati, Matt Mathis, Yuchung Cheng, Monia Ghobadi
Google, Inc.
Mountain View
California, U.S.A
{nanditad, mattmathis, ycheng}@google.com, monia@cs.toronto.edu

ABSTRACT

Packet losses increase latency for Web users. Fast recovery is a key mechanism for TCP to recover from packet losses. In this paper, we explore some of the weaknesses of the standard algorithm described in RFC 3517 and the non-standard algorithms implemented in Linux. We find that these algorithms deviate from their intended behavior in the real world due to the combined effect of short flows, application stalls, burst losses, acknowledgment (ACK) loss and reordering, and stretch ACKs. Linux suffers from excessive congestion window reductions while RFC 3517 transmits large bursts under high losses, both of which harm the rest of the flow and increase Web latency.

Our primary contribution is a new design to control transmission in fast recovery called proportional rate reduction (PRR). PRR recovers from losses quickly, smoothly and accurately by pacing out retransmissions across received ACKs. In addition to PRR, we evaluate the TCP early retransmit (ER) algorithm which lowers the duplicate acknowledgment threshold for short transfers, and show that delaying early retransmissions for a short interval is effective in avoiding spurious retransmissions in the presence of a small degree of reordering. PRR and ER reduce the TCP latency of connections experiencing losses by 3-10% depending on the response size. Based on our instrumentation on Google Web and YouTube servers in U.S. and India, we also present key statistics on the nature of TCP retransmissions.

Categories and Subject Descriptors

C.2 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Protocols

General Terms

Algorithms, Design, Experimentation, Measurement, Performance.

IMC'11, November 2–4, 2011, Berlin, Germany.

Keywords

TCP, fast recovery, proportional rate reduction, early retransmit, retransmission statistics.

1. INTRODUCTION

Web latency plays a key role in producing responsive Web applications, making information more accessible and helping to advance new cloud-based applications. There are many factors that contribute to Web latency including content that is not optimized for speed, inefficient Web servers, slow browsers, limited network bandwidth, excess losses and suboptimal network protocols. In this paper, we focus on reducing latency for TCP connections experiencing packet losses. Measurements show that over 6% of HTTP responses served from Google.com experience losses and that these losses impact user experience. We investigate some of these loss statistics and revisit TCP's loss recovery mechanisms with the goal of reducing Web latency for users.

To get a sense for how much packet losses increase Web latency, we compare the TCP latency of HTTP responses experiencing losses to their ideal achievable latency. Figure 1 (top plot) shows the average TCP latency for responses with sizes ranging between 4kB and 8kB broken down in 200ms round-trip time (RTT) buckets, measured from billions of user TCP connections to Google Web servers world-wide. The TCP latency of a HTTP response is measured from when the server sends the first byte until it receives the acknowledgment (ACK) for the last byte. We define the ideal response time to be the fixed portion of the network delay, which we approximate to be the minimum measured RTT for any given HTTP response. The approximation works well because the 4-8kB responses fit well within TCP's initial congestion window of 10 segments used on Google servers. Responses experiencing losses last 7-10 times longer than the ideal, while those with no losses are very close to the ideal. The CDF in Figure 1 (bottom plot) shows that the latency spread for responses with losses is 200 RTTs - about 10x greater than those without losses. Several independent factors, including slow user network bandwidths, long queuing delays, and TCP's mechanisms must be addressed for latency to approach the ideal.

TCP has two primary means of detecting and recovering from losses. First is fast retransmit, where TCP performs a retransmission of the missing segment after receiving a certain number of duplicate acknowledgments (dupacks) [4]. As a fall back, whenever fast retransmission is unsuccessful or when a sender does not receive enough duplicate ACKs, TCP uses a second slower but more robust mechanism where

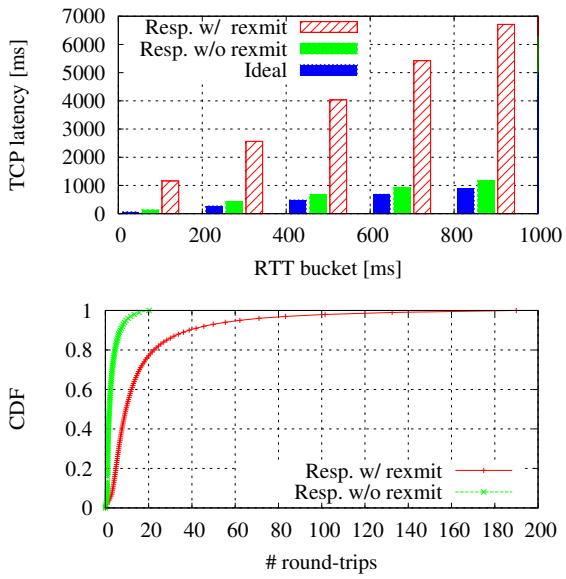

Figure 1: Top plot shows the average TCP latency of Google HTTP responses in different round-trip time buckets for response sizes between 4kB and 8kB. Bottom plot shows CDF of the number of RTTs taken by responses of all sizes with and without retransmissions.

it waits for a duration of retransmission timeout (RTO) before deducing that a segment is lost.

Our measurements show that fast recovery accounts for about 25% of all retransmissions in short flows served from Google Web servers and over 50% of retransmissions in bulk video traffic. There are two widely deployed algorithms used to adjust the congestion window (*cwnd*) during fast recovery, RFC 3517 [8] and rate halving [17], implemented in Linux. After extensive analysis of these algorithms on Google servers, we find that in practice both can be either too conservative or too aggressive resulting in a long recovery time or in excessive retransmissions.

The three main contributions of this paper are as follows:

1. **Proportional rate reduction:** We designed a new fast recovery algorithm, proportional rate reduction (PRR), which is inspired by the rate halving algorithm [17]. PRR recovers from losses both *quickly* and *smoothly* by using Van Jacobson's packet conservation principle to pace out retransmissions across received ACKs. Specifically, PRR improves upon the existing (non-standard) Linux fast recovery and RFC specified standard fast recovery under a number of conditions including: a) burst losses where the losses implicitly reduce the amount of outstanding data (*pipe*) below the slow start threshold value selected by the congestion control algorithm and, b) losses near the end of short flows where application runs out of data to send. PRR also performs accurate *cwnd* adjustments even under stretch ACKs or ACK loss and reordering. Furthermore, PRR is designed to work with the diverse set of congestion control algorithms present in today's Internet [28]: when the recovery finishes, TCP completes the window adjustment as intended by the congestion control algorithm.

   PRR has been approved to become the default Linux fast recovery algorithm for Linux 3.x.

2. **Experiments with Early Retransmit (ER):** We address the question of how to trigger fast retransmit for short flows. This question is important because short HTTP transactions frequently do not receive the three dupacks necessary to trigger conventional fast recovery. We evaluate the TCP early retransmit (ER) algorithm [2] that lowers duplicate acknowledgment threshold (*dupthresh*) for short transfers. Google Web server experiments show that the Internet has enough reordering to cause naive ER to exhibit excessive spurious retransmissions. We show that delaying early retransmissions for a short interval is effective at mitigating the spurious retransmissions.

3. **Retransmission statistics of Google Web servers:** We analyse TCP data from Google Web and video servers and present key statistics on the nature of retransmissions.

The rest of the paper is organized as follows: Section 2 describes TCP measurements of Google's user traffic. Section 3 reviews state-of-the-art fast recovery as described in the standards and the non-standard Linux variant. In Section 4, we present the design and properties of proportional rate reduction. Section 5 evaluates PRR's performance on Google Web and YouTube servers. In Section 6, we describe the experimental results of early retransmit. Section 7 summarizes related work. Section 8 concludes the paper along with a discussion on future work.

2. GOOGLE MEASUREMENTS

We sampled TCP statistics on unmodified Google Web servers world-wide for one week in May 2011. The servers use a recent version of Linux 2.6 with settings listed in Table 4. The servers do not include the changes proposed in this paper. The global statistics for interactive Web services are summarized in Table 1. Among the billions of connections sampled, 96% of the connections negotiated SACK but only 12% of the connections negotiated Timestamps. The majority of clients are Microsoft Windows which by default do not use TCP Timestamps.

Although over 94% of connections use HTTP/1.1 and the Google Web Servers keep idle TCP connections up to 4 minutes, there were on average only 3.1 HTTP requests per connection. The average HTTP response size from Google.com was 7.5KB, which is similar to the average Web object size of 7.2KB measured for billions of Web sites in 2010 [21]. The average user network bandwidth as observed from Google is 1.9Mbps and is in agreement with another study in 2010 [1]. While the average per segment TCP retransmission rate is 2.8%, 6.1% of the HTTP responses have TCP retransmissions.

2.1 Detailed retransmission statistics

We examine the existing[1] Linux loss recovery mechanisms as measured from the Google Web servers. We measure the

[1]Note PRR is already slated to be released in upstream kernels.

TCP	
Total connections	Billions
Connections support SACK	96%
Connections support Timestamp	12%
HTTP/1.1 connections	94%
Average requests per connection	3.1
Average retransmissions rate	2.8%
HTTP	
Average response size	7.5kB
Responses with TCP retransmissions	6.1%

Table 1: Summary of Google TCP and HTTP statistics sampled for one week in May 2011. The data include both port 80 and 443 but exclude YouTube videos and bulk downloads.

	DC1	DC2
Fast retransmits	24%	54%
Timeout retransmits	43%	17%
Timeout in Open	30%	8%
Timeout in Disorder	2%	3%
Timeout in Recovery	1%	2%
Timeout Exp. Backoff	10%	4%
Slow start retransmits	17%	29%
Failed retransmits	15%	0%

Table 2: A breakdown of retransmissions on Google Web servers in two data centers. All percentages are with reference to the total number of retransmissions.

types of retransmissions from a large U.S. data center, DC1, which primarily serves users from the U.S. east coast and South America. We selected this data center because it has a good mix of different RTTs, user bandwidths and loss rates. We also measured another data-center, DC2, which is in India and exclusively serves YouTube videos. For ease of comparison analysis, we also use the same data centers to experiment with our own changes to fast recovery described in later sections.

We collect the Linux TCP SNMP statistics from Google Web servers for 72 hours at DC1 in April 2011 and DC2 in August 2011. Observations described here are consistent across several such sample sizes taken in different weeks and months. The average (server) retransmission rates are 2.5% in DC1 and 5.6% in DC2. DC2 has a higher retransmission rate because the network in India has lower capacity and higher congestion.

Table 2 shows the breakdown of TCP retransmission types. It shows that fast recovery is a key mechanism to recover losses in both bulk download (video) and short flows (Web pages). In DC2, the retransmissions sent during fast recovery, labeled as *fast retransmits*, comprise 54% of all retransmissions. In DC1, fast recovery still recovers nearly a quarter (24%) of losses. This difference between DC1 and DC2 is because the long video flows of DC2 have a greater chance of entering fast recovery compared to the shorter Web traffic of DC1. The first retransmission upon a timeout, labeled as *timeout retransmits*, constitute 43% of the retransmissions in DC1. This is mainly caused by the Web

	DC1	DC2
Fast retransmits/FR	3.15	2.93
DSACKs/FR	12%	4%
DSACKs/retransmit	3.8%	1.4%
Lost (fast) retransmits/FR	6%	9%
Lost retransmits/retransmit	1.9%	3.1%

Table 3: Fast recovery related statistics on Google Web servers. All numbers are with respect to the total number of fast recovery events or fast retransmits as indicated.

objects that are too small to trigger fast recovery. Moreover, the highest query volume in DC1 turns out to be from statistics collection applications such as Google Analytics. Typically their HTTP responses are tiny and fit entirely into one segment. As a result losses can not cause any dupacks so the only available recovery mechanism is a timeout. Interestingly, DC1 and DC2 shows very different distribution of timeouts in various TCP recovery states. In DC1, the majority of timeouts happen in the open state, i.e., without any preceding dupacks or other indication of losses.[2] However in DC2, more timeouts occur in non-Open states.

In DC1 and DC2, 17% and 29% of total retransmissions occur in the slow start phase after the timeout retransmission has successfully advanced *snd.una*. These are called *slow start retransmissions* because typically the sender has reset *cwnd* to one after a timeout and it is operating in slow start phase. DC2 has more timeout retransmissions than DC1 because the timeout happens when outstanding data are typically much larger for video download. The three types of retransmissions, fast retransmits, timeout retransmits, and slow start retransmits, successfully recover the losses and constitute 85% and 100% of the total retransmissions in data centers DC1 and DC2 respectively.

For the remaining 15% retransmissions in DC1 termed *failed retransmits*, TCP failed to make any forward progress because no further TCP acknowledgments were received from the client and the server eventually aborts the connection. This difference is partially due to differing maximum timeout setting on the servers in DC1 and DC2, however we suspect that there may be other contributing factors.

Table 3 shows some additional statistics on how well fast recovery is performing.

Both data centers exhibit about three fast retransmits per fast recovery event. This suggests that loss is highly correlated (a property of the network) and provides a sense of how much the network around each data center exercises TCP's loss recovery machinery. This metric should be mostly insensitive to changes in recovery algorithms, although there may be some higher order effects.

Duplicate segments at the TCP receiver normally trigger DSACKs [9]. The DSACK statistics provide a measure of network resources that were wasted by overly aggressive retransmits of nearly all types, except segments with no pay-

[2]This statistic is particularly odd, because a timeout from the open state implies that an entire window of data or ACKs was lost in a single RTT. Although the high volume of tiny transactions contribute to timeouts from open, they can not explain numbers this large. Some other mechanism must be present.

load. In DC1, we see an average of 12% fast-recoveries causing DSACKS, implying at least a 3.8% spurious retransmission rate. In DC2 the spurious retransmission rate is at least 1.4%. Note that since some Web clients don't generate DSACKS, these are lower bounds of the actual spurious retransmissions.

Linux also detects lost retransmissions as described in section 3.2. Since the retransmissions are delivered in a new round trip, they provide some indication of how quickly congestion subsides following the initial packet losses. With lost retransmission detection, lost retransmissions cause additional RTTs in recovery. Without it, they cause timeouts. In DC1 and DC2, about 2% and 3% fast-retransmits are lost.

The data shown here is for the Linux baseline, however these metrics are important for comparing any recovery algorithms, and are used elsewhere.

3. STATE-OF-THE-ART FAST RECOVERY

Our work is based on the Linux TCP implementation, which includes several non-standard algorithms and experimental RFCs. The goal of this section is two-fold: first, we highlight the differences between standard TCP recovery algorithms [4, 8] and the design of widely deployed Linux recovery. Second, we discuss the drawbacks of each design, specifically in the context of short Web transactions.

3.1 Fast recovery in RFC standards

Standard congestion control in RFC 5681 [4] requires that TCP reduce the congestion window in response to losses. Fast recovery, described in the same document, is the reference algorithm for making this adjustment. Its stated goal is to recover TCP's self clock by relying on returning dupacks during recovery to clock more data into the network. Depending on the pattern of losses, we find that the standard can be either too aggressive or too conservative.

Algorithm 1: RFC 3517 fast recovery

On entering recovery:

`// cwnd used during and after recovery.`
$cwnd = ssthresh = FlightSize/2$

`// Retransmit first missing segment.`
$fast\_retransmit()$

`// Transmit more if cwnd allows.`
$Transmit\ MAX(0, cwnd - pipe)$

For every ACK during recovery:

$update\_scoreboard()$ $pipe$ = (RFC 3517 pipe algorithm)

$Transmit\ MAX(0, cwnd - pipe)$

Algorithm 1 briefly presents the standard RFC design for recovery. A TCP sender enters fast recovery upon receiving *dupthresh* number of dupacks (typically three). On entering recovery, the sender performs the following: 1) sets *cwnd* and *ssthresh* to half of the data outstanding in the network and, 2) *fast retransmit* the first unacknowledged segment, and further transmits more segments if allowed by *cwnd*.

On subsequent ACKs during recovery, the sender recomputes the *pipe* variable using the procedure specified in RFC 3517. *pipe* is an estimate of the number of total segments in the network based on the SACK scoreboard. It is only an estimate because it relies on heuristics that determine if a segment can be marked as lost. The algorithm then transmits up to $cwnd - pipe$ segments. Fast recovery ends when all data that was outstanding before entering recovery is cumulatively acknowledged or when a timeout occurs.

There are two main problems exhibited by the standard:

1. *Half RTT silence*: The algorithm waits for half of the received ACKs to pass by before transmitting anything after the first fast rertansmit. This is because *cwnd* is brought down to *ssthresh* in one step, so it takes *cwnd-ssthresh* ACKs for *pipe* to go below *cwnd*, creating a silent period for half of an RTT. This design wastes precious opportunities to transmit which sometimes results in nothing being sent during recovery. This in turn increases the chances of timeouts.

2. *Aggressive and bursty retransmissions*: The standard can transmit large bursts on a single received ACK. This is because $pipe - cwnd$ can be arbitrary large under burst losses or inaccurate estimation of losses. Furthermore, the more losses there are, the larger the bursts transmitted by the standard.

Note that both problems occur in the context of *heavy* losses, wherein greater than or equal to half of the *cwnd* is lost. We will see in Section 5 that such heavy losses are surprisingly common for both Web and YouTube.

Features	RFC	Linux	Default
Initial cwnd	3390	p	10
Cong. control (NewReno)	5681	+	CUBIC
SACK	2018	+	on
D-SACK	3708	+	on
Rate-Halving [17]		+	always on
FACK [16]		+	on
Limited-transmit	3042	+	always on
Dynamic *dupthresh*		+	always on
RTO	2988	p	min=200ms
F-RTO	5682	+	on
Cwnd undo (Eifel)	3522	p	always on
TCP segmentation offload		+	determined by NIC

+ indicates the feature is fully implemented.
p indicates a partially implemented feature.

Table 4: Loss recovery related features in Linux. Non-standard features are those without RFC numbers.

3.2 Fast recovery in Linux

Linux implements a rich set of standard and non-standard loss recovery algorithms [23]. Table 4 lists the main algorithms related to loss recovery and their default settings.

Linux keeps a SACK scoreboard as described in RFC 2018 [18] and computes *pipe*, the estimate of the number of segments in the network, per RFC 3517. Our detailed analysis of TCP retransmissions is based on the following four recovery states in the Linux TCP code base:

- Open: There is no missing data and the sender is receiving in-sequence ACKs.
- Disorder: The sender has received dupacks indicating that there is data missing, but has not yet retransmitted anything.

- Recovery: There has been a fast retransmit.
- Loss: There has been a timeout and resetting of *cwnd* to one to recover TCP's self clock.

There are three key differences between Linux and RFC 3517.

Linux is more aggressive in marking segments lost because it implements several algorithms from FACK [16] in addition to the standard loss detection described in RFC 3517. FACK was developed at an earlier time when network reordering was relatively rare, so it makes some simplifying assumptions. Specifically, fast retransmit can be triggered immediately by the very first SACK if it indicates that more than dupthresh segments are missing (so called threshold retransmission). Furthermore, once in fast recovery, all holes below the highest SACK block are assumed to be lost and marked for retransmission. Linux also includes a slightly later algorithm for detecting lost retransmissions. If any new data sent after a retransmission is later SACKed, the retransmission is deemed to have been lost [17]. If any reordering is detected then FACK and some of the related algorithms are disabled, the loss detection falls back to use conventional *dupthresh*.

Linux implements the rate halving algorithm [17, 19] in recovery. When *cwnd* is reduced, Linux sends data in response to alternate ACKs during recovery, instead of waiting for *cwnd*/2 dupacks to pass as specified in the standard. A minor problem with rate *halving* is that it is based on the original Reno TCP that always halved the *cwnd* during fast recovery. Several modern congestion control algorithms, such as CUBIC [10], reduce the window by less than 50% so unconditionally *halving* the rate is no longer appropriate.

While in recovery, Linux prevents bursts by reducing *cwnd* to *pipe*+1 on every ACK that reduces *pipe* for some reason. This implies that if there is insufficient new data available (e.g., because the application temporarily stalls), *cwnd* can be become one by the end of recovery. If this happens, then after recovery the sender will slow start from a very small *cwnd* even though only one segment was lost and there was no timeout!

The main drawbacks of the fast recovery in Linux are its excessive window reductions and conservative retransmissions, which occur for the following reasons:

1. *Slow start after recovery* : Even for a single loss event, a connection carrying short Web responses can complete recovery with a very small *cwnd*, such that subsequent responses using the same connection will slow start even when not otherwise required.

2. *Conservative retransmissions* : There are at least two scenarios where retransmissions in Linux are overly conservative. In the presence of heavy losses, when *pipe* falls below *ssthresh*, Linux (re)transmits at most one packet per ACK during the rest of recovery. As a result, recovery is either prolonged or it enters an RTO.

   A second scenario is that rate halving assumes every ACK represents one data packet delivered. However, lost ACKs will cause Linux to retransmit less than half of the congestion window.

4. PROPORTIONAL RATE REDUCTION

Proportional Rate Reduction is designed to overcome all four problems mentioned in the previous section.

The PRR algorithm determines the number of segments to be sent per ACK during recovery to balance two goals: 1) a speedy and smooth recovery from losses, and 2) end recovery at a congestion window close to *ssthresh*. The foundation of the algorithm is Van Jacobson's packet conservation principle: segments delivered to the receiver are used as the clock to trigger sending additional segments into the network.

PRR has two main parts. The first part, the proportional part is active when the number of outstanding segments (*pipe*) is larger than *ssthresh*, which is typically true early during the recovery and under light losses. It gradually reduces the congestion window clocked by the incoming acknowledgments. The algorithm is patterned after rate halving, but uses a fraction that is appropriate for the target window chosen by the congestion control algorithm. For example, when operating with CUBIC congestion control, the proportional part achieves the 30% window reduction by spacing out seven new segments for every ten incoming ACKs (more precisely, for ACKs reflecting 10 segments arriving at the receiver).

If *pipe* becomes smaller than *ssthresh* (such as due to excess losses or application stalls during recovery), the second part of the algorithm attempts to inhibit any further congestion window reductions. Instead it performs slow start to build the *pipe* back up to *ssthresh* subject to the availability of new data to send.[3]

Note that both parts of the PRR algorithm are independent of the congestion control algorithm (CUBIC, New Reno, GAIMD [29] etc.) used to determine the new value of *ssthresh*.

We also introduced a complete description of PRR into the IETF as a possible future RFC [15].

4.1 Examples

We present TCP time-sequence graphs of packet traces to illustrate and compare the three fast recovery algorithms. The traces are produced by our Linux kernel implementation and an internally developed TCP testing tool. The measurement setup is a simple network with a 100ms RTT and 1.2Mbps link and a 1000 byte MSS. The server application writes 20kB at time 0 and 10kB at time 500ms. TCP use traditional Reno congestion control, which sets *ssthresh* to half of the *cwnd* at the beginning of recovery.

The first example is shown in Figure 2. Vertical double ended arrows represent transmitted data segments that carry a range of bytes at a particular time. Retransmitted data is red. The green staircase represents advancing *snd.una* carried by the ACKs returning to the sender, and the vertical purple lines represent SACK blocks, indicating data that has arrived at the receiver, but is above *snd.una*.

In this example, the first 4 segments are dropped. Since TCP uses FACK, in all three traces TCP enters fast recovery after receiving first dupack and sets *ssthresh* to 10 (half of *cwnd*). In the PRR trace (top), the proportion part of

[3]We considered several alternative reduction bound algorithms, as described in [15], and this variant provides the best combination of features. This combined algorithm should more properly be called Proportional Rate Reduction with Slow-Start Reduction Bound (PRR-SSRB), but we shortened it to PRR.

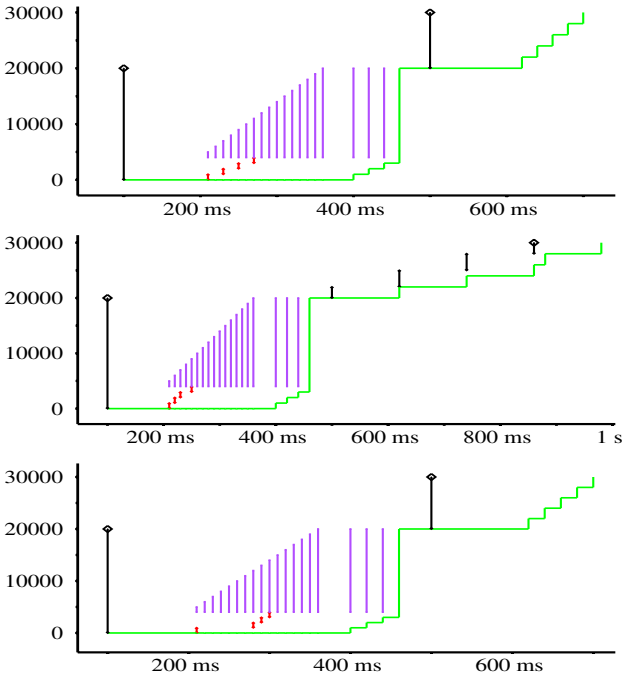

Figure 2: A comparison of PRR (top), Linux recovery (middle), and RFC 3517 (bottom). Legend for the plot: Original data transmissions (Black), retransmissions (Red), snd.una (Green), duplicate ACKs with SACK blocks (Purple).

PRR retransmits one segment every other ACK. At time 460ms, the sender completes the recovery and sets the *cwnd* to *ssthresh* (10). When the application writes 10 segments into the socket at time 500ms, the segments are sent in one RTT. The Linux trace (middle) highlights the first problem in Section 3.2. The retransmission timings are similar to the PRR trace because Linux does rate-halving. But the key difference is that *cwnd* remains $pipe + 1$ when fast recovery completes, *pipe* is 1 right before recovery completes. Therefore it takes 4 RTTs to deliver the next 10 segments in slow-start. The RFC trace (bottom) highlights the first problem in Section 3.1: the 2nd retransmission happens after about half of the *cwnd* of ACKs are received resulting, in the half RTT silence.

Figure 3 illustrates how PRR reacts to heavy losses. We use the same setup but drop segments 1 to 4 and 11 to 16. After first cluster of drops and *pipe* is larger than *ssthresh*, the proportional part of the algorithm is active transmitting on alternate received ACKs. However, after the second cluster of losses when *pipe* falls below *ssthresh*, PRR operates in slow start part and transmits two segments for every ACK.

4.2 Pseudo code for PRR

Algorithm 2 shows how PRR determines the number of bytes that should be sent in response to each ACK in recovery. The algorithm relies on three new state variables:

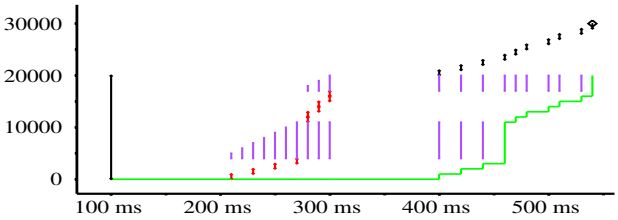

Figure 3: PRR algorithm under heavy losses.

Algorithm 2: Proportional Rate Reduction (PRR)

Initialization on entering recovery:

`// Target cwnd after recovery.`
$ssthresh = CongCtrlAlg()$

`// Total bytes delivered during recovery.`
$prr\_delivered = 0$

`// Total bytes sent during recovery.` $prr\_out = 0$

`// FlightSize at the start of recovery.`
$RecoverFS = snd.nxt - snd.una$

On every ACK during recovery compute:

`// DeliveredData is number of new bytes that the current acknowledgment indicates have been delivered to the receiver.`
$DeliveredData = delta(snd.una) + delta(SACKd)$
$prr\_delivered += DeliveredData$
$pipe =$ (RFC 3517 pipe algorithm)

if $pipe > ssthresh$ **then**
  `// Proportional Rate Reduction`
  $sndcnt = CEIL(prr\_delivered * ssthresh/RecoverFS) - prr\_out$
else
  `// Slow start`
  $ss\_limit =$
  $MAX(prr\_delivered - prr\_out, DeliveredData) + 1$
  $sndcnt = MIN(ssthresh - pipe, ss\_limit)$
$sndcnt = MAX(sndcnt, 0)$ `// positive`
$cwnd = pipe + sndcnt$

On any data transmission or retransmission:
$prr\_out += data\_sent$

At the end of recovery:
$cwnd = ssthresh$

1. *prr_delivered* is the total number of unique bytes delivered to the receiver since the start of recovery accounted through cumulative ACKs or through SACKs.

2. *prr_out* is the total bytes transmitted during recovery.

3. $RecoverFS$ is the FlightSize (defined as $snd.nxt - snd.una$ in RFC 3517 [8]) at the start of recovery.

In addition to these, PRR maintains two local variables:

1. *DeliveredData* is the number of new bytes that the current acknowledgment indicates have been delivered to the receiver.

2. *sndcnt* determines how many bytes to send in response to each ACK. Note that the decision of which data to send (e.g., retransmit lost data or send additional new data) is independent of PRR.

Algorithm 2 shows how PRR updates *sndcnt* on every ACK. When *pipe* is larger than *ssthresh*, the proportional part of the algorithm spreads the the window reductions across a full round-trip time, such that at the end of recovery, *prr_delivered* approaches $RecoverFS$ and *prr_out* approaches *ssthresh*. If there are excess losses that bring *pipe* below *ssthresh*, the algorithm tries to build the *pipe* close to *ssthresh*. It achieves this by first undoing the past congestion window reductions performed by the PRR part, reflected in the difference $prr\_delivered - prr\_out$. Second, it grows the congestion window just like TCP does in slow start algorithm. We note that this part of the algorithm increases the number of segments in flight during recovery, however it does so more smoothly compared to RFC 3517 which would send *ssthresh-pipe* segments in a single burst.

A key part of Algorithm 2 is its reliance on the newly delivered data per acknowledgment, referred above as *DeliveredData*. This is a stark difference from the existing standard and widely implemented TCP algorithms which make their congestion window adjustments based on the number of ACKs received as opposed to the actual data delivered. The use of *DeliveredData* is especially robust during recovery. The consequence of missing ACKs is that later ones will show a larger *DeliveredData*. Furthermore, for any TCP connection, the sum of *DeliveredData* must agree with the forward progress over the same time interval. In addition, for TCP using SACK, *DeliveredData* is an invariant that can be precisely computed anywhere in the network just by inspecting the returning ACKs.

4.3 Properties of PRR

In this section we discuss a few key properties of PRR that follow from its design. For each property, we also discuss the corresponding behavior for RFC 3517 and Linux fast recovery.

1. **Maintains ACK clocking:** PRR maintains ACK clocking even for large burst segment losses, primarily due to the slow start part of the algorithm. This property is not true for the existing RFC 3517 standard which under heavy losses, can send an arbitrarily large burst of size ($ssthresh - pipe$), as mentioned in the RFC problem of bursty retransmissions (Section 3.1). Linux fast recovery maintains the ACK clocking property, however it does so by bringing the congestion window down to $pipe + 1$.

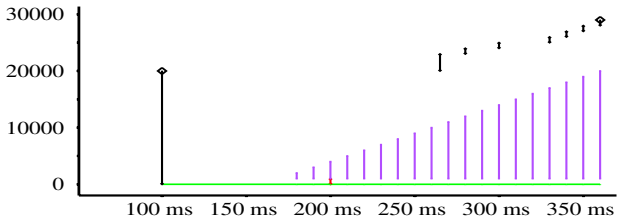

Figure 4: Example illustrating that PRR banks sending opportunities during recovery. Application first has 20 segments to send of which 1 segment is lost. In the middle of recovery, application has 10 more segments to transmit. RTT=100ms. Legend for the plot: Original data transmissions (Black), retransmissions (Red), duplicate ACKs with SACK blocks (Purple), snd.una (Green).

2. **Convergence to ssthresh:** For small number of losses ($losses \leq RecoverFS - ssthresh$), PRR converges to exactly the target window chosen by the congestion control algorithm, *ssthresh*. The algorithm for the most part operates in the proportional mode which decrements the congestion window until *pipe* reaches *ssthresh* at which point the second part of the algorithm maintains the *pipe* at the *ssthresh* value. We note that for very heavy losses, the property will not always hold true because there may not be sufficient ACKs to raise *pipe* all the way back to *ssthresh*.

   If there is sufficient new data and at least a few segments delivered near the end of the lossy round trip, RFC 3517 achieves this property regardless of the amount of loss, because it transmits a large burst. Linux fast recovery does not achieve this property when losses are heavy or when the application temporarily runs out of data to send. In both of these cases, *pipe* becomes too small and does not grow until after the recovery ends.

3. **Banks sending opportunities during application stalls:** If an application stalls during recovery, e.g., when the sending application does not queue data for transmission quickly enough or the receiver stops advancing *rwnd*, PRR stores these missed opportunities for transmission. During application stalls, *prr_out* falls behind *prr_delivered*, causing their difference $prr\_delivered - prr\_out$ to be positive. If the application catches up while TCP is still in recovery, TCP will send a burst that is bounded by $prr\_delivered - prr\_out + 1$.[4] Note that this property holds true for both parts of the PRR algorithm.

   Figure 4 shows an example of the banking property where after an idle period, the application has new data to transmit half way through the recovery process. Note that the algorithm in the PRR mode allows

[4]Although this burst might be viewed as being hard on the network, this is exactly what happens every time there is a partial RTT application stall while not in recovery. We have made the response to partial RTT stalls uniform in all states.

a burst of up to $ratio \times (prr\_delivered - prr\_out)$ which in this example is three segments (ratio is 0.5 for New Reno). Thereafter PRR continues spreading the new segments among incoming ACKs.

RFC 3517 also banks missed sending opportunities through the difference $ssthresh - pipe$. However, these banked opportunities are subject to the inaccuracies of the *pipe* variable. As discussed below, *pipe* is only an estimate of the outstanding data that can be inaccurate in certain situations, e.g., when reordered segments are incorrectly marked as lost. Linux does not achieve this property since its congestion window tracks *pipe* closely, thereby losing any history of missed sending opportunities.

4. **Robustness of DeliveredData:** An integral part of PRR is its reliance on the newly delivered data per acknowledgment or *DeliveredData*, as opposed to RFC 3517 and Linux recovery which make their congestion window adjustments based on the number of ACKs received and the *pipe* size. With SACK, *DeliveredData* allows a TCP sender to learn of the precise number of segments that arrived at the receiver. The properties below follow from the robustness of *DeliveredData*.

   Decouples data transmission from loss estimation/marking: PRR is less sensitive to errors in the *pipe* estimator compared to RFC 3517 as well as Linux fast recovery. In recovery, *pipe* is an estimator, using incomplete information to continually guess if segments that are not SACKed yet are actually lost or out-of-order in the network. *pipe* can have significant errors for some conditions, e.g., when a burst of reordered data is presumed to be lost and is retransmitted, but then the original data arrives before the retransmission. Both RFC 3517 and Linux recovery use *pipe* directly to regulate transmission rate in recovery. Errors and discontinuities in the *pipe* estimator can cause significant errors in the amount of data sent.

   On the other hand, PRR regulates the transmission rate based on the actual amount of data delivered at the receiver, *DeliveredData*. It only uses *pipe* to determine which of the two algorithm modes, proportional reduction or slow start, should compute the number of segments to send per ACK. Since both parts of the algorithm progressively converge to the same target congestion window, transient errors in the *pipe* estimator have much less impact on the final outcome.

   Precision in the number of transmitted segments: PRR retains its precision in the number of transmitted segments even in the presence of ACK loss, ACK reordering, and stretch ACKs such as those caused by Large Receive Offload (LRO) and Generic Receive Offload (GRO). The rate halving algorithm in Linux is not robust under these same scenarios as it relies heavily on the *number* of ACKs received, e.g., Linux transmits one segment on receipt of every two ACKs during rate halving, and fails to transmit the right number of segments when receiving stretch ACKs. Similarly when *pipe* estimation is incorrect in any of the above scenarios, RFC 3517 also doesn't achieve transmission of a precise number of segments.

   Data transmitted during recovery is in proportion to that delivered: For PRR the following expression holds true for the amount of data sent during recovery:

   $$prr\_out \leq 2 \times prr\_delivered$$

   The relation holds true for both parts of the PRR algorithm. Transmitted data in RFC 3517 and Linux do not have such a relation to data delivered due to their reliance on *pipe* estimate. In fact under heavy losses, the transmission rate in RFC 3517 is directly proportional to the extent of losses because of the correspondingly small value of *pipe* value.

5. EXPERIMENT RESULTS

In this section, we evaluate the effectiveness and performance of PRR in comparison to RFC 3517 and the widely deployed Linux fast recovery. We ran several 3-way experiments with PRR, RFC 3517, and Linux fast recovery on Google Web servers for five days in May 2011 and on YouTube servers in India for four days during September 2011. For fair comparisons, all three recovery algorithms use FACK loss marking. Furthermore they all use CUBIC congestion control. These newer algorithms have the effect of making RFC 3517 slightly more aggressive than envisioned by its original authors.

There are three metrics of interest when evaluating the fast recovery schemes: length of time spent in loss recovery, number of timeouts experienced, and TCP latency for HTTP responses. The recovery time is the interval from when a connection enters recovery to when it re-enters Open state. A connection may have multiple recovery events, in which case the recovery time for each of the events is logged.

We first describe our experiments on Google Web servers and then go on to describe the YouTube experiments in India.

5.1 Experiment setup on Google Web servers

All of our Web server PRR experiments were performed in a production data center (DC1 described in Section 2.1) which serves traffic for a diverse set of Google applications. The Web servers run Linux 2.6 with the default settings shown in Table 4 except that ECN is disabled. The servers terminate user TCP connections and are load balanced by steering new connections to randomly selected Web servers based on the server and client IP addresses and ports.

Calibration measurements over 24-hour time periods show that the SNMP and HTTP latency statistics agree within 0.5% between individual servers. This property permits us to run N-way experiments concurrently by changing TCP configurations on groups of servers. If we run a 4-way experiment, then 5% of current connections are served by an experimental Web server while the remaining 80% connections are served by unmodified production Web servers.

The transactions are sampled such that the aggregate rate for the experiment is roughly one million samples per day.

Note that multiple simultaneous connections opened by a single client are likely to be served by different Web servers in different experiment bins. Since we are not looking at interactions between TCP connections this does not compromise the results presented here.

$pipe < ssthresh$ [slow start]	32%
$pipe == ssthresh$	13%
$pipe > ssthresh$ [PRR]	45%
$pipe - ssthresh$	
Min	-338
1%	-10
50%	+1
99%	+11
Max	+144

Table 5: Statistics of $pipe - ssthresh$ at the start of recovery.

Quantiles for $cwnd - ssthresh$ (segments).								
Quantile:	5	10	25	50	75	90	95	99
PRR:	-8	-3	0	0	0	0	0	0

Table 6: Values for $cwnd - ssthresh$ just prior to exiting recovery.

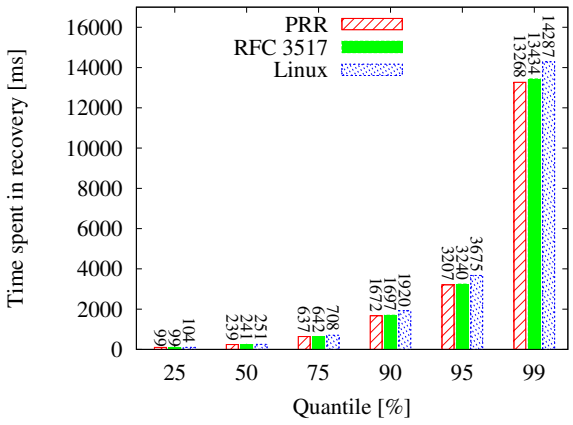

Figure 5: A comparison of time spent in recovery for PRR, RFC 3517 and Linux.

Quantiles for $cwnd$ after recovery (segments).							
Quantile:	10	25	50	75	90	95	99
PRR:	2	3	6	9	15	21	35
RFC 3517:	2	3	5	8	14	19	31
Linux:	1	2	3	5	9	12	19

Table 7: A comparison of PRR, RFC 3517 and Linux recovery.

5.2 PRR in practice

We measured the percentage of recovery events that PRR begins operation in the proportional versus the slow start modes. Measurements, summarized in Table 5, show that approximately 45% of the fast recovery events start with operating in the proportional part of the algorithm ($pipe > ssthresh$), 32% of the events start in the slow start part ($pipe < ssthresh$) and 13% of the events start with $pipe == ssthresh$. Table 6 shows that in approximately 90% of the recovery events, the congestion window in PRR converges to $ssthresh$ by the end of recovery. In the rest of the recoveries, the segment losses were too heavy for slow start to complete building the $pipe$ value to $ssthresh$.

5.3 Performance of PRR vs RFC 3517 and Linux

We first compare the recovery times, followed by retransmission statistics and impact on TCP latency for HTTP responses.

Figure 5 shows that PRR reduces the time spent in recovery as compared to both RFC 3517 and Linux fast recovery. PRR has a shorter recovery time primarily due to its smaller number of timeouts during recovery, as we will see shortly.

Table 7 shows the value of $cwnd$ on exiting recovery. PRR sets $cwnd$ to $ssthresh$ on exiting recovery, and therefore has a similar $cwnd$ distribution to that of RFC 3517. Note that the slightly larger values of $cwnd$ for PRR compared to RFC 3517 are because of the reduced number of timeouts in the recovery phase. As described in Section 3.2, Linux pulls the $cwnd$ value down to at most $pipe + 1$ in recovery. A consequence of this is that for short HTTP responses which have limited new data to transmit, over 50% of recovery events in Linux end with $cwnd$ smaller than three segments.

Table 8 compares the retransmission statistics. The main point is that PRR reduces the number of timeouts in recovery by 5% compared with Linux and by 2.6% compared with RFC 3517. Furthermore, PRR reduces the number of retransmissions by 3% compared to RFC 3517. We also note that RFC 3517 incurs greater lost retransmissions as compared to PRR. This is due to the fact that when $pipe$ falls far below $ssthresh$, RFC 3517 sends most of the retransmissions as a single large burst, thereby increasing the likelihood of drops in retransmissions. About 32% of the fast recovery events start of with $pipe$ strictly lower than $ssthresh$. In these cases, RFC 3517 has the possibility of transmitting large bursts, with the largest burst size up to 338 segments, and about 1% of the recovery events might generate bursts greater than ten segments (Table 5).

We picked two representative Google services, search and page ads, to evaluate the latency differences. Table 9 shows the quantiles for TCP latency measured. Compared to Linux fast recovery, PRR and RFC 3517 reduce the average latency of the responses with retransmissions by 3-10%, and the overall latency (including responses without losses) by 3-5%. PRR achieves similar latency as compared to RFC 3517, however, it does so without sending large bursts.

5.4 YouTube in India

We evaluated the three algorithms in our India YouTube video servers, to observe how they performed with long-running TCP transfers. The YouTube service is built on progressive HTTP which normally uses one TCP connection to deliver an entire video. The server sends the first couple tens of seconds of video as fast as possible, then it rate-limits the transfer based on the video encoding rate to minimize wasted network capacity if a user cancels the rest of the video. However, for most video downloaded in India, there is little or no difference between the throttled and un-throttled data rates because there is little or no surplus network capacity above the video encoding rate. In this case

Retransmissions measured in 1000's of segments.			
Retransmission type	Linux baseline	RFC 3517 diff. [%]	PRR diff [%]
Total Retransmission	85016	+3119 [+3.7%]	+2147 [+2.5%]
Fast Retransmission	18976	+3193 [+17%]	+2456 [+13%]
TimeoutOnRecovery	649	-16 [-2.5%]	-32 [-5.0%]
Lost Retransmission	393	+777 [+198%]	+439 [+117%]

Table 8: A comparison of retransmission statistics and timeouts of PRR, RFC 3517 and Linux recovery.

	Google Search			Page Ads		
Quantile	Linux	RFC 3517	PRR	Linux	RFC 3517	PRR
25	487	-39 [-8%]	-34 [-7%]	464	-34 [-7.3%]	-24 [-5.2%]
50	852	-50 [-5.8%]	-48 [-5.6%]	1059	-83 [-7.8%]	-100 [-9.4%]
90	4338	-108 [-2.4%]	-88 [-2%]	4956	-461 [-9.3%]	-481 [-9.7%]
99	31581	-1644 [-5.2%]	-1775 [-5.6%]	24640	-2544 [-10%]	-2887 [-11.7%]
Mean	2410	-89 [-3.7%]	-85 [-3.5%]	2441	-220 [-9%]	-239 [-9.8%]

Table 9: A comparison of TCP latency (ms) for PRR, RFC 3517 and Linux recovery. Quantiles are shown for responses that have at least one retransmission.

TCP is busy most of the time and there are long stretches where the data rate is determined entirely by TCP congestion control and the network, i.e., TCP is effectively in bulk transfer mode.

We sampled roughly 0.8 million connections in 96 hours and list key statistics in Table 10. Video downloads served with different fast recovery algorithms have almost identical sample size and average transfer size (2.3MB per connection).

The *network transmit time* measures the total time per connection when there is unacknowledged data in the TCP write queue, i.e., when TCP is actively transmitting data. Similarly, the *network recovery time* is that part of the network transmit time when the sender is in either fast recovery or timeout-recovery. Overall, RFC 3517 has the best performance since on average it delivered video in 4.7% less network transmit time than the Linux baseline. PRR is about 3% faster than the baseline.

The video downloads in India spent 43% to 46% of the network transmit time recovering from losses. While RFC 3517 spent more time and sent more retransmission in recovery than PRR and Linux baseline did, it also delivers more data during fast recovery: 5% more bytes compared to Linux baseline. This is because the network losses are clustered such that in fast recovery, *pipe* drops below *ssthresh* for about 40% of the events (not shown in table), causing RFC 3517 to send as much data necessary to keep the *pipe* at *ssthresh*. This aggressive transmission, described as Problem 2 in Section 3.1, causes 16.4% of the RFC 3517 fast-retransmits to drop, while Linux and PRR both lose less than 5% of the fast-retransmits. Furthermore after recovery, Linux transitions directly into slow-start 56% of the time. Both PRR and RFC 3517 end with *cwnd* at or close to *sshtresh* and do not need to perform this extra slow-start.

In summary, RFC performs the best but it retransmits too aggressively and has much higher (retransmission) drop rate. PRR achieves good performance with a slight increase of retransmission rate.

6. EARLY RETRANSMIT

As observed in section 2.1, the average Google HTTP response is only 7.5kB or about 5-6 segments. Early retransmit (ER) [2] is designed to overcome the well known limitation with fast retransmit: if a loss occurs too close to the end of a stream, there will not be enough dupacks to trigger a fast retransmission. ER lowers the *dupthresh* to 1 or 2 when the outstanding data drops to 2 or 3 segments respectively.

Clearly, any reordering can falsely trigger early retransmit. If this happens near the end of one HTTP response, the sender will falsely enter fast recovery which lowers the *cwnd* and slows the next HTTP response over the same connection. To make ER more robust in the presence of reordering, RFC 5827 describes three mitigation algorithms:

1. Disabling early retransmit if the connection has detected past reordering.
2. Adding a small delay to early retransmit so it might be canceled if the missing segment arrives slightly late.
3. Throttling the total early retransmission rate.

We implemented the first two algorithms. The first one is straightforward because Linux already detects reordering based on SACK. For the second algorithm, we use the RTO timer to delay the early retransmission for a configurable short interval. The sender cancels early retransmit if it receives an ACK during this interval. We do not implement the last mitigation because it only makes sense for servers facing a homogeneous user pool.

6.1 Results

We used the experimental framework described in Section 5.1 to run a 4-way experiment for 72 hours in April 2011. We compared: the baseline (original kernel), the naive ER without any mitigation, ER with first mitigation, and ER with both mitigations.

The statistics show that naive early retransmit causes a significant increase, 31%, in the number of fast retransmits for a 2% reduction in the number of timeouts relative to unmodified Linux TCP. These come at a substantial cost: a

	Linux baseline	RFC 3517	PRR
Network Transmit Time (s)	87.4	83.3	84.8
% Time in Loss Recovery	42.7%	46.3%	44.9%
Retransmission Rate %	5.0%	6.6%	5.6%
% Bytes Sent in FR	7%	12%	10%
% Fast-retransmit Lost	2.4%	16.4%	4.8%
Slow-start after FR	56%	1%	0%

Table 10: India YouTube video transfers loss recovery statistics. The average transfer size is 2.3MB and average RTT is 860ms.

Quantile	Linux	ER
5	282	258 [-8.5%]
10	319	301 [-5.6%]
50	1084	997 [-8.0%]
90	4223	4084 [-3.3%]
99	26027	25861 [-0.6%]

Table 11: A comparison of TCP latency (ms) for Linux baseline and ER with both mitigations.

27% increase in the number of "undo" events where it was determined that the retransmission was spurious, and the *cwnd* change is reversed. We were surprised by the amount of small reordering in the network. After we inspected the TCP traces with Google network operators, we confirmed that the reordering happens outside of Google networks, i.e., in the Internet.[5]

The first mitigation is not as effective because most HTTP connections are short. Although Linux also implements RFC 2140 [27] which caches the reordering history of past connections of the same IP, the cache hit rate is very low due to the server load-balancing and the size of web server farm.

The second mitigation, using a short timer to slightly delay the early retransmit, provides a substantial improvement, because it gives time for the ACKs from out–of–order segments to arrive and cancel the pending early retransmission. For the timer, we used 1/4 of the smoothed RTT bound to the range of 25ms to 500ms. One of the things that we noticed when we were tinkering with the timer design was that it does not affect the results much. This insensitivity to timer details is interesting, because it gives some hints about typical reordering distance, but we chose not to investigate it at this point and picked a reasonable compromise design.

ER with both mitigations can reduce 34% of the timeouts in Disorder state with 6% of early retransmits identified as spurious retransmissions via DSACKs. Some losses in ER with mitigation are now repaired by fast recovery instead of by timeout, therefore the total number of retransmissions remain almost the same as the baseline (1% increase). We compare the TCP latencies in ER with both mitigations and the original Linux in Table 11. Responses that do not experience losses or fit entirely into one segment are excluded because ER can not repair these responses. For the remaining responses, ER with both mitigations reduces latencies up to 8.5% and is most effective for short transactions. However, the overall latency reduction by ER is significantly limited in Google Web servers. Since the number of timeouts in disorder is very small compared to timeouts occurring in the open state.

We end the section with a note on the combined experiments with PRR and early retransmit. The main observation is that the two mechanisms are independent features of TCP fast recovery, wherein the early retransmit algorithm determines when a connection should enter recovery, while PRR improves the recovery process itself.

[5]The traces also reveal that such reorderings are likely due to router load-balancing on the forward paths where the last sub-MSS segment is SACKed before the prior full MSS segment. Clearly, the mitigations in ER are needed [6].

7. RELATED WORK

Several researchers have investigated the performance of TCP loss recovery in TCP traces of real user traffic [12, 5, 22, 26]. In data from 1995 it was observed that 85% of timeouts were due to receiving insufficient duplicate acknowledgments to trigger Fast Retransmit [12]. Interestingly, Google still shows 46% to 60% retransmissions happen during timeout recovery. In a study of the 1996 Olympic Web servers it was estimated that SACK might only save 4% of the timeouts [5]. The authors invented limited transmit which was standardized [3] and widely deployed. An analysis of the Coral CDN service identified loss recovery as one of the major performance bottlenecks [26]. The idea of progressively reducing the window, rather than a half RTT silence, has been explored before [11], including prior work by Mathis [19].

Improving the robustness and performance of TCP loss recovery has been a recurrent theme in the literature. TCP recovery improvements fall into several broad categories: better strategies for managing the window during recovery [11, 16, 5]; detecting and compensating for spurious retransmissions triggered by reordering [7, 13, 14]; disambiguating loss and reordering at the end of a stream [24]; improvements to the retransmission timer and its estimator.

The Early Retransmit algorithm shares many similarities with thin stream support for TCP [20], which is designed to help applications such as interactive games and telephony that persistently send small messages over TCP.

8. CONCLUSION

Proportional rate reduction improves fast recovery under practical network conditions. PRR operates smoothly even when losses are heavy, is quick in recovery for short flows, and accurate even when acknowledgments are stretched, lost or reordered. In live experiments, PRR was able to reduce the latency of short Web transfers by 3-10% compared to Linux recovery and proved to be a smoother recovery for video traffic compared to the standard RFC 3517. While

these seem like small improvements, we would like to emphasize that even delays as small as 100ms in page load times have been shown to impact user satisfaction in several independent experiments [25]. Based on its promise in live experiments, PRR was accepted to be in the mainline Linux as the default fast recovery algorithm, and is proposed as an experimental RFC in the IETF [15].

Going forward, we will revisit the effectiveness of the RTO mechanisms in practice. Measurements show that timeouts (and any subsequent exponential backoffs) in short flows constitute over 60% of the retransmissions. Most of these occur when flows are in the Open state and receive no duplicate acknowledgments. Our research will address if and how timeouts can be improved in practice, especially for short flows.

9. ACKNOWLEDGMENTS

We would like to thank Ankur Jain, Bernhard Beck, Tayeb Karim, John Reese for facilitating our experiments on live traffic. The following people greatly helped to improve the paper itself: Larry Brakmo, Neal Cardwell, Jerry Chu, Ilpo Järvinen, Sivasankar Radhakrishnan, Neil Spring, Mukaram Bin Tariq, and our anonymous reviewers.

10. REFERENCES

[1] State of the internet, 2010. http://www.akamai.com/stateoftheinternet/.

[2] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and Hurtig, P. Early retransmit for TCP and SCTP, May 2010. RFC 5827.

[3] Allman, M., Balakrishnan, H., and Floyd, S. Enhancing TCP's loss recovery using limited transmit, January 2001. RFC 3042.

[4] Allman, M., Paxson, V., and Blanton, E. TCP congestion control, September 2009. RFC 5681.

[5] Balakrishnan, H., Padmanabhan, V. N., Seshan, S., Stemm, M., and Katz, R. H. TCP behavior of a busy internet server: Analysis and improvements. *In Proc. of INFOCOMM* (1998).

[6] Bennet, J. C., Patridge, C., and Shectman, N. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. on Networking* (December 1999).

[7] Blanton, E., and Allman, M. Using TCP DSACKs and SCTP duplicate TSNs to detect spurious retransmissions, February 2004. RFC 3708.

[8] Blanton, E., Allman, M., Fall, K., and Wang, L. A conservative SACK-based loss recovery algorithm for TCP, 2003. RFC 3517.

[9] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An extension to the SACK option for TCP, July 2000. RFC 2883.

[10] Ha, S., Rhee, I., and Xu, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev. 42* (July 2008), 64–74.

[11] Hoe, J. Improving the start-up behavior of a congestion control scheme for TCP. *SIGCOMM Comput. Commun. Rev.* (August 1996).

[12] Lin, D., and Kung, H. TCP fast recovery strategies: Analysis and improvements. *In Proc. of INFOCOMM* (1998).

[13] Ludwig, R., and Katz, R. H. The eifel algorithm: Making TCP robust against spurious retransmissions. *(ACM) Computer Communication Review 30* (January 2000).

[14] Ludwig, R., and Meyer, M. The eifel detection algorithm for TCP, April 2003. RFC 3522.

[15] Mathis, M., Dukkipati, N., and Cheng, Y. Proportional rate reduction for TCP, March 2011. Work in progress, draft-mathis-tcpm-proportional-rate-reduction-00.txt.

[16] Mathis, M., and Mahdavi, J. Forward acknowledgment: refining TCP congestion control. *SIGCOMM Comput. Commun. Rev. 26* (August 1996), 281–291.

[17] Mathis, M., and Mahdavi, J. TCP rate-halving with bounding parameters, December 1997. http://www.psc.edu/networking/papers/FACKnotes/current/.

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

[19] Mathis, M., Semke, J., Mahdavi, J., and Lahey, K. The rate-halving algorithm for TCP congestion control, June 1999. draft-mathis-tcp-ratehalving-00.txt, http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt.

[20] Petlund, A., Evensen, K., Griwodz, C., and Halvorsen, P. TCP enhancements for interactive thin-stream applications. In *NOSSDAV* (2008).

[21] Ramachandran, S. Web metrics: Size and number of resources. http://code.google.com/speed/articles/web-metrics.html.

[22] Rewaskar, S., Kaur, J., and Smith, F. D. A performance study of loss detection/recovery in real-world TCP implementations. *In Proc. of ICNP* (2007).

[23] Sarolahti, P., and Kuznetsov, A. Congestion control in linux tcp. In *Proceedings of USENIX* (2002), Springer, pp. 49–62.

[24] Scheffenegger, R. Improving SACK-based loss recovery for TCP, November 2010. Work in progress, draft-scheffenegger-tcpm-sack-loss-recovery-00.txt.

[25] Schurman, E., and Brutlag, J. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. http://velocityconf.com/velocity2009/public/schedule/detail/8523.

[26] Sun, P., Yu, M., Freedman, M. J., and Rexford, J. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *SIGCOMM Workshop on Measurements Up the Stack* (August 2011).

[27] Touch, J. TCP control block interdependence, April 1997. RFC 2140.

[28] Yang, P., Luo, W., Xu, L., Deogun, J., and Lu, Y. TCP congestion avoidance algorithm identification. In *Proc. of ICDCS* (June 2011).

[29] Yang, Y., and Lam, S. General aimd congestion control. *Proc. International Conference on Network Protocols.* (November 2000).