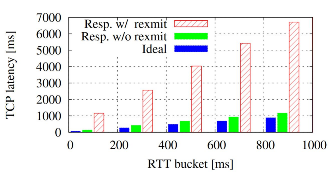
Dukkipati, N. *et al.*, "Proportional Rate Reduction for TCP," *Proc. of ACM IMC '11*, pp. 155-170, 2011.

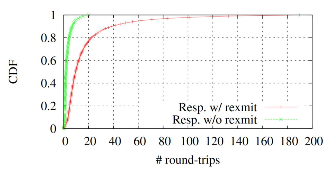
Xinghao Li
Nitish Paradkar

Introduction - Web Latency

- Web latency is a key factor that determines the user experience for web services.
- Sources of web latency:
  - Non speed-optimized content
  - Slow web servers, slow browsers and low bandwidth
  - Network protocols
  - Packet losses

Introduction - Statistics About Latency





- Over 6% of HTTP responses from Google.com got losses that impact user experience.
- Responses that experience losses have 7-10 times longer latency than those without packet loss.
- RTT range for responses with losses are 10 times larger than those without loss.

Introduction - TCP Loss Recovery Mechanisms

- Fast retransmit (cwnd will be adjusted accordingly)
  - Perform retransmission after receiving a certain number of duplicate ACKs
  - Accounts for 25% retransmissions in short flows from Google Web servers
  - Accounts for 50% retransmissions in bulk video traffic
- Wait for retransmission timeout (RTO) before consider the data was lost
  - When fast retransmission is failed or when there are insufficient packets to trigger it

Introduction - Algorithms to Adjust the cwnd

- RFC 3517
- Rate Halving (in Linux)
- Proportional rate reduction (PRR, discussed in this paper)

Contribution of this paper

- Introducing Proportional Rate Reduction (PRR)
- Introducing Early Retransmit (ER) to deal with losses in short transfers
- Demonstrating retransmission statistics of Google Web Servers

Google TCP and HTTP Measurements

- Collected data from Google web servers for one week in 2011

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%

Retransmission Statistics

- Examined loss recovery mechanisms in two data centers
  - DC 1 serving users in South America and the east coast
  - DC 2 serving YouTube videos in India
- DC 1 has short flows whereas DC 2 has long flows
- Average retransmission rates
  - 2.5% for DC 1
  - 5.6% for DC 2

Retransmission Statistics

	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%

- Fast retransmit: packets sent during fast recovery
- Timeout retransmits: retransmit upon timeout.
  - DC 1 doesn't get enough dupack's to cause fast recovery
- Slow start retransmits: sender is operating in slow start phase
- Failed retransmits: No TCP ACK's received, so connection aborted.

Fast Recovery Statistics

- DSACK measures wasted network resources by aggressive retransmits

	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%

RFC 3517 Fast Recovery

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

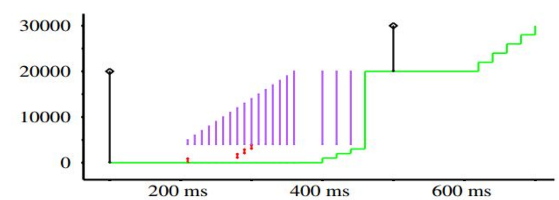
For every ACK during recovery:

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

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

- Enter recovery on receiving dupthresh dupACKs (normally 3).
- Pipe: Estimate of amount of data in network
- FlightSize: Amount of unACK'd data when entering recovery

RFC 3517 Example



- 20kB sent at 0 ms, and 10kB sent at 500 ms
- First four segments dropped
- Green represents next segment that needs an ACK
- Red shows retransmitted data
- Purple lines represent data that has arrived using SACK

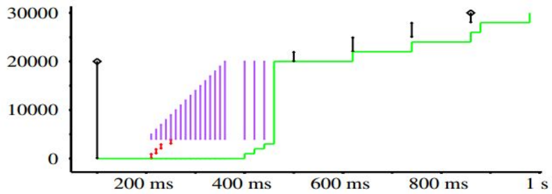
Drawbacks of RFC 3517

- Half RTT silence
  - Need to wait for at least half the cwnd before (cwnd - pipe) is positive
  - Wastes opportunities for transmitting data
- Bursty retransmissions
  - Pipe is only an estimate of the amount of data in the network
  - Cwnd - pipe can be really large, and so a large burst of data can be sent

Linux Fast Recovery

- Triggers fast retransmit with the first SACK if it indicates more than dupthresh segments have been lost
  - Results in more aggressively entering fast recovery
- Uses rate halving algorithm
  - When cwnd is reduced, send data for every 2nd ACK received
  - As opposed to waiting for cwnd/2 dupACKs to pass by before retransmitting
- Reduces cwnd to pipe + 1 for every ACK that reduces pipe
  - Can lead to extremely small cwnd at the end of fast recovery

Linux Fast Recovery Example



- 20kB sent at 0 ms, and 10kB sent at 500 ms
- First four segments are dropped
- Green represents latest unACK'd segment
- Red shows retransmitted data
- Purple lines represent data that has arrived

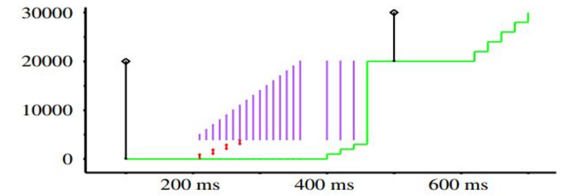
Drawbacks of Linux Fast Recovery

- Slow start after recovery
  - Can exit recovery with a very small cwnd
  - Goal of fast recovery is to end recovery without having to slow start
- Conservative retransmissions
  - Rate halving uses received ACK's to send more data into the network
  - Lost ACKs can result in less data being sent

Proportional Rate Reduction (PRR)

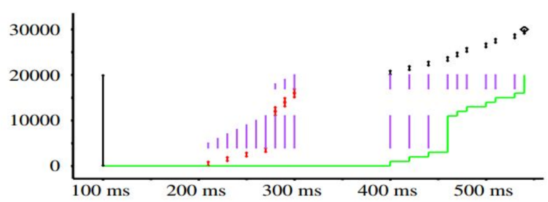
- Goals
  - Speedy and smooth recovery from losses
  - End recovery with cwnd close to ssthresh
- Proportional part
  - Active when pipe > ssthresh
  - Similar to rate halving, but uses fraction appropriate for congestion control algorithm
  - CUBIC has 30% window reduction, so send 7 segments for every 10 ACKs
- Slow start
  - Active when pipe < ssthresh
  - Perform slow start to build pipe back up the ssthresh

PRR Example



- 20kB sent at 0 ms, and 10kB sent at 500 ms
- Green represents latest unACK'd segment
- Red shows retransmitted data
- Purple lines represent data that has arrived

PRR Heavy Losses



- Segments 1-4 and 11-16 dropped
- After second series of losses, pipe < ssthresh and it is in slow start mode

PRR Pseudo Code

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:
$cwnd = ssthresh$

- prr_delivered: number of unique bytes delivered to receiver
- prr_out: total bytes transmitted during recover
- RecoverFS: flight size at the start of recovery
- DeliveredData: number of new bytes received from ACK
- DeliveredData makes algorithm less vulnerable to lost ACKs

PRR Example

```
ack#    X   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
cwnd:      20  20  11  11  11  11  11  11  11  11  11  11  11  11  11  11  11  11  11
pipe:      19  19  18  18  17  16  15  14  13  12  11  10  10  10  10  10  10  10  10
sent:       N   N   R                                   N   N   N   N   N   N   N   N


Rate-Halving (Linux)
ack#    X   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
cwnd:      20  20  19  18  18  17  17  16  16  15  15  14  14  13  13  12  12  11  11
pipe:      19  19  18  18  17  17  16  16  15  15  14  14  13  13  12  12  11  11  10
sent:       N   N   R       N       N       N       N       N       N       N       N


PRR
ack#    X   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
pipe:      19  19  18  18  18  17  17  16  16  15  15  14  14  13  13  12  12  11  10
sent:       N   N   R       N       N       N       N       N       N           N   N
RB:                                                                             s   s
```

snd_cnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
prr_out is updated on every transmission
prr_delivered updated on every ACK received

Source: https://tools.ietf.org/pdf/rfc6937.pdf

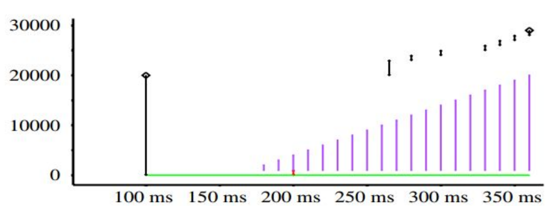
- 'N' is new data send, 'R' is retransmitted data
- Packet 0 is dropped
- Assume ssthresh is 12

Ack #	prr_ delive red	snd_ cnt	prr_ out
3	1	1	1
4	2	0	1
5	3	1	2

Properties of PRR

- Maintains ACK clocking
  - Not true for RFC 3517
- Convergence to ssthresh
  - In proportional mode reduces pipe to reach ssthresh
  - Tries to maintain pipe at ssthresh in slow start part
- Banks sending opportunities
  - When application doesn't have data to send, prr_out falls behind prr_delivered
  - This is taken care of in the slow start stage
  - ss_limit = MAX(prr_delivered - prr_out, DeliveredData) + 1

PRR Banking



Application gets more data at to send at around 275 ms

Properties of PRR

- DeliveredData allows sender to get a better idea of how much data received
- Only uses pipe to determine which mode to send in
  - RFC 3517 uses pipe to determine how much to send
  - PRR uses DeliveredData
- DelieveredData determines how many packets to send based on transmitted segments
  - Rate halving in Linux relies on number of ACKs received
- Data transmitted during recovery is in proportion to that delivered
  - prr_out <= 2 * prr_delivered

Environment Setup

- Performed in a production datacenter (DC1)
- Running on Linux 2.6 with settings (in table at right)
- ECN disabled
- TCP load balancing
- N-Way experiments
- 1 million samples per day

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.

Experiment Results - PRR in practice

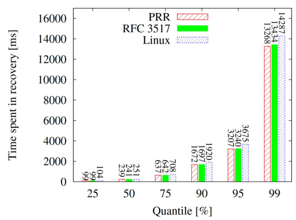
- PRR accounts for 45% of the fast recovery events
- The cwnd in PRR converges to ssthresh in about 90% of its fast recovery events.

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

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

Experiment Results - PRR vs RFC 3517 vs Linux

- PRR has the shortest recovery time
  - Since PRR has less recovery timeouts (see later)
- PRR has similar final cwnd distribution than RFC 3517
  - It sets cwnd = ssthresh on exiting
- Final cwnd values for PRR are a bit larger than that of RFC 3517
  - PRR has less recovery timeouts
- Linux algorithm has the lowest cwnd after recovery
  - It sets cwnd be at most pipe + 1 in recovery



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

Experiment Results - PRR vs RFC 3517 vs Linux

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%]

- PRR has 2.5% lower number of timeout on recovery compare to RFC 3517 and 5% lower compare to Linux
- PRR has lower number of retransmissions compare to RFC 3517
  - PRR smoothes the retransmission bursts and thus reduces the additional losses

Experiment Results - PRR vs RFC 3517 vs Linux

	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%]

- Both PRR and RFC 3517 reduce the mean latency by about 4% in Google Search and about 10% in Page Ads.

Experiment Results - YouTube in India

	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%

- RFC 3517 spends longer time (46.3%) in loss recovery, but it transfers more data (12%).
- PRR and RFC 3517 set cwnd close to ssthresh at the end of recovery, thus they do not need to perform slow-start.
- RFC 3517 does best job but has highest fast-retransmit loss rate because of larger retransmission bursts.

Early Retransmit (ER)

- Used to avoid waiting for timeout if:
  - A loss occurs at the end of a stream
  - There are insufficient duplicate ACKs to trigger the fast retransmission

- Solution: Lower the dupthresh (number of duplicate ACKs to trigger the fast retransmission) to 1 (or 2) when outstanding data drops to 2 (or 3)
  - But it may be falsely triggered by reordered packets

Early Retransmit (ER) - Mitigation Algorithms

1. Disabling early retransmit if the connection has detected past reordering
   - Detects reordering using SACK/DSACK
2. Adding a small delay to early retransmit so it might be canceled if the missing segment arrives slightly late
   - Using RTO timer to delay the early retransmission
3. Throttling the total early retransmission rate
   - Not implemented in this paper

Early Retransmit (ER) - Experiment Setting

- Test in 4-Way experiment for 72 hours
  - 4*5% connections served by 4 experimental servers
  - 80% connections served by original servers
- Compare:
  - Original kernel (baseline)
  - ER without mitigation
  - ER with first mitigation
  - ER with first and second mitigations

Early Retransmit (ER) - Results

- ER without mitigation
  - Increases 31% fast retransmits
  - Reduces 2% of timeouts
  - 27% undo events
- ER with first mitigation
  - Not effective because most HTTP connections are short
- ER with both mitigations
  - Reduces 34% of the timeouts in disorder state
  - Reduces latency by up to 8.5%

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%]

Conclusions

- PRR reduces the latency of short Web transfers by 3-10% compared to Linux recovery algorithm.
- PRR is a smoother recovery for video traffic compared to RFC 3517.
- PRR was accepted to be the default fast recovery algorithm in mainline Linux, and is proposed as an experimental RFC in the IETF.

Discussions

- Default Linux fast recovery algorithm in Linux 3.x
- Doesn’t make a strong enough case for using PRR over RFC 3517
- 6% of HTTP responses experience losses that impact user experience
  - This latency could also be correlated with bad network infrastructure
- Trying out different delays to add to ER (for cancellation)
- More explanation on prr_delivered and prr_out relationship