Internet Engineering Task Force
INTERNET DRAFT
draft-mathis-tcp-ratehalving-00.txt

Matt Mathis
Jeff Semke
PSC
Jamshid Mahdavi
Novell

August 1999
Expires: February 2000

The Rate-Halving Algorithm for TCP Congestion Control

Status of this Memo

Abstract

This draft provides a detailed description of the Rate-Halving algorithm. As specified by RFC2581, Fast Recovery adjusts the congestion window (cwnd) by transmitting new data only during the second half of the recovery interval. The Rate-Halving algorithm adjusts the congestion window by spacing transmissions at the rate of one data segment per two segments acknowledged over the entire recovery period, thereby sustaining the self-clocking of TCP and avoiding a burst. Since it is largely independent of the details of the data retransmission strategy, the Rate-Halving algorithm can be used with several standard and experimental TCP implementations: NewReno, SACK, and ECN.

This algorithm was first proposed by Janey Hoe [Hoe95].

1. Introduction

All "Reno TCP" implementations include TCP Fast Retransmit and Fast Recovery algorithms [RFC2581]. Fast retransmit relies on three duplicate acknowledgements to trigger the retransmission of a single lost segment. Once the Fast Retransmit has occurred, TCP then waits for enough additional duplicate ACKs to arrive, indicating that half of the data in flight has left the network. Only when this has occurred will TCP send additional new data.

The consequence of this delay is that the entire new window of data is transmitted in one half of one Round Trip Time (RTT). This burst can cause repeated bursts in successive RTTs following the recovery, which can result in overall additional burstiness on the network.

Hoe [Hoe95] suggested that during Fast Recovery the TCP data sender space out retransmissions and new data on alternate acknowledgements across the entire recovery RTT. (Note that this eliminates the half RTT lull in sending which occurs in Reno TCP.)

This document describes a Rate-Halving algorithm which implements Hoe's idea. It explains how to implement the algorithm under NewReno, SACK, and ECN-style TCP implementations. This document does not discuss the other suggestions in [Hoe95] and [Hoe96], such as a change to the ssthresh parameter during Slow-Start, or the NewReno proposal [RFC2582].

Rate-Halving has a number of other useful properties as well. It results in a slightly lower final value for cwnd following recovery, which has been suggested by Floyd and others as the more correct value. The Rate-Halving algorithm provides proper adjustments to the congestion window in response to congestion signals such as a lost segment or an ECN-Echo bit [RFC2481]. These adjustments are largely independent of the strategy used to retransmit missing segments, allowing Rate-Halving to be extended for use in other TCP implementations or even non-TCP transport protocols.

Definitions of terms and variables that are used throughout the rest of the document can be found in Section 2. Section 3 provides an overview of the Rate-Halving algorithm. Section 4 presents a detailed specification of the algorithm. Section 5 contains related commentary, and Section 6 discusses interactions between Rate-Halving and other components of TCP.

2. Definitions

The following terms are used throughout the document, and are defined here for clarity. The definitions within quotations in this section are included from the cited documents.

SENDER MAXIMUM SEGMENT SIZE (SMSS): "The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [RFC1191] algorithm, RMSS, or other factors. The size does not include the TCP/IP headers and options." from [RFC2581]

ADJUSTMENT INTERVAL: The time during which TCP reduces the congestion window after experiencing congestion.

REPAIR INTERVAL: The time during which TCP retransmits lost segments.

RATE-HALVING STATE (rh_state): A state variable indicating what form of adjustments are being performed during adjustment intervals. The Rate-Halving states are described in Section 4.1.

CONGESTION WINDOW (cwnd): "A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd." from [RFC2581]. Cwnd, as used by Reno and NewReno TCPs, is used as an offset from snd_una (the highest-sequence segment acknowledged by the receiver, indicating that all segments less than snd_una have been received). During recovery, Reno and NewReno artificially inflate cwnd as dupacks are received.

RATE-HALVING CONGESTION WINDOW (rhcwnd): In order to differentiate the usage of cwnd within Rate-Halving from the cwnd used by Reno and NewReno [RFC2582], we introduce a new but comparable congestion window variable called rhcwnd. Rhcwnd is not inflated by dupacks during congestion episodes. It represents the amount of data that the sender is allowed to have outstanding. It is not considered an offset to snd_una, but rather a count of the data in flight, regardless of whether the segments contain new or retransmitted data.

PRIOR_RHCWND (prior_rhcwnd): The value of rhcwnd saved when the adjustment interval begins.

LOSS WINDOW (LW): "The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer." from [RFC2581]

MAX_SEQ (max_seq): The maximum sequence number transmitted for the current connection.

PRIOR_MAX_SEQ (prior_max_seq): The maximum sequence number transmitted before the start of an adjustment interval.

FORWARD ACKNOWLEDGEMENT (fack): The highest sequence number known to have reached the receiver, plus one. (This includes SACK information, and is different than the variable snd.una when losses have occurred). The use of the fack state variable (and the retran_data state variable, below) was originally described by Mathis [MM96]. The details for setting fack can be found in Section 6.1.2.

DUPLICATE ACKNOWLEDGMENTS (dupacks): Acknowledgments that carry the same sequence number as an earlier acknowledgment. In order to qualify as a dupack, the ack must carry no data, and must not be a window update. Dupacks provide an indication that the segment following the sequence number in the dupack may have been lost, but that some future segment has been received. The state variable "dupacks" is a count of the current number of "outstanding" dupacks. By "outstanding", we mean dupacks which have been received, and which represent data not yet covered by snd.una.

PARTIAL ACKNOWLEDGMENTS: Following congestion, these are "ACKs which cover new data, but not all the data outstanding when loss was detected" from [RFC2582].

FULL ACKNOWLEDGMENT: Following congestion, an ACK which covers all data outstanding when a loss was detected (i.e. an ack that is greater than or equal to prior_max_seq.)

RETRANSMITTED DATA (retran_data): The amount of retransmitted data outstanding in the network.

NUM_RETRANS (num_retrans): The number of bytes retransmitted during the current repair interval.

ECN-ECHO ACK PACKET: "If the sender receives an ECN-Echo ACK packet (that is, an ACK packet with the ECN-Echo flag set in the TCP header), then the sender knows that congestion was encountered in the network on the path from the sender to the receiver. The indication of congestion should be treated just as a congestion loss in non-ECN-Capable TCP." from [RFC2481].

In addition to the above definitions, we use the following commonly-used TCP state variables and terms: ssthresh [RFC2581], snd.una, snd.nxt [RFC793], and segment [RFC2581].

3. Overview of the Rate-Halving Algorithm

The Rate-Halving (RH) algorithm is designed to reduce the congestion window following the detection of congestion in the network. Rate-Halving reduces rhcwnd over one round trip by transmitting one new segment for every two segments acknowledged by the receiver. The Rate-Halving algorithm also detects the end of this round trip, employing several different checks to perform this detection most accurately. The choice of which segment to send at any given point in time is completely independent of the Rate-Halving algorithm, allowing Rate-Halving to be used simultaneously with ECN (where no retransmissions occur) [RFC2481], NewReno (where retransmissions are spread out one per RTT) [RFC2582], or SACK (where holes are filled soon after they are detected) [RFC2018].

As a result of the separation of window reduction from data retransmission, the term "recovery" ceases to have a clear meaning. We replace it with two terms: "adjustment interval" (referring to the round trip containing the window adjustment) and "repair interval" (referring to the retransmission of missing data). The adjustment interval is entered immediately upon indication of the possible onset of congestion. Two possible indications of congestion are presence of the ECN-Echo bit, or indications of a packet loss via duplicate ACK or via presence of a SACK block.

During the adjustment interval, the window is reduced by sending one data segment for each two segments which are acknowledged, as per the Rate-Halving algorithm (described in detail below). At the end of the adjustment interval, additional checks are made to update ssthresh and to ensure that the final value of rhcwnd is appropriate. Due to the additional window reduction for the lost data, the final value for rhcwnd is 1/2 of the correctly delivered portion of the window of data which was in the network at the time of detection.

4. Complete Rate-Halving Algorithm

This section describes the complete Rate-Halving algorithm. Most of the complexity of the specification is a result of the need for backwards compatibility with hosts that do not support SACK or ECN.

For all subsections of this section, additional discussion and explanation may be found in Section 5 under the same subsection number.

4.1 Rate-Halving states

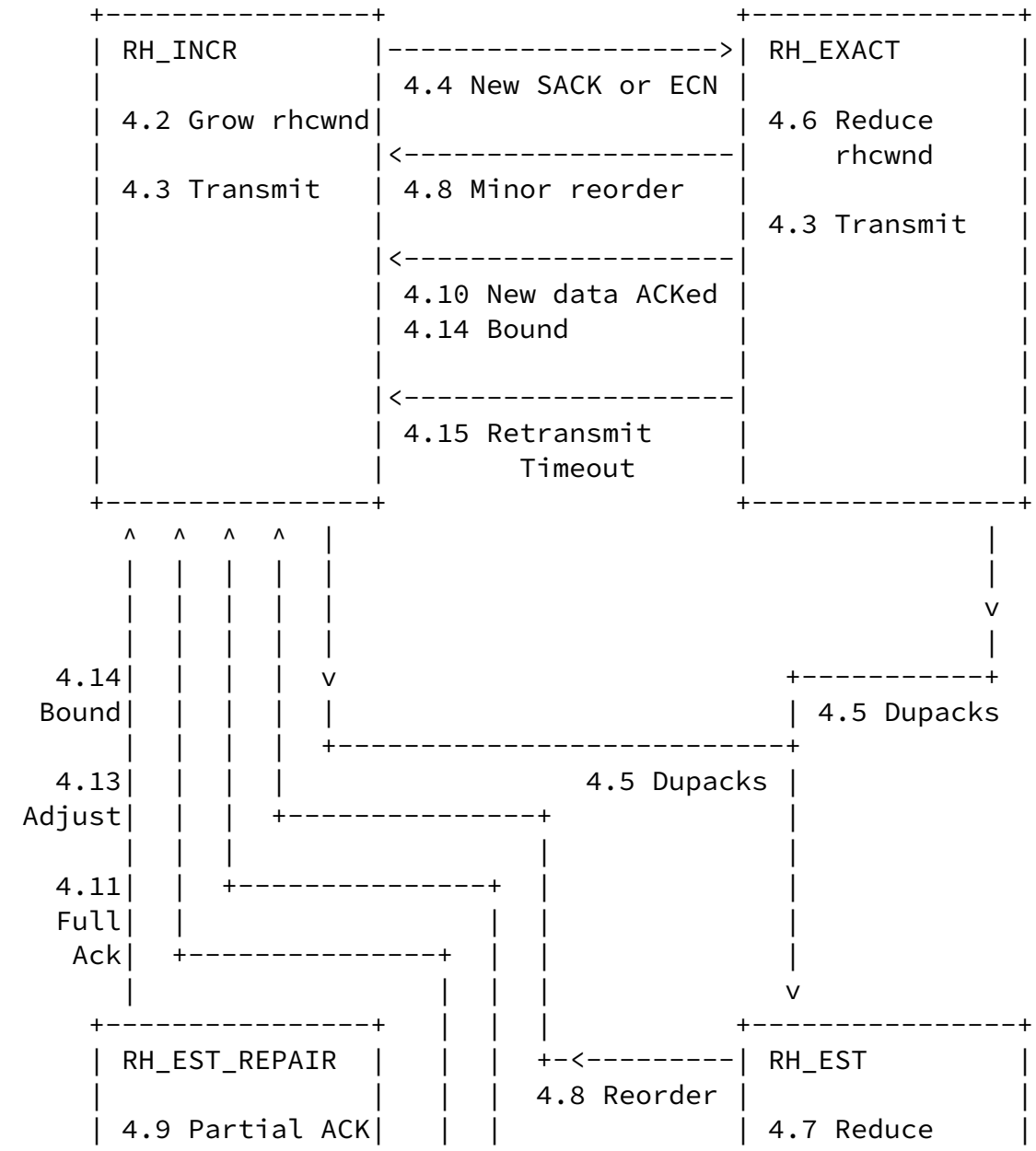
Rate-Halving is specified as a state machine with the following states:

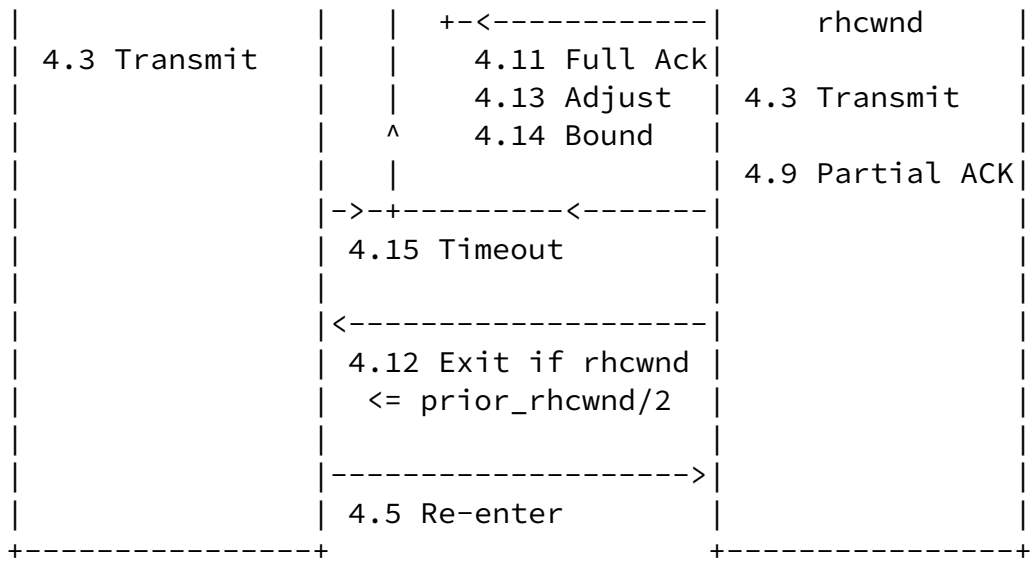
RH_INCR - Rate-Halving is not performing any downward window adjustment. This corresponds to TCP being in either Slow-start or the linear increase region.

RH_EXACT - Rate-Halving is reducing the window with accurate knowledge of what data has left the network. This state normally follows congestion indicated by ECN or SACK.

RH_EST - Rate-Halving is reducing the window while estimating the amount of data that has left the network. This state is analagous to Reno's Fast Recovery.

RH_EST_REPAIR - Rate-Halving is holding the window constant while repairing multiple holes in a manner similar to NewReno on successive round trips after the window adjustment has completed.





4.2 Increasing Rhcwnd

Upon receipt of a new ACK in RH_INCR, apply Congestion Avoidance or Slow-start per RFC2581 only when:

```
snd.nxt - fack + retran_data + SMSS >= rhcwnd
```

4.3 Data Transmission

While honoring all other sending constraints (such as receiver window), in any state, transmit data of length "len" if

```
(snd.nxt - fack + retran_data + len) < rhcwnd
```

If the data is a retransmission,

```
retran_data += len
```

4.4 Entering RH_EXACT State

Transition from RH_INCR to RH_EXACT if the ACK either carries an ECN-Echo bit, or indicates a new hole via a SACK option.

Store the following information:

```
prior_rhcwnd = rhcwnd
prior_max_seq = max_seq
```

4.5 Entering RH_EST State

From either RH_INCR or RH_EXACT, transition to RH_EST upon receipt of a dupack with no SACK options.

From the RH_EST_REPAIR state, transition to RH_EST upon receipt of an

ACK with the ECN-Echo bit set.

If the transition is from RH_INCR or RH_EST_REPAIR to RH_EST, store the following information:

```
prior_rhcwnd = rhcwnd
prior_max_seq = max_seq
```

4.6 Reduction of Rhcwnd in RH_EXACT State

For each ACK received while in state RH_EXACT, reduce rhcwnd by 1/2 the distance fack advances plus 1/2 the size of any new holes.

4.7 Reduction of Rhcwnd in RH_EST State

For each dupack received while in state RH_EST:

```
rhcwnd  -= SMSS/2
fack     = min((fack + SMSS), max_seq)
dupacks += 1
```

4.8 Detection of Reordering; Return to RH_INCR

Reordering detection applies only if an ECN-Echo has not been received and a retransmission has not been sent since the adjustment interval began.

In RH_EXACT, reordering has occurred if an ACK arrives carrying no ECN-Echo bit and no SACK blocks.

In RH_EST, reordering has occurred if a partial or full ACK arrives.

If reordering is detected, transition to RH_INCR and restore the value of rhcwnd, without performing the bounding checks of 4.14:

```
rhcwnd = prior_rhcwnd;
```

4.9 Processing Partial Acks in RH_EST / RH_EST_REPAIR

For each partial acknowledgement received in RH_EST or RH_EST_REPAIR:

1. Set retran_data to 0.
2. Reduce dupacks by [(bytes of data acked)/SMSS - 1].
3. For the purpose of data transmission, note that a new hole exists at snd.una. The new hole is eligible for retransmission if dupacks > 3, after being reduced in step 2.

4.10 Completion of RH_EXACT; Return to RH_INCR

In RH_EXACT, Rate Halving is complete if one of the following occurs (indicating that a round trip time has passed):

1. An ACK or SACK arrives which acknowledges data beyond prior_max_seq.

2. An ACK or SACK arrives which acknowledges a segment retransmitted after entering the RH_EXACT state.

Transition to state RH_INCR and perform the bounding checks in 4.14.

4.11 Completion of RH_EST and RH_EST_REPAIR; Return to RH_INCR

In state RH_EST or RH_EST_REPAIR, adjustment and repair are both complete when a full acknowledgement is received (i.e. snd.una >= prior_max_seq).

When this occurs, transition to state RH_INCR, perform the estimation adjustment in 4.13, followed by the bounding checks in 4.14.

4.12 "NewReno" Case: Transition from RH_EST to RH_EST_REPAIR

In state RH_EST, adjustment (but not repair) is complete when

```
rhcwnd <= prior_rhcwnd / 2
```

Transition from state RH_EST to state RH_EST_REPAIR.

4.13 Corrections for Estimation upon return to RH_INCR

When adjustment and repair are complete after an estimated recovery, perform the following adjustment to compute the correct value for rhcwnd:

```
rhcwnd = (prior_rhcwnd - num_retrans) / 2;
```

4.14 Application of Bounding Paremeters

When transitioning back to RH_INCR, update the TCP state variables as follows:

```
if (rhcwnd > prior_rhcwnd/2) rhcwnd = prior_rhcwnd/2;
ssthresh = rhcwnd;
if (ssthresh < prior_rhcwnd/4) ssthresh = prior_rhcwnd/4;
```

Check for a new condition that would cause entry into another adjustment interval (e.g. the presence of SACK blocks reporting a new hole (beyond prior_max_seq).

4.15 Retransmission Timeouts

If a timeout occurs while in a non-RH_INCR state, set

```
   ssthresh = prior_rhcwnd / 2
   rhcwnd = LW
```

and return to the RH_INCR.

Otherwise, if a timeout occurs while in RH_INCR, set

```
   ssthresh = rhcwnd /2
   rhcwnd = LW
```

5. Discussion of Rate-Halving Algorithm

Each of the subsections of Section 5 provide discussion for the corresponding subsection of Section 4.

5.1 Rate-Halving States

The Rate-Halving algorithm may be viewed as a finite state machine with four states. These states are independent of whether the TCP connection is in Congestion Avoidance or Slow-start.

5.2 Increasing Rhcwnd

The left hand term in 4.2 includes all outstanding transmitted and retransmitted segments that are still in the network, plus one additional segment to account for the effects of rounding due to segmentation.

This restriction assures that rhcwnd only grows as long as TCP actually succeeds in injecting enough data into the network to test the path. If TCP is throttled by something other than rhcwnd, then there is no assurance that rhcwnd of data will actually fit into the network.

This test should be done prior to taking any data which was ACKed or SACKed in the incoming segment into account. This may be accomplished either by performing this check immediately upon receiving an ACK, or by remembering the total amount of new and retransmitted data ACKed or SACKed by the segment, and adding this to the term on the left hand side.

Since it is never safe to open the window while estimating what data has left the network, Reno and NewReno can never open the window while dupacks are present - e.g during any adjustment or repair interval.

5.3 Data Transmission

Rhcwnd always specifies an amount of data allowed to be in flight in the network. The expression (snd.nxt - fack + retran_data + len) is a measure of the outstanding data. Data may be transmitted whenever the amount of outstanding data is less than the amount allowed by rhcwnd.

Contrast this to Reno, where cwnd is used as an offset to snd.una, specifying which sequence numbers may be sent. As a consequence the cwnd variable is overloaded during recovery (e.g. setting cwnd=SMSS to force Fast Retransmit.)

To illustrate that the new semantics for rhcwnd in Rate-Halving are compatible with those of cwnd in Reno, observe that in Reno the boundary between being able to send data and not send data is:

```
   snd.nxt = snd.una + cwnd
```

where
```
   cwnd = rhcwnd - retran_data + dupacks*SMSS
```
and
```
   retran_data = 0 or 1 depending on whether a retransmission has
                 been sent.
```

Substituting and rearranging the Reno equations yields equation 1:

```
   (1) rhcwnd = snd.nxt - snd.una - dupacks*SMSS + retran_data
```

In Rate-Halving, the boundary occurs at:

```
   awnd = rhcwnd,
```

where
```
   awnd = snd.nxt - fack + retran_data
```
and
```
   fack = snd.una + losses + SACKed data
```

Substituting and rearranging the Rate-Halving equations yields:

```
   (2) rhcwnd = snd.nxt - snd.una - losses - SACKed data +
                                             retran_data
```

It can be seen that equations 1 and 2 are quite similar. The losses known by Rate-Halving (via SACK options) are not known to Reno. In addition, the amount of data that has left the network (indicated by SACK options to Rate-Halving) must be estimated by Reno as dupacks*SMSS. Reno only allows one retransmitted segment per round trip, while Rate-Halving permits (and keeps track of) more.

When SACK options are not used by the receiver, the Rate-Halving sender must estimate by using equation 1.

Similar analysis can be applied to The "pipe" algorithm used for SACK in the "ns" [NS] simulator and some SACK implementations.

5.4 Entering RH_EXACT State

Note that the sender can infer congestion by receiving a segment with the ECN-Echo bit set, or by receiving an Ack carrying a SACK option which indicates a new hole. In both of these cases, the amount of data that has left the network is known, either because no data was lost, or the amount of data that was lost is reported by SACK options.

In the case of a suspected loss, enter the adjustment interval immediately. While this may seem to be a major change to TCP, in fact we retain the "3 Dupacks" check (and make it more robust) which is included in Reno TCP. Segment retransmissions are determined by the SACK scoreboard, as discussed in section 6.2.1. During the early portion of the adjustment interval, one new segment will generally be transmitted prior to retransmitting the lost segment which triggered the adjustment.

The algorithm will detect if the segments have merely been reordered, and the adjustment is terminated (see section 5.8 below).

Prior_rhcwnd and prior_max_seq will be used for end-of-adjustment checks.

5.5 Entering RH_EST State

The significance of this state is that the amount of data in the network is estimated, since exact information is not available through SACK options.

If the prior state was RH_EST_REPAIR, then this is a new reduction triggered by detecting a new ECN prior to the completion of a NewReno-style repair.

The sender can also infer congestion by receiving duplicate acknowledgements. In this case, it is known that a segment was lost or misordered, but the number of missing segments is not known. The number can only be estimated by counting the number of duplicate ACKs received. Even if RH_EXACT was entered with an ECN-Echo, the RH_EST state can be entered from RH_EXACT if a dupack is later received.

5.6 Reduction of Rhcwnd in RH_EXACT State

Rather than counting received ACKs, the distance (in bytes) that fack advances is explicitly used. This ensures that the window is

properly reduced even when the receiver is sending delayed ACKs during congestion episodes (such as with ECN).

Each SACK option that indicates a new hole triggers a reduction of rhcwnd by all of data that was lost plus 1/2 the data that was received.

During RH, we reduce rhcwnd as data leaves the network. The net effect is to cause the transmission of one new segment for every two segments reported by the receiver.

5.7 Reduction of Rhcwnd in RH_EST State

Assume that each dupack is for a segment of size SMSS, and that the rate is being halved. This estimate is designed to result in rhcwnd and fack values that are reasonable, given that more exact information is not available about which segments have left the network. These estimates are comparable to (and no worse than) the estimates that Reno uses during Fast Recovery. When the repair interval ends, final adjustments recalibrate rhcwnd and fack to their correct values.

5.8 Detection of Reordering; Return to RH_INCR

Note that there is a choice of what do with the congestion window when segment reordering is detected. In principle, rhcwnd could be set to an arbitrary function of prior_rhcwnd.

For example, one could choose to penalize networks which reorder segments by forcibly reducing rhcwnd:

```
rhcwnd = prior_rhcwnd / 2;
```

A gentler form of reordering penalty would be to leave rchwnd as updated by Rate-Halving (Reduce rhcwnd as indicated in section 4.6 or 4.7 until the reordering has been detected).

The choice to restore rhcwnd results in no penalty for reordering:

```
rhcwnd = prior_rhcwnd;
```

This is functionally equivalent to today's Reno implementations, and may be less bursty in some cases. (RH is less bursty in cases in which a single new segment is transmitted prior to detecting the reordering, resulting in a burst which is one segment smaller than the equivalent Reno burst).

The optimal action is probably between these extremes, and should be the subject of future research.

5.9 Processing Partial Acks in RH_EST / RH_EST_REPAIR

A partial ACK indicates that one hole has been filled (since it is now covered by an ACK) and that another one exists. In step 1, retran_data is cleared since the retransmitted segments left the network when the partial ACK was generated.

Some of the segments that generated duplicate ACKs are acknowledged by the partial ACK. These segments must be accounted for in the count of dupacks, so step 2 reduces dupacks by the amount of data that has just been acknowledged.

A new hole is indicated by the partial ack one round trip after the retransmission for the previous hole was sent. The only way in which the segment indicated by the partial ack would be falsely retransmitted would be if the segment was reordered (delayed) by more than a round trip time.

If the previous hole was caused by a reordered segment, then the partial ack may appear without sending a retransmission (and with dupacks < 3). In this case, the new hole is not eligible for retransmission until dupacks >= 3.

5.10 Completion of RH_EXACT; Return to RH_INCR

Under severe reordering (after retransmission) condition 2 may cause a premature exit from the adjustment interval. This would leave rhcwnd too large. However the bounding check in 4.14 will force the required 1/2 window reduction.

Case 2 might be particularly tricky to implement for cases in which SACK, ECN, and NewReno are all supported. Here are some guidelines for implementation:

ACKs or SACKs which show the arrival of retransmitted data may not be reliable indicators of case 2. This is because it is possible (and even likely) for the data repair interval to continue well beyond the end of the adjustment interval. Should a second adjustment interval begin as a result of subsequent lost data, it is possible for holes to be filled which don't satisfy case 2. Therefore, a more rigorous test is needed for retransmitted segments.

Assuming that there is a SACK scoreboard, one possible implementation is to store an adjustment interval ID number with each retransmission. This ID will clearly indicate during which adjustment interval a retransmission occurred. (The ID may also be "zero" indicating the retransmissions occurred outside of an adjustment interval). ACKs and SACKs which are received for retransmissions can then be definitively tested against condition 2.

5.11 Completion of RH_EST and RH_EST_REPAIR; Return to RH_INCR

A full ack indicates that all segments that were outstanding at the first indication of congestion have been successfully received. Therefore, the adjustment and repair intervals should be completed, but final adjustments and checks must be made to ensure the validity of state variables.

5.12 "NewReno" Case: Transition from RH_EST to RH_EST_REPAIR

When the window has been reduced by half, but a full ack has not yet been received, it is necessary to end the adjustment interval, holding the window constant while the repair interval completes.

In the RH_EST_REPAIR state, rhcwnd is not increased or decreased. While in this state, retransmissions will be sent once per round trip time, as in NewReno until a full ACK ends the repair interval.

5.13 Corrections for Estimation upon return to RH_INCR

The final adjustment ensures that the resulting congestion window is half of the data that successfully passed through the network during the congested round trip.

5.14 Application of Bounding Paremeters

At the end of the RH_EXACT state, rhcwnd has been set to (prior_rhcwnd-loss)/2, where loss is the number of bytes of data that have been lost during the current adjustment interval.

In some cases, it is possible for rhcwnd to be set to an invalid result (such as severe reordering mentioned above or improper acknowledgement strategies of the receiver).

Rate-Halving includes a set of Bounding Parameters which are used to guarantee that the Rate-Halving algorithm will make appropriate adjustments to the window. The prior_rhcwnd/2 check guarantees that, no matter what else happens, rhcwnd will always be reduced by a factor of two.

The prior_rhcwnd/4 check ensures that the TCP connection is not unduly harmed by extreme network conditions. The choice to set ssthresh (rather than rhcwnd) to prior_rhcwnd/4 prevents sending a burst into the network as a result of suddenly increasing rhcwnd. A short period of Slow-start will follow, allowing rhcwnd to quickly reach the bounding parameter of prior_rhcwnd/4.

Note that it is possible to exit the adjustment interval on an ACK, and re-enter a new adjustment interval as a result of the same acknowledgement. For example, an ACK may carry both acknowledgement of retransmitted data, causing an exit per 4.10 case 2, and an ECN-Echo bit (indicating that the retransmitted segment was marked),

causing a new adjustment interval per 4.4.

5.15 Retransmission Timeouts

Timeout behavior is nearly identical to Reno. Since rhcwnd is reduced during non-RH_INCR states, it is necessary to use prior_rhcwnd to prevent an overly agressive adjustment in the ssthresh calculations.

See section 6.2.2 for information about SACK reneging (SACK receiver discards data that it has already sent SACK blocks for [RFC2018]).

Do not clear the exponential backoff multiplier after the timeout if the ECN-Echo bit is set on the ACK that results from the retransmission.

6. Additional Implementation Notes

6.1 ACK processing

6.1.1 Order of actions to be taken

Upon receiving an acknowledgement, the following actions should be taken:

1. Save a copy of the following state variables for use during processing of this ACK: snd.una, fack, retran_data.
2. Update the scoreboard, snd.una, fack, and retran_data.
3. Note if the ECN-Echo bit is set.
4. Check for exit conditions from RH states (4.8, 4.10, 4.11).
5. Check for transition to RH_EST_REPAIR state (4.12).
6. Process partial ACKs (4.9).
7. Check for entrance conditions for RH (4.4, 4.5).
8. Adjust the window using state variables as they existed at the time the ACK was received (4.2, 4.6, 4.7).
9. If the window allows (4.3), send a new segment or a retransmission as determined by the scoreboard (6.2.1).

6.1.2 Setting FACK

When in the RH_EXACT or RH_INCR state:
When each ACK is received, set the fack state variable to be the maximum of: (1) the highest sequence number which has been acknowledged plus one, and (2) the highest sequence number which has appeared in a SACK block plus one.

When in the RH_EST or RH_EST_REPAIR state:
When each ACK is received, set the fack state variable to be

snd_una + (1 + dupacks) * SMSS.

6.2 Data Recovery

6.2.1 SACK Scoreboard

A SACK scoreboard keeps track of data blocks that have been received after some data has been lost, as indicated by SACK options. The data transmission rule of 4.3 indicates only WHEN to send data (based on congestion control), not WHAT data to send (based on reliable delivery). The scoreboard keeps track of what data has been received, and indicates what data is eligible for retransmission.

Reno TCP uses a threshold of 3 dupacks to determine that a missing segment has been lost, not just reordered. By immediately entering the adjustment interval, not only can a Rate-Halving implementation apply the dupack threshold to the first loss, but also to all subsequent losses. Mathis and Mahdavi presented a method for implementing this logic, termed thresholded retransmissions, in the scoreboard [MM97]. A data block is eligible for retransmission only after the hole has been observed at least three times, as indicated by dupacks, or SACK options with sequence numbers that are higher than the data block in question.

If no data is eligible for retransmission, new data may be sent when allowed by the congestion window and the receiver's advertised window (4.3).

When no SACK information is available, such as in states RH_EST or RH_EST_REPAIR, a single-element scoreboard may be used to store loss and retransmission state information.

6.2.2 Retransmission timeout/SACK reneging

RFC2018 recommends (with a SHOULD) that the scoreboard be cleared when a timeout is experienced, since the timeout might indicate that the SACK receiver may have reneged (discarded data that it has already sent a SACK option for). If this RFC2018 recommendation is followed, then perform the following actions in addition to the actions specified in 4.15:

```
snd.nxt = snd.una
fack    = snd.una
retran_data = 0
```

It is believed to be safe to relax the SHOULD in RFC2018, such that when a retransmission timeout is experienced, the scoreboard can be kept in order to retain the information about which segments have already been received. When allowing the scoreboard to be maintained through a timeout (conflicting with RFC2018), the following actions must be performed in addition to the actions specified in 4.15:

```
   snd.nxt = fack
   retran_data = 0
```

In this case, snd.nxt is pulled back to fack, which is the highest sequence number known to have reached the receiver. Normal transmissions will begin at this sequence number, and retransmissions will continue to be issued by the scoreboard. (Following the timeout, fack will advance to the highest ACK or SACK block received. snd.nxt should be pulled ahead any time that fack > snd.nxt.)

If the scoreboard is not discarded during a timeout, the sender must be able to detect that the receiver has reneged. If snd.una advances to a block stored in the scoreboard, the sender knows that a SACK renege has occurred, and must clear the scoreboard, setting

```
   fack = snd.una
   snd.nxt = snd.una
   retran_data = 0
```

There are three options for what should happen to the window when the sender detects that the receiver has reneged: 1) force a retransmit timeout, 2) cut the window in half, or 3) do nothing to the window. More experimentation is needed to determine the correct approach.

7. Acknowledgements

The authors would like to acknowledge Janie Hoe's work, which introduced the concepts of sending new data during recovery and spacing out data segments during the round trip following a loss. The authors would also like to thank Van Jacobson and Sally Floyd for inspirational conversations about many of the ideas in this document. In addition, the authors are grateful to Mark Allman, Jitendra Padhye, and Khrys Myrddin for their constructive feedback about earlier drafts of this document.

8. References

[Hoe95] J. Hoe, Startup Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's Thesis, MIT, 1995. URL "http://ana-www.lcs.mit.edu/anaweb/ps-papers/hoe-thesis.ps".

[Hoe96] J. Hoe, Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In ACM SIGCOMM, August 1996. URL "http://www.acm.org/sigcomm/sigcomm96/program.html".

[MM96] M. Mathis, J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control," SIGCOMM 96, August 1996.

[MM97] M. Mathis, J. Mahdavi, "TCP Rate-Halving with Bounding Parameters," December 1997.
URL "http://www.psc.edu/networking/papers/FACKnotes/current/".

[NS] The UCB/LBNL/VINT Network Simulator (NS). URL "http://www-mash.cs.berkeley.edu/ns/".

[RFC793] J. Postel, "Transmission Control Protocol," RFC793, September 1981.

[RFC1191] J. Mogul, and S. Deering, "Path MTU Discovery," RFC1191, November 1990.

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

[RFC2481] K. Ramakrishnan, and S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP," RFC2481, January 1999.

[RFC2581] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC2581, April 1999.

[RFC2582] S. Floyd, and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC2582, April 1999.

9. Security Considerations

RFC 2581 discusses general security considerations concerning TCP congestion control. This document describes a specific algorithm that conforms with the congestion control requirements of RFC 2581, and so those considerations apply to this algorithm, too. There are no known additional security concerns for this specific algorithm.

AUTHORS' ADDRESSES

Matthew Mathis and Jeffrey Semke
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA 15213

EMail: mathis@psc.edu, semke@psc.edu

Jamshid Mahdavi
Novell, Inc.
Mailstop: SJF-B492
2211 North First Street
San Jose, CA 95131

EMail: mahdavi@novell.com, kml@novell.com