Internet Engineering Task Force (IETF)
Request for Comments: 6937
Category: Experimental
ISSN: 2070-1721

M. Mathis
N. Dukkipati
Y. Cheng
Google, Inc.

May 2013

Proportional Rate Reduction for TCP

Abstract

This document describes an experimental Proportional Rate Reduction (PRR) algorithm as an alternative to the widely deployed Fast Recovery and Rate-Halving algorithms. These algorithms determine the amount of data sent by TCP during loss recovery. PRR minimizes excess window adjustments, and the actual window size at the end of recovery will be as close as possible to the ssthresh, as determined by the congestion control algorithm.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for examination, experimental implementation, and evaluation.

This document defines an Experimental Protocol for the Internet community. This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6937.

Table of Contents

1. Introduction

This document describes an experimental algorithm, PRR, to improve the accuracy of the amount of data sent by TCP during loss recovery.

Standard congestion control [RFC5681] requires that TCP (and other protocols) reduce their congestion window (cwnd) 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 ACKs during recovery to clock more data into the network. Fast Recovery typically adjusts the window by waiting for one half round-trip time (RTT) of ACKs to pass before sending any data. It is fragile because it cannot compensate for the implicit window reduction caused by the losses themselves.

RFC 6675 [RFC6675] makes Fast Recovery with Selective Acknowledgement (SACK) [RFC2018] more accurate by computing "pipe", a sender side estimate of the number of bytes still outstanding in the network. With RFC 6675, Fast Recovery is implemented by sending data as necessary on each ACK to prevent pipe from falling below slow-start threshold (ssthresh), the window size as determined by the congestion control algorithm. This protects Fast Recovery from timeouts in many cases where there are heavy losses, although not if the entire second half of the window of data or ACKs are lost. However, a single ACK carrying a SACK option that implies a large quantity of missing data can cause a step discontinuity in the pipe estimator, which can cause Fast Retransmit to send a burst of data.

The Rate-Halving algorithm sends data on alternate ACKs during recovery, such that after 1 RTT the window has been halved. Rate-Halving is implemented in Linux after only being informally published [RHweb], including an uncompleted document [RHID]. Rate-Halving also does not adequately compensate for the implicit window reduction caused by the losses and assumes a net 50% window reduction, which was completely standard at the time it was written but not appropriate for modern congestion control algorithms, such as CUBIC [CUBIC], which reduce the window by less than 50%. As a consequence, Rate-Halving often allows the window to fall further than necessary, reducing performance and increasing the risk of timeouts if there are additional losses.

PRR avoids these excess window adjustments such that at the end of recovery the actual window size will be as close as possible to ssthresh, the window size as determined by the congestion control algorithm. It is patterned after Rate-Halving, but using the fraction that is appropriate for the target window chosen by the congestion control algorithm. During PRR, one of two additional Reduction Bound algorithms limits the total window reduction due to all mechanisms, including transient application stalls and the losses themselves.

We describe two slightly different Reduction Bound algorithms: Conservative Reduction Bound (CRB), which is strictly packet conserving; and a Slow Start Reduction Bound (SSRB), which is more aggressive than CRB by, at most, 1 segment per ACK. PRR-CRB meets the Strong Packet Conservation Bound described in Appendix A; however, in real networks it does not perform as well as the algorithms described in RFC 6675, which prove to be more aggressive in a significant number of cases. SSRB offers a compromise by allowing TCP to send 1 additional segment per ACK relative to CRB in some situations. Although SSRB is less aggressive than RFC 6675

(transmitting fewer segments or taking more time to transmit them), it outperforms it, due to the lower probability of additional losses during recovery.

The Strong Packet Conservation Bound on which PRR and both Reduction Bounds are based is patterned after Van Jacobson's packet conservation principle: segments delivered to the receiver are used as the clock to trigger sending the same number of segments back into the network. As much as possible, PRR and the Reduction Bound algorithms rely on this self clock process, and are only slightly affected by the accuracy of other estimators, such as pipe [RFC6675] and cwnd. This is what gives the algorithms their precision in the presence of events that cause uncertainty in other estimators.

The original definition of the packet conservation principle [Jacobson88] treated packets that are presumed to be lost (e.g., marked as candidates for retransmission) as having left the network. This idea is reflected in the pipe estimator defined in RFC 6675 and used here, but it is distinct from the Strong Packet Conservation Bound as described in Appendix A, which is defined solely on the basis of data arriving at the receiver.

We evaluated these and other algorithms in a large scale measurement study presented in a companion paper [IMC11] and summarized in Section 5. This measurement study was based on RFC 3517 [RFC3517], which has since been superseded by RFC 6675. Since there are slight differences between the two specifications, and we were meticulous about our implementation of RFC 3517, we are not comfortable unconditionally asserting that our measurement results apply to RFC 6675, although we believe this to be the case. We have instead chosen to be pedantic about describing measurement results relative to RFC 3517, on which they were actually based. General discussions of algorithms and their properties have been updated to refer to RFC 6675.

We found that for authentic network traffic, PRR-SSRB outperforms both RFC 3517 and Linux Rate-Halving even though it is less aggressive than RFC 3517. We believe that these results apply to RFC 6675 as well.

The algorithms are described as modifications to RFC 5681 [RFC5681], "TCP Congestion Control", using concepts drawn from the pipe algorithm [RFC6675]. They are most accurate and more easily implemented with SACK [RFC2018], but do not require SACK.

2. Definitions

The following terms, parameters, and state variables are used as they are defined in earlier documents:

RFC 793: snd.una (send unacknowledged)

RFC 5681: duplicate ACK, FlightSize, Sender Maximum Segment Size (SMSS)

RFC 6675: covered (as in "covered sequence numbers")

Voluntary window reductions: choosing not to send data in response to some ACKs, for the purpose of reducing the sending window size and data rate

We define some additional variables:

SACKd: The total number of bytes that the scoreboard indicates have been delivered to the receiver. This can be computed by scanning the scoreboard and counting the total number of bytes covered by all SACK blocks. If SACK is not in use, SACKd is not defined.

DeliveredData: The total number of bytes that the current ACK indicates have been delivered to the receiver. When not in recovery, DeliveredData is the change in snd.una. With SACK, DeliveredData can be computed precisely as the change in snd.una, plus the (signed) change in SACKd. In recovery without SACK, DeliveredData is estimated to be 1 SMSS on duplicate acknowledgements, and on a subsequent partial or full ACK, DeliveredData is estimated to be the change in snd.una, minus 1 SMSS for each preceding duplicate ACK.

Note that DeliveredData is robust; for TCP using SACK, DeliveredData can be precisely computed anywhere in the network just by inspecting the returning ACKs. The consequence of missing ACKs is that later ACKs will show a larger DeliveredData. Furthermore, for any TCP (with or without SACK), the sum of DeliveredData must agree with the forward progress over the same time interval.

We introduce a local variable "sndcnt", which indicates exactly how many bytes should be sent in response to each ACK. Note that the decision of which data to send (e.g., retransmit missing data or send more new data) is out of scope for this document.

3. Algorithms

At the beginning of recovery, initialize PRR state. This assumes a modern congestion control algorithm, CongCtrlAlg(), that might set ssthresh to something other than FlightSize/2:

```
ssthresh = CongCtrlAlg()  // Target cwnd after recovery
prr_delivered = 0         // Total bytes delivered during recovery
prr_out = 0               // Total bytes sent during recovery
RecoverFS = snd.nxt-snd.una // FlightSize at the start of recovery
```

On every ACK during recovery compute:

```
DeliveredData = change_in(snd.una) + change_in(SACKd)
prr_delivered += DeliveredData
pipe = (RFC 6675 pipe algorithm)
if (pipe > ssthresh) {
   // Proportional Rate Reduction
   sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
} else {
   // Two versions of the Reduction Bound
   if (conservative) {    // PRR-CRB
     limit = prr_delivered - prr_out
   } else {               // PRR-SSRB
     limit = MAX(prr_delivered - prr_out, DeliveredData) + MSS
   }
   // Attempt to catch up, as permitted by limit
   sndcnt = MIN(ssthresh - pipe, limit)
}
```

On any data transmission or retransmission:

```
prr_out += (data sent) // strictly less than or equal to sndcnt
```

3.1. Examples

We illustrate these algorithms by showing their different behaviors for two scenarios: TCP experiencing either a single loss or a burst of 15 consecutive losses. In all cases we assume bulk data (no application pauses), standard Additive Increase Multiplicative Decrease (AIMD) congestion control, and cwnd = FlightSize = pipe = 20 segments, so ssthresh will be set to 10 at the beginning of recovery. We also assume standard Fast Retransmit and Limited Transmit [RFC3042], so TCP will send 2 new segments followed by 1 retransmit in response to the first 3 duplicate ACKs following the losses.

Each of the diagrams below shows the per ACK response to the first round trip for the various recovery algorithms when the zeroth segment is lost. The top line indicates the transmitted segment number triggering the ACKs, with an X for the lost segment. "cwnd" and "pipe" indicate the values of these algorithms after processing each returning ACK. "Sent" indicates how much 'N'ew or 'R'etransmitted data would be sent. Note that the algorithms for deciding which data to send are out of scope of this document.

When there is a single loss, PRR with either of the Reduction Bound algorithms has the same behavior. We show "RB", a flag indicating which Reduction Bound subexpression ultimately determined the value of sndcnt. When there are minimal losses, "limit" (both algorithms) will always be larger than ssthresh - pipe, so the sndcnt will be ssthresh - pipe, indicated by "s" in the "RB" row.

RFC 6675

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

Cwnd is not shown because PRR does not use it.

Key for RB

```
s: sndcnt = ssthresh - pipe                 // from ssthresh
b: sndcnt = prr_delivered - prr_out + SMSS  // from banked
d: sndcnt = DeliveredData + SMSS            // from DeliveredData
```

(Sometimes, more than one applies.)

Note that all 3 algorithms send the same total amount of data. RFC 6675 experiences a "half window of silence", while the Rate-Halving and PRR spread the voluntary window reduction across an entire RTT.

Next, we consider the same initial conditions when the first 15 packets (0-14) are lost. During the remainder of the lossy RTT, only 5 ACKs are returned to the sender. We examine each of these algorithms in succession.

RFC 6675

ack#	X	X	X	X	X	X	X	X	X	X	X	X	X	X	15	16	17	18	19
cwnd:															20	20	11	11	11
pipe:															19	19	4	10	10
sent:															N	N	7R	R	R

Rate-Halving (Linux)

ack#	X	X	X	X	X	X	X	X	X	X	X	X	X	X	15	16	17	18	19
cwnd:															20	20	5	5	5
pipe:															19	19	4	4	4
sent:															N	N	R	R	R

PRR-CRB

ack#	X	X	X	X	X	X	X	X	X	X	X	X	X	X	15	16	17	18	19
pipe:															19	19	4	4	4
sent:															N	N	R	R	R
RB:																	b	b	b

PRR-SSRB

ack#	X	X	X	X	X	X	X	X	X	X	X	X	X	X	15	16	17	18	19
pipe:															19	19	4	5	6
sent:															N	N	2R	2R	2R
RB:																	bd	d	d

In this specific situation, RFC 6675 is more aggressive because once Fast Retransmit is triggered (on the ACK for segment 17), TCP immediately retransmits sufficient data to bring pipe up to cwnd. Our measurement data (see Section 5) indicates that RFC 6675 significantly outperforms Rate-Halving, PRR-CRB, and some other similarly conservative algorithms that we tested, showing that it is significantly common for the actual losses to exceed the window reduction determined by the congestion control algorithm.

The Linux implementation of Rate-Halving includes an early version of the Conservative Reduction Bound [RHweb]. In this situation, the 5 ACKs trigger exactly 1 transmission each (2 new data, 3 old data), and cwnd is set to 5. At a window size of 5, it takes 3 round trips to retransmit all 15 lost segments. Rate-Halving does not raise the window at all during recovery, so when recovery finally completes, TCP will slow start cwnd from 5 up to 10. In this example, TCP operates at half of the window chosen by the congestion control for more than 3 RTTs, increasing the elapsed time and exposing it to timeouts in the event that there are additional losses.

PRR-CRB implements a Conservative Reduction Bound. Since the total losses bring pipe below ssthresh, data is sent such that the total data transmitted, prr_out, follows the total data delivered to the receiver as reported by returning ACKs. Transmission is controlled by the sending limit, which is set to prr_delivered - prr_out. This is indicated by the RB:b tagging in the figure. In this case, PRR-CRB is exposed to exactly the same problems as Rate-Halving; the excess window reduction causes it to take excessively long to recover the losses and exposes it to additional timeouts.

PRR-SSRB increases the window by exactly 1 segment per ACK until pipe rises to ssthresh during recovery. This is accomplished by setting limit to one greater than the data reported to have been delivered to the receiver on this ACK, implementing slow start during recovery, and indicated by RB:d tagging in the figure. Although increasing the window during recovery seems to be ill advised, it is important to remember that this is actually less aggressive than permitted by RFC 5681, which sends the same quantity of additional data as a single burst in response to the ACK that triggered Fast Retransmit.

For less extreme events, where the total losses are smaller than the difference between FlightSize and ssthresh, PRR-CRB and PRR-SSRB have identical behaviors.

4. Properties

The following properties are common to both PRR-CRB and PRR-SSRB, except as noted:

PRR maintains TCP’s ACK clocking across most recovery events, including burst losses. RFC 6675 can send large unclocked bursts following burst losses.

Normally, PRR will spread voluntary window reductions out evenly across a full RTT. This has the potential to generally reduce the burstiness of Internet traffic, and could be considered to be a type of soft pacing. Hypothetically, any pacing increases the probability that different flows are interleaved, reducing the opportunity for ACK compression and other phenomena that increase traffic burstiness. However, these effects have not been quantified.

If there are minimal losses, PRR will converge to exactly the target window chosen by the congestion control algorithm. Note that as TCP approaches the end of recovery, prr_delivered will approach RecoverFS and sndcnt will be computed such that prr_out approaches ssthresh.

Implicit window reductions, due to multiple isolated losses during recovery, cause later voluntary reductions to be skipped. For small numbers of losses, the window size ends at exactly the window chosen by the congestion control algorithm.

For burst losses, earlier voluntary window reductions can be undone by sending extra segments in response to ACKs arriving later during recovery. Note that as long as some voluntary window reductions are not undone, the final value for pipe will be the same as ssthresh, the target cwnd value chosen by the congestion control algorithm.

PRR with either Reduction Bound improves the situation when there are application stalls, e.g., when the sending application does not queue data for transmission quickly enough or the receiver stops advancing rwnd (receiver window). When there is an application stall early during recovery, prr_out will fall behind the sum of the transmissions permitted by sndcnt. The missed opportunities to send due to stalls are treated like banked voluntary window reductions; specifically, they cause prr_delivered - prr_out to be significantly positive. If the application catches up while TCP is still in recovery, TCP will send a partial window burst to catch up to exactly where it would have been had the application never stalled. 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 partial RTT stall behavior uniform in all states. Changing this behavior is out of scope for this document.

PRR with Reduction Bound is less sensitive to errors in the pipe estimator. While in recovery, pipe is intrinsically an estimator, using incomplete information to estimate if un-SACKed segments are actually lost or merely out of order in the network. Under some conditions, pipe can have significant errors; for example, pipe is underestimated when a burst of reordered data is prematurely assumed to be lost and marked for retransmission. If the transmissions are regulated directly by pipe as they are with RFC 6675, a step discontinuity in the pipe estimator causes a burst of data, which cannot be retracted once the pipe estimator is corrected a few ACKs later. For PRR, pipe merely determines which algorithm, PRR or the Reduction Bound, is used to compute sndcnt from DeliveredData. While pipe is underestimated, the algorithms are different by at most 1 segment per ACK. Once pipe is updated, they converge to the same final window at the end of recovery.

Under all conditions and sequences of events during recovery, PRR-CRB strictly bounds the data transmitted to be equal to or less than the amount of data delivered to the receiver. We claim that this Strong Packet Conservation Bound is the most aggressive algorithm that does

not lead to additional forced losses in some environments. It has the property that if there is a standing queue at a bottleneck with no cross traffic, the queue will maintain exactly constant length for the duration of the recovery, except for +1/-1 fluctuation due to differences in packet arrival and exit times. See Appendix A for a detailed discussion of this property.

Although the Strong Packet Conservation Bound is very appealing for a number of reasons, our measurements summarized in Section 5 demonstrate that it is less aggressive and does not perform as well as RFC 6675, which permits bursts of data when there are bursts of losses. PRR-SSRB is a compromise that permits TCP to send 1 extra segment per ACK as compared to the Packet Conserving Bound. From the perspective of a strict Packet Conserving Bound, PRR-SSRB does indeed open the window during recovery; however, it is significantly less aggressive than RFC 6675 in the presence of burst losses.

5. Measurements

In a companion IMC11 paper [IMC11], we describe some measurements comparing the various strategies for reducing the window during recovery. The experiments were performed on servers carrying Google production traffic and are briefly summarized here.

The various window reduction algorithms and extensive instrumentation were all implemented in Linux 2.6. We used the uniform set of algorithms present in the base Linux implementation, including CUBIC [CUBIC], Limited Transmit [RFC3042], threshold transmit (Section 3.1 in [FACK]) (this algorithm was not present in RFC 3517, but a similar algorithm has been added to RFC 6675), and lost retransmission detection algorithms. We confirmed that the behaviors of Rate-Halving (the Linux default), RFC 3517, and PRR were authentic to their respective specifications and that performance and features were comparable to the kernels in production use. All of the different window reduction algorithms were all present in a common kernel and could be selected with a sysctl, such that we had an absolutely uniform baseline for comparing them.

Our experiments included an additional algorithm, PRR with an unlimited bound (PRR-UB), which sends ssthresh-pipe bursts when pipe falls below ssthresh. This behavior parallels RFC 3517.

An important detail of this configuration is that CUBIC only reduces the window by 30%, as opposed to the 50% reduction used by traditional congestion control algorithms. This accentuates the tendency for RFC 3517 and PRR-UB to send a burst at the point when Fast Retransmit gets triggered because pipe is likely to already be below ssthresh. Precisely this condition was observed for 32% of the

recovery events: pipe fell below ssthresh before Fast Retransmit was triggered, thus the various PRR algorithms started in the Reduction Bound phase, and RFC 3517 sent bursts of segments with the Fast Retransmit.

In the companion paper, we observe that PRR-SSRB spends the least time in recovery of all the algorithms tested, largely because it experiences fewer timeouts once it is already in recovery.

RFC 3517 experiences 29% more detected lost retransmissions and 2.6% more timeouts (presumably due to undetected lost retransmissions) than PRR-SSRB. These results are representative of PRR-UB and other algorithms that send bursts when pipe falls below ssthresh.

Rate-Halving experiences 5% more timeouts and significantly smaller final cwnd values at the end of recovery. The smaller cwnd sometimes causes the recovery itself to take extra round trips. These results are representative of PRR-CRB and other algorithms that implement strict packet conservation during recovery.

6. Conclusion and Recommendations

Although the Strong Packet Conservation Bound used in PRR-CRB is very appealing for a number of reasons, our measurements show that it is less aggressive and does not perform as well as RFC 3517 (and by implication RFC 6675), which permits bursts of data when there are bursts of losses. RFC 3517 and RFC 6675 are conservative in the original sense of Van Jacobson's packet conservation principle, which included the assumption that presumed lost segments have indeed left the network. PRR-CRB makes no such assumption, following instead a Strong Packet Conservation Bound in which only packets that have actually arrived at the receiver are considered to have left the network. PRR-SSRB is a compromise that permits TCP to send 1 extra segment per ACK relative to the Strong Packet Conservation Bound, to partially compensate for excess losses.

From the perspective of the Strong Packet Conservation Bound, PRR-SSRB does indeed open the window during recovery; however, it is significantly less aggressive than RFC 3517 (and RFC 6675) in the presence of burst losses. Even so, it often outperforms RFC 3517 (and presumably RFC 6675) because it avoids some of the self-inflicted losses caused by bursts.

At this time, we see no reason not to test and deploy PRR-SSRB on a large scale. Implementers worried about any potential impact of raising the window during recovery may want to optionally support PRR-CRB (which is actually simpler to implement) for comparison

studies. Furthermore, there is one minor detail of PRR that can be improved by replacing pipe by total_pipe, as defined by Laminar TCP [Laminar].

One final comment about terminology: we expect that common usage will drop "Slow Start Reduction Bound" from the algorithm name. This document needed to be pedantic about having distinct names for PRR and every variant of the Reduction Bound. However, we do not anticipate any future exploration of the alternative Reduction Bounds.

7. Acknowledgements

This document is based in part on previous incomplete work by Matt Mathis, Jeff Semke, and Jamshid Mahdavi [RHID] and influenced by several discussions with John Heffner.

Monia Ghobadi and Sivasankar Radhakrishnan helped analyze the experiments.

Ilpo Jarvinen reviewed the code.

Mark Allman improved the document through his insightful review.

8. Security Considerations

PRR does not change the risk profile for TCP.

Implementers that change PRR from counting bytes to segments have to be cautious about the effects of ACK splitting attacks [Savage99], where the receiver acknowledges partial segments for the purpose of confusing the sender’s congestion accounting.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

9.2. Informative References

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[RFC3517] Blanton, E., Allman, M., Fall, K., and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC 3517, April 2003.

[IMC11] Dukkipati, N., Mathis, M., Cheng, Y., and M. Ghobadi, "Proportional Rate Reduction for TCP", Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement 2011, Berlin, Germany, November 2011.

[FACK] Mathis, M. and J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control", ACM SIGCOMM SIGCOMM96, August 1996.

[RHID] Mathis, M., Semke, J., and J. Mahdavi, "The Rate-Halving Algorithm for TCP Congestion Control", Work in Progress, August 1999.

[RHweb] Mathis, M. and J. Mahdavi, "TCP Rate-Halving with Bounding Parameters", Web publication, December 1997, <http://www.psc.edu/networking/papers/FACKnotes/current/>.

[CUBIC] Rhee, I. and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant", PFLDnet 2005, February 2005.

[Jacobson88] Jacobson, V., "Congestion Avoidance and Control", SIGCOMM Comput. Commun. Rev. 18(4), August 1988.

[Savage99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP congestion control with a misbehaving receiver", SIGCOMM Comput. Commun. Rev. 29(5), October 1999.

[Laminar] Mathis, M., "Laminar TCP and the case for refactoring TCP congestion control", Work in Progress, July 2012.

Appendix A. Strong Packet Conservation Bound

PRR-CRB is based on a conservative, philosophically pure, and aesthetically appealing Strong Packet Conservation Bound, described here. Although inspired by Van Jacobson's packet conservation principle [Jacobson88], it differs in how it treats segments that are missing and presumed lost. Under all conditions and sequences of events during recovery, PRR-CRB strictly bounds the data transmitted to be equal to or less than the amount of data delivered to the receiver. Note that the effects of presumed losses are included in the pipe calculation, but do not affect the outcome of PRR-CRB, once pipe has fallen below ssthresh.

We claim that this Strong Packet Conservation Bound is the most aggressive algorithm that does not lead to additional forced losses in some environments. It has the property that if there is a standing queue at a bottleneck that is carrying no other traffic, the queue will maintain exactly constant length for the entire duration of the recovery, except for +1/-1 fluctuation due to differences in packet arrival and exit times. Any less aggressive algorithm will result in a declining queue at the bottleneck. Any more aggressive algorithm will result in an increasing queue or additional losses if it is a full drop tail queue.

We demonstrate this property with a little thought experiment:

Imagine a network path that has insignificant delays in both directions, except for the processing time and queue at a single bottleneck in the forward path. By insignificant delay, we mean when a packet is "served" at the head of the bottleneck queue, the following events happen in much less than one bottleneck packet time: the packet arrives at the receiver; the receiver sends an ACK that arrives at the sender; the sender processes the ACK and sends some data; the data is queued at the bottleneck.

If sndcnt is set to DeliveredData and nothing else is inhibiting sending data, then clearly the data arriving at the bottleneck queue will exactly replace the data that was served at the head of the queue, so the queue will have a constant length. If queue is drop tail and full, then the queue will stay exactly full. Losses or reordering on the ACK path only cause wider fluctuations in the queue size, but do not raise its peak size, independent of whether the data is in order or out of order (including loss recovery from an earlier RTT). Any more aggressive algorithm that sends additional data will overflow the drop tail queue and cause loss. Any less aggressive algorithm will under-fill the queue. Therefore, setting sndcnt to DeliveredData is the most aggressive algorithm that does not cause forced losses in this simple network. Relaxing the assumptions

(e.g., making delays more authentic and adding more flows, delayed ACKs, etc.) is likely to increase the fine grained fluctuations in queue size but does not change its basic behavior.

Note that the congestion control algorithm implements a broader notion of optimal that includes appropriately sharing the network. Typical congestion control algorithms are likely to reduce the data sent relative to the Packet Conserving Bound implemented by PRR, bringing TCP's actual window down to ssthresh.

Authors' Addresses

Matt Mathis
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

EMail: mattmathis@google.com

Nandita Dukkipati
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

EMail: nanditad@google.com

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

EMail: ycheng@google.com