Network Working Group
Internet-Draft
Obsoletes: 3782 (if approved)
Intended status: Standards Track
Expires: June 24, 2011

T. Henderson
Boeing
S. Floyd
ICSI
A. Gurtov
HIIT
Y. Nishida
WIDE Project

January 24, 2011

The NewReno Modification to TCP's Fast Recovery Algorithm
draft-ietf-tcpm-rfc3782-bis-00.txt

Abstract

RFC 5681 [RFC5681] documents the following four intertwined TCP congestion control algorithms: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. RFC 5681 explicitly allows certain modifications of these algorithms, including modifications that use the TCP Selective Acknowledgement (SACK) option [RFC2883], and modifications that respond to "partial acknowledgments" (ACKs which cover new data, but not all the data outstanding when loss was detected) in the absence of SACK. This document describes a specific algorithm for responding to partial acknowledgments, referred to as NewReno. This response to partial acknowledgments was first proposed by Janey Hoe in [Hoe95].

The purpose of this revision from [RFC3782] is to make errata changes and to adopt a proposal from Yoshifumi Nishida to slightly increase the minimum window size after Fast Recovery from one to two segments, to improve performance when the receiver uses delayed acknowledgments.

This Internet-Draft will expire on June 24, 2011.

Copyright Notice

1. Introduction

For the typical implementation of the TCP Fast Recovery algorithm described in [RFC5681] (first implemented in the 1990 BSD Reno release, and referred to as the Reno algorithm in [FF96]), the TCP data sender only retransmits a packet after a retransmit timeout has occurred, or after three duplicate acknowledgements have arrived triggering the Fast Retransmit algorithm. A single retransmit timeout might result in the retransmission of several data packets, but each invocation of the Fast Retransmit algorithm in RFC 5681 leads to the retransmission of only a single data packet.

Problems can arise, therefore, when multiple packets are dropped from a single window of data and the Fast Retransmit and Fast Recovery algorithms are invoked. In this case, if the SACK option is available, the TCP sender has the information to make intelligent decisions about which packets to retransmit and which packets not to retransmit during Fast Recovery. This document applies only for TCP connections that are unable to use the TCP Selective Acknowledgement (SACK) option, either because the option is not locally supported or because the TCP peer did not indicate a willingness to use SACK.

In the absence of SACK, there is little information available to the TCP sender in making retransmission decisions during Fast Recovery. From the three duplicate acknowledgements, the sender infers a packet loss, and retransmits the indicated packet. After this, the data sender could receive additional duplicate acknowledgements, as the data receiver acknowledges additional data packets that were already in flight when the sender entered Fast Retransmit.

In the case of multiple packets dropped from a single window of data, the first new information available to the sender comes when the sender receives an acknowledgement for the retransmitted packet (that is, the packet retransmitted when Fast Retransmit was first entered). If there is a single packet drop and no reordering, then the acknowledgement for this packet will acknowledge all of the packets transmitted before Fast Retransmit was entered. However, if there are multiple packet drops, then the acknowledgement for the retransmitted packet will acknowledge some but not all of the packets transmitted before the Fast Retransmit. We call this acknowledgement a partial acknowledgment.

Along with several other suggestions, [Hoe95] suggested that during Fast Recovery the TCP data sender responds to a partial acknowledgment by inferring that the next in-sequence packet has been lost, and retransmitting that packet. This document describes a modification to the Fast Recovery algorithm in RFC 5681 that

incorporates a response to partial acknowledgements received during Fast Recovery. We call this modified Fast Recovery algorithm NewReno, because it is a slight but significant variation of the basic Reno algorithm in RFC 5681. 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 proposal to send a new packet for every two duplicate acknowledgements during Fast Recovery. The version of NewReno in this document also draws on other discussions of NewReno in the literature [LM97, Hen98].

We do not claim that the NewReno version of Fast Recovery described here is an optimal modification of Fast Recovery for responding to partial acknowledgements, for TCP connections that are unable to use SACK. Based on our experiences with the NewReno modification in the NS simulator [NS] and with numerous implementations of NewReno, we believe that this modification improves the performance of the Fast Retransmit and Fast Recovery algorithms in a wide variety of scenarios.

2. Terminology and Definitions

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119]. This RFC indicates requirement levels for compliant TCP implementations implementing the NewReno Fast Retransmit and Fast Recovery algorithms described in this document.

This document assumes that the reader is familiar with the terms SENDER MAXIMUM SEGMENT SIZE (SMSS), CONGESTION WINDOW (cwnd), and FLIGHT SIZE (FlightSize) defined in [RFC5681]. FLIGHT SIZE is defined as in [RFC5681] as follows:

FLIGHT SIZE:
The amount of data that has been sent but not yet cumulatively acknowledged.

3. The Fast Retransmit and Fast Recovery Algorithms in NewReno

The standard implementation of the Fast Retransmit and Fast Recovery algorithms is given in [RFC5681]. This section specifies the basic NewReno algorithm. Sections 4 through 6 describe some optional variants, and the motivations behind them, that an implementor may want to consider when tuning performance for certain network scenarios. Sections 7 and 8 provide some guidance to implementors based on experience with NewReno implementations.

The NewReno modification concerns the Fast Recovery procedure that

begins when three duplicate ACKs are received and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to and including the data that was outstanding when the Fast Recovery procedure began.

The NewReno algorithm specified in this document differs from the implementation in [RFC5681] in the introduction of the variable "recover" in step 1, in the response to a partial or new acknowledgement in step 5, and in modifications to step 1 and the addition of step 6 for avoiding multiple Fast Retransmits caused by the retransmission of packets already received by the receiver.

The algorithm specified in this document uses a variable "recover", whose initial value is the initial send sequence number.

1) Three duplicate ACKs:
   When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, check to see if the Cumulative Acknowledgement field covers more than "recover". If so, go to Step 1A. Otherwise, go to Step 1B.

1A) Invoking Fast Retransmit:
   If so, then set ssthresh to no more than the value given in equation 1 below. (This is equation 4 from [RFC5681]).

```
ssthresh = max (FlightSize / 2, 2*SMSS)           (1)
```

   In addition, record the highest sequence number transmitted in the variable "recover", and go to Step 2.

1B) Not invoking Fast Retransmit:
   Do not enter the Fast Retransmit and Fast Recovery procedure. In particular, do not change ssthresh, do not go to Step 2 to retransmit the "lost" segment, and do not execute Step 3 upon subsequent duplicate ACKs.

2) Entering Fast Retransmit:
   Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and the receiver has buffered.

3) Fast Recovery:
   For each additional duplicate ACK received while in Fast Recovery, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.

4) Fast Recovery, continued:
Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.

5) When an ACK arrives that acknowledges new data, this ACK could be the acknowledgment elicited by the retransmission from step 2, or elicited by a later retransmission.

Full acknowledgements:
If this ACK acknowledges all of the data up to and including "recover", then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK. Set cwnd to either (1) min (ssthresh, max(FlightSize, SMSS) + SMSS) or (2) ssthresh, where ssthresh is the value set in step 1; this is termed "deflating" the window. (We note that "FlightSize" in step 1 referred to the amount of data outstanding in step 1, when Fast Recovery was entered, while "FlightSize" in step 5 refers to the amount of data outstanding in step 5, when Fast Recovery is exited.) If the second option is selected, the implementation is encouraged to take measures to avoid a possible burst of data, in case the amount of data outstanding in the network is much less than the new congestion window allows. A simple mechanism is to limit the number of data packets that can be sent in response to a single acknowledgement; this is known as "maxburst_" in the NS simulator. Exit the Fast Recovery procedure.

Partial acknowledgements:
If this ACK does *not* acknowledge all of the data up to and including "recover", then this is a partial ACK. In this case, retransmit the first unacknowledged segment. Deflate the congestion window by the amount of new data acknowledged by the cumulative acknowledgement field. If the partial ACK acknowledges at least one SMSS of new data, then add back SMSS bytes to the congestion window. As in Step 3, this artificially inflates the congestion window in order to reflect the additional segment that has left the network. Send a new segment if permitted by the new value of cwnd. This "partial window deflation" attempts to ensure that, when Fast Recovery eventually ends, approximately ssthresh amount of data will be outstanding in the network. Do not exit the Fast Recovery procedure (i.e., if any duplicate ACKs subsequently arrive, execute Steps 3 and 4 above).

For the first partial ACK that arrives during Fast Recovery, also reset the retransmit timer. Timer management is discussed in more detail in Section 4.

6) Retransmit timeouts:
After a retransmit timeout, record the highest sequence number transmitted in the variable "recover" and exit the Fast Recovery procedure if applicable.

Step 1 specifies a check that the Cumulative Acknowledgement field covers more than "recover". Because the acknowledgement field contains the sequence number that the sender next expects to receive, the acknowledgement "ack_number" covers more than "recover" when:

```
ack_number - 1 > recover;
```

i.e., at least one byte more of data is acknowledged beyond the highest byte that was outstanding when Fast Retransmit was last entered.

Note that in Step 5, the congestion window is deflated after a partial acknowledgement is received. The congestion window was likely to have been inflated considerably when the partial acknowledgement was received. In addition, depending on the original pattern of packet losses, the partial acknowledgement might acknowledge nearly a window of data. In this case, if the congestion window was not deflated, the data sender might be able to send nearly a window of data back-to-back.

This document does not specify the sender's response to duplicate ACKs when the Fast Retransmit/Fast Recovery algorithm is not invoked. This is addressed in other documents, such as those describing the Limited Transmit procedure [RFC3042]. This document also does not address issues of adjusting the duplicate acknowledgement threshold, but assumes the threshold specified in the IETF standards; the current standard is RFC 5681, which specifies a threshold of three duplicate acknowledgements.

As a final note, we would observe that in the absence of the SACK option, the data sender is working from limited information. When the issue of recovery from multiple dropped packets from a single window of data is of particular importance, the best alternative would be to use the SACK option.

4. Resetting the Retransmit Timer in Response to Partial Acknowledgements

One possible variant to the response to partial acknowledgements specified in Section 3 concerns when to reset the retransmit timer after a partial acknowledgement. The algorithm in Section 3, Step 5, resets the retransmit timer only after the first partial ACK. In this case, if a large number of packets were dropped from a window of

data, the TCP data sender's retransmit timer will ultimately expire, and the TCP data sender will invoke Slow-Start. (This is illustrated on page 12 of [F98].) We call this the Impatient variant of NewReno. We note that the Impatient variant in Section 3 doesn't follow the recommended algorithm in RFC 2988 of restarting the retransmit timer after every packet transmission or retransmission (step 5.1 of [RFC2988]).

In contrast, the NewReno simulations in [FF96] illustrate the algorithm described above with the modification that the retransmit timer is reset after each partial acknowledgement. We call this the Slow-but-Steady variant of NewReno. In this case, for a window with a large number of packet drops, the TCP data sender retransmits at most one packet per roundtrip time. (This behavior is illustrated in the New-Reno TCP simulation of Figure 5 in [FF96], and on page 11 of [F98]).

When N packets have been dropped from a window of data for a large value of N, the Slow-but-Steady variant can remain in Fast Recovery for N round-trip times, retransmitting one more dropped packet each round-trip time; for these scenarios, the Impatient variant gives a faster recovery and better performance. The tests "ns test-suite-newreno.tcl impatient1" and "ns test-suite-newreno.tcl slow1" in the NS simulator illustrate such a scenario, where the Impatient variant performs better than the Slow-but-Steady variant. The Impatient variant can be particularly important for TCP connections with large congestion windows, as illustrated by the tests "ns test-suite-newreno.tcl impatient4" and "ns test-suite-newreno.tcl slow4" in the NS simulator.

One can also construct scenarios where the Slow-but-Steady variant gives better performance than the Impatient variant. As an example, this occurs when only a small number of packets are dropped, the RTO is sufficiently small that the retransmit timer expires, and performance would have been better without a retransmit timeout. The tests "ns test-suite-newreno.tcl impatient2" and "ns test-suite-newreno.tcl slow2" in the NS simulator illustrate such a scenario.

The Slow-but-Steady variant can also achieve higher goodput than the Impatient variant, by avoiding unnecessary retransmissions. This could be of special interest for cellular links, where every transmission costs battery power and money. The tests "ns test-suite-newreno.tcl impatient3" and "ns test-suite-newreno.tcl slow3" in the NS simulator illustrate such a scenario. The Slow-but-Steady variant can also be more robust to delay variation in the network, where a delay spike might force the Impatient variant into a timeout and go-back-N recovery.

Neither of the two variants discussed above are optimal. Our recommendation is for the Impatient variant, as specified in Section 3 of this document, because of the poor performance of the Slow-but-Steady variant for TCP connections with large congestion windows.

One possibility for a more optimal algorithm would be one that recovered from multiple packet drops as quickly as does slow-start, while resetting the retransmit timers after each partial acknowledgement, as described in the section below. We note, however, that there is a limitation to the potential performance in this case in the absence of the SACK option.

5. Retransmissions after a Partial Acknowledgement

One possible variant to the response to partial acknowledgements specified in Section 3 would be to retransmit more than one packet after each partial acknowledgement, and to reset the retransmit timer after each retransmission. The algorithm specified in Section 3 retransmits a single packet after each partial acknowledgement. This is the most conservative alternative, in that it is the least likely to result in an unnecessarily-retransmitted packet. A variant that would recover faster from a window with many packet drops would be to effectively Slow-Start, retransmitting two packets after each partial acknowledgement. Such an approach would take less than N roundtrip times to recover from N losses [Hoe96]. However, in the absence of SACK, recovering as quickly as slow-start introduces the likelihood of unnecessarily retransmitting packets, and this could significantly complicate the recovery mechanisms.

We note that the response to partial acknowledgements specified in Section 3 of this document and in RFC 2582 differs from the response in [FF96], even though both approaches only retransmit one packet in response to a partial acknowledgement. Step 5 of Section 3 specifies that the TCP sender responds to a partial ACK by deflating the congestion window by the amount of new data acknowledged, adding back SMSS bytes if the partial ACK acknowledges at least SMSS bytes of new data, and sending a new segment if permitted by the new value of cwnd. Thus, only one previously-sent packet is retransmitted in response to each partial acknowledgement, but additional new packets might be transmitted as well, depending on the amount of new data acknowledged by the partial acknowledgement. In contrast, the variant of NewReno illustrated in [FF96] simply set the congestion window to ssthresh when a partial acknowledgement was received. The approach in [FF96] is more conservative, and does not attempt to accurately track the actual number of outstanding packets after a partial acknowledgement is received. While either of these approaches gives acceptable performance, the variant specified in

Section 3 recovers more smoothly when multiple packets are dropped from a window of data. (The [FF96] behavior can be seen in the NS simulator by setting the variable "partial_window_deflation_" for "Agent/TCP/Newreno" to 0; the behavior specified in Section 3 is achieved by setting "partial_window_deflation_" to 1.)

6. Avoiding Multiple Fast Retransmits

This section describes the motivation for the sender's state variable "recover", and discusses possible heuristics for distinguishing between a retransmitted packet that was dropped, and three duplicate acknowledgements from the unnecessary retransmission of three packets.

In the absence of the SACK option or timestamps, a duplicate acknowledgement carries no information to identify the data packet or packets at the TCP data receiver that triggered that duplicate acknowledgement. In this case, the TCP data sender is unable to distinguish between a duplicate acknowledgement that results from a lost or delayed data packet, and a duplicate acknowledgement that results from the sender's unnecessary retransmission of a data packet that had already been received at the TCP data receiver. Because of this, with the Retransmit and Fast Recovery algorithms in Reno TCP, multiple segment losses from a single window of data can sometimes result in unnecessary multiple Fast Retransmits (and multiple reductions of the congestion window) [F94].

With the Fast Retransmit and Fast Recovery algorithms in Reno TCP, the performance problems caused by multiple Fast Retransmits are relatively minor compared to the potential problems with Tahoe TCP, which does not implement Fast Recovery. Nevertheless, unnecessary Fast Retransmits can occur with Reno TCP unless some explicit mechanism is added to avoid this, such as the use of the "recover" variable. (This modification is called "bugfix" in [F98], and is illustrated on pages 7 and 9 of that document. Unnecessary Fast Retransmits for Reno without "bugfix" is illustrated on page 6 of [F98].)

Section 3 of [RFC2582] defined a default variant of NewReno TCP that did not use the variable "recover", and did not check if duplicate ACKs cover the variable "recover" before invoking Fast Retransmit. With this default variant from RFC 2582, the problem of multiple Fast Retransmits from a single window of data can occur after a Retransmit Timeout (as in page 8 of [F98]) or in scenarios with reordering (as in the validation test "./test-all-newreno newreno5_noBF" in directory "tcl/test" of the NS simulator. This gives performance similar to that on page 8 of [F03].) RFC 2582 also defined Careful and Less Careful variants of the NewReno algorithm, and recommended

the Careful variant.

The algorithm specified in Section 3 of this document corresponds to the Careful variant of NewReno TCP from RFC 2582, and eliminates the problem of multiple Fast Retransmits. This algorithm uses the variable "recover", whose initial value is the initial send sequence number. After each retransmit timeout, the highest sequence number transmitted so far is recorded in the variable "recover".

If, after a retransmit timeout, the TCP data sender retransmits three consecutive packets that have already been received by the data receiver, then the TCP data sender will receive three duplicate acknowledgements that do not cover more than "recover". In this case, the duplicate acknowledgements are not an indication of a new instance of congestion. They are simply an indication that the sender has unnecessarily retransmitted at least three packets.

However, when a retransmitted packet is itself dropped, the sender can also receive three duplicate acknowledgements that do not cover more than "recover". In this case, the sender would have been better off if it had initiated Fast Retransmit. For a TCP that implements the algorithm specified in Section 3 of this document, the sender does not infer a packet drop from duplicate acknowledgements in this scenario. As always, the retransmit timer is the backup mechanism for inferring packet loss in this case.

There are several heuristics, based on timestamps or on the amount of advancement of the cumulative acknowledgement field, that allow the sender to distinguish, in some cases, between three duplicate acknowledgements following a retransmitted packet that was dropped, and three duplicate acknowledgements from the unnecessary retransmission of three packets [Gur03, GF04]. The TCP sender MAY use such a heuristic to decide to invoke a Fast Retransmit in some cases, even when the three duplicate acknowledgements do not cover more than "recover".

For example, when three duplicate acknowledgements are caused by the unnecessary retransmission of three packets, this is likely to be accompanied by the cumulative acknowledgement field advancing by at least four segments. Similarly, a heuristic based on timestamps uses the fact that when there is a hole in the sequence space, the timestamp echoed in the duplicate acknowledgement is the timestamp of the most recent data packet that advanced the cumulative acknowledgement field [RFC1323]. If timestamps are used, and the sender stores the timestamp of the last acknowledged segment, then the timestamp echoed by duplicate acknowledgements can be used to distinguish between a retransmitted packet that was dropped and three duplicate acknowledgements from the unnecessary

retransmission of three packets. The heuristics are illustrated in the NS simulator in the validation test "./test-all-newreno".

6.1. ACK Heuristic

If the ACK-based heuristic is used, then following the advancement of the cumulative acknowledgement field, the sender stores the value of the previous cumulative acknowledgement as prev_highest_ack, and stores the latest cumulative ACK as highest_ack. In addition, the following step is performed if Step 1 in Section 3 fails, before proceeding to Step 1B.

1*) If the Cumulative Acknowledgement field didn't cover more than "recover", check to see if the congestion window is greater than SMSS bytes and the difference between highest_ack and prev_highest_ack is at most 4*SMSS bytes. If true, duplicate ACKs indicate a lost segment (proceed to Step 1A in Section 3). Otherwise, duplicate ACKs likely result from unnecessary retransmissions (proceed to Step 1B in Section 3).

The congestion window check serves to protect against fast retransmit immediately after a retransmit timeout, similar to the "exitFastRetrans_" variable in NS. Examples of applying the ACK heuristic are in validation tests "./test-all-newreno newreno_rto_loss_ack" and "./test-all-newreno newreno_rto_dup_ack" in directory "tcl/test" of the NS simulator.

If several ACKs are lost, the sender can see a jump in the cumulative ACK of more than three segments, and the heuristic can fail. A validation test for this scenario is "./test-all-newreno newreno_rto_loss_ackf". RFC 5681 recommends that a receiver should send duplicate ACKs for every out-of-order data packet, such as a data packet received during Fast Recovery. The ACK heuristic is more likely to fail if the receiver does not follow this advice, because then a smaller number of ACK losses are needed to produce a sufficient jump in the cumulative ACK.

6.2. Timestamp Heuristic

If this heuristic is used, the sender stores the timestamp of the last acknowledged segment. In addition, the second paragraph of step 1 in Section 3 is replaced as follows:

1**) If the Cumulative Acknowledgement field didn't cover more than "recover", check to see if the echoed timestamp in the last non-duplicate acknowledgment equals the stored timestamp. If true, duplicate ACKs indicate a lost segment (proceed to Step 1A in Section 3). Otherwise, duplicate ACKs likely result from unnecessary retransmissions (proceed to Step 1B in Section 3).

Examples of applying the timestamp heuristic are in validation tests "./test-all-newreno newreno_rto_loss_tsh" and "./test-all-newreno newreno_rto_dup_tsh". The timestamp heuristic works correctly, both when the receiver echoes timestamps as specified by [RFC1323], and by its revision attempts. However, if the receiver arbitrarily echoes timestamps, the heuristic can fail. The heuristic can also fail if a timeout was spurious and returning ACKs are not from retransmitted segments. This can be prevented by detection algorithms such as [RFC3522].

7. Implementation Issues for the Data Receiver

[RFC5681] specifies that "Out-of-order data segments SHOULD be acknowledged immediately, in order to accelerate loss recovery." Neal Cardwell has noted that some data receivers do not send an immediate acknowledgement when they send a partial acknowledgment, but instead wait first for their delayed acknowledgement timer to expire [C98]. As [C98] notes, this severely limits the potential benefit of NewReno by delaying the receipt of the partial acknowledgement at the data sender. Echoing RFC 5681, our recommendation is that the data receiver send an immediate acknowledgement for an out-of-order segment, even when that out-of-order segment fills a hole in the buffer.

8. Implementation Issues for the Data Sender

In Section 3, Step 5 above, it is noted that implementations should take measures to avoid a possible burst of data when leaving Fast Recovery, in case the amount of new data that the sender is eligible to send due to the new value of the congestion window is large. This can arise during NewReno when ACKs are lost or treated as pure window updates, thereby causing the sender to underestimate the number of new segments that can be sent during the recovery procedure. Specifically, bursts can occur when the FlightSize is much less than the new congestion window when exiting from Fast Recovery. One simple mechanism to avoid a burst of data when leaving Fast Recovery is to limit the number of data packets that can be sent in response to a single acknowledgment. (This is known as "maxburst_" in the ns simulator.) Other possible mechanisms for avoiding bursts include rate-based pacing, or setting the slow-start threshold to the resultant congestion window and then resetting the congestion window to FlightSize. A recommendation on the general mechanism to avoid excessively bursty sending patterns is outside the scope of this document.

An implementation may want to use a separate flag to record whether or not it is presently in the Fast Recovery procedure. The use of the value of the duplicate acknowledgment counter for this purpose is

not reliable because it can be reset upon window updates and out-of-order acknowledgments.

When updating the Cumulative Acknowledgement field outside of Fast Recovery, the "recover" state variable may also need to be updated in order to continue to permit possible entry into Fast Recovery (Section 3, step 1). This issue arises when an update of the Cumulative Acknowledgement field results in a sequence wraparound that affects the ordering between the Cumulative Acknowledgement field and the "recover" state variable. Entry into Fast Recovery is only possible when the Cumulative Acknowledgment field covers more than the "recover" state variable.

It is important for the sender to respond correctly to duplicate ACKs received when the sender is no longer in Fast Recovery (e.g., because of a Retransmit Timeout). The Limited Transmit procedure [RFC3042] describes possible responses to the first and second duplicate acknowledgements. When three or more duplicate acknowledgements are received, the Cumulative Acknowledgement field doesn't cover more than "recover", and a new Fast Recovery is not invoked, it is important that the sender not execute the Fast Recovery steps (3) and (4) in Section 3. Otherwise, the sender could end up in a chain of spurious timeouts. We mention this only because several NewReno implementations had this bug, including the implementation in the NS simulator. (This bug in the NS simulator was fixed in July 2003, with the variable "exitFastRetrans_".)

It has been observed that some TCP implementations enter a slow start or congestion avoidance window updating algorithm immediately after the cwnd is set by the equation found in (Section 3, step 5), even without a new external event generating the cwnd change. Note that after cwnd is set based on the procedure for exiting Fast Recovery (Section 3, step 5), cwnd SHOULD NOT be updated until a further event occurs (e.g., arrival of an ack, or timeout) after this adjustment.

9. Simulations

Simulations with NewReno are illustrated with the validation test "tcl/test/test-all-newreno" in the NS simulator. The command "../../ns test-suite-newreno.tcl reno" shows a simulation with Reno TCP, illustrating the data sender's lack of response to a partial acknowledgement. In contrast, the command "../../ns test-suite-newreno.tcl newreno_B" shows a simulation with the same scenario using the NewReno algorithms described in this paper.

10. Comparisons between Reno and NewReno TCP

As we stated in the introduction, we believe that the NewReno modification described in this document improves the performance of the Fast Retransmit and Fast Recovery algorithms of Reno TCP in a wide variety of scenarios. This has been discussed in some depth in [FF96], which illustrates Reno TCP's poor performance when multiple packets are dropped from a window of data and also illustrates NewReno TCP's good performance in that scenario.

We do, however, know of one scenario where Reno TCP gives better performance than NewReno TCP, that we describe here for the sake of completeness. Consider a scenario with no packet loss, but with sufficient reordering so that the TCP sender receives three duplicate acknowledgements. This will trigger the Fast Retransmit and Fast Recovery algorithms. With Reno TCP or with Sack TCP, this will result in the unnecessary retransmission of a single packet, combined with a halving of the congestion window (shown on pages 4 and 6 of [F03]). With NewReno TCP, however, this reordering will also result in the unnecessary retransmission of an entire window of data (shown on page 5 of [F03]).

While Reno TCP performs better than NewReno TCP in the presence of reordering, NewReno's superior performance in the presence of multiple packet drops generally outweighs its less optimal performance in the presence of reordering. (Sack TCP is the preferred solution, with good performance in both scenarios.) This document recommends the Fast Retransmit and Fast Recovery algorithms of NewReno TCP instead of those of Reno TCP for those TCP connections that do not support SACK. We would also note that NewReno's Fast Retransmit and Fast Recovery mechanisms are widely deployed in TCP implementations in the Internet today, as documented in [PF01]. For example, tests of TCP implementations in several thousand web servers in 2001 showed that for those TCP connections where the web browser was not SACK-capable, more web servers used the Fast Retransmit and Fast Recovery algorithms of NewReno than those of Reno or Tahoe TCP [PF01].

11. Changes Relative to RFC 2582

The purpose of this document is to advance the NewReno's Fast Retransmit and Fast Recovery algorithms in RFC 2582 to Standards Track.

The main change in this document relative to RFC 2582 is to specify the Careful variant of NewReno's Fast Retransmit and Fast Recovery algorithms. The base algorithm described in RFC 2582 did not attempt to avoid unnecessary multiple Fast Retransmits that can occur after a timeout (described in more detail in the section above). However, RFC 2582 also defined "Careful" and "Less Careful" variants that avoid these unnecessary Fast Retransmits, and recommended the Careful

variant. This document specifies the previously-named "Careful" variant as the basic version of NewReno. As described below, this algorithm uses a variable "recover", whose initial value is the send sequence number.

The algorithm specified in Section 3 checks whether the acknowledgement field of a partial acknowledgement covers *more* than "recover", as defined in Section 3. Another possible variant would be to simply require that the acknowledgement field covers *more than or equal to* "recover" before initiating another Fast Retransmit. We called this the Less Careful variant in RFC 2582.

There are two separate scenarios in which the TCP sender could receive three duplicate acknowledgements acknowledging "recover" but no more than "recover". One scenario would be that the data sender transmitted four packets with sequence numbers higher than "recover", that the first packet was dropped in the network, and the following three packets triggered three duplicate acknowledgements acknowledging "recover". The second scenario would be that the sender unnecessarily retransmitted three packets below "recover", and that these three packets triggered three duplicate acknowledgements acknowledging "recover". In the absence of SACK, the TCP sender is unable to distinguish between these two scenarios.

For the Careful variant of Fast Retransmit, the data sender would have to wait for a retransmit timeout in the first scenario, but would not have an unnecessary Fast Retransmit in the second scenario. For the Less Careful variant to Fast Retransmit, the data sender would Fast Retransmit as desired in the first scenario, and would unnecessarily Fast Retransmit in the second scenario. This document only specifies the Careful variant in Section 3. Unnecessary Fast Retransmits with the Less Careful variant in scenarios with reordering are illustrated in page 8 of [F03].

The document also specifies two heuristics that the TCP sender MAY use to decide to invoke Fast Retransmit even when the three duplicate acknowledgements do not cover more than "recover". These heuristics, an ACK-based heuristic and a timestamp heuristic, are described in Sections 6.1 and 6.2 respectively.

12. Changes Relative to RFC 3782

In [RFC3782], the cwnd after Full ACK reception will be set to (1) min (ssthresh, FlightSize + SMSS) or (2) ssthresh. However, there is a risk in the first logic which results in performance degradation. With the first logic, if FlightSize is zero, the result will be 1 SMSS. This means TCP can transmit only 1 segment at this moment, which can cause delay in ACK transmission at receiver due to

delayed ACK algorithm.

The FlightSize on Full ACK reception can be zero in some situations. A typical example is where sending window size during fast recovery is small. In this case, the retransmitted packet and new data packets can be transmitted within a short interval. If all these packets successfully arrive, the receiver may generate a Full ACK that acknowledges all outstanding data. Even if window size is not small, loss of ACK packets or receive buffer shortage during fast recovery can also increase the possibility to fall into this situation.

The proposed fix in this document ensures that sender TCP transmits at least two segments on Full ACK reception.

In addition, errata for RFC3782 (editorial clarification to Section 8) has been applied.

13. Conclusions

This document specifies the NewReno Fast Retransmit and Fast Recovery algorithms for TCP. This NewReno modification to TCP can even be important for TCP implementations that support the SACK option, because the SACK option can only be used for TCP connections when both TCP end-nodes support the SACK option. NewReno performs better than Reno (RFC 5681) in a number of scenarios discussed herein.

A number of options to the basic algorithm presented in Section 3 are also described. These include the handling of the retransmission timer (Section 4), the response to partial acknowledgments (Section 5), and the value of the congestion window when leaving Fast Recovery (section 3, step 5). Our belief is that the differences between these variants of NewReno are small compared to the differences between Reno and NewReno. That is, the important thing is to implement NewReno instead of Reno, for a TCP connection without SACK; it is less important exactly which of the variants of NewReno is implemented.

14. Security Considerations

RFC 5681 discusses general security considerations concerning TCP congestion control. This document describes a specific algorithm that conforms with the congestion control requirements of RFC 5681, and so those considerations apply to this algorithm, too. There are no known additional security concerns for this specific algorithm.

15. IANA Considerations

This document has no actions for IANA.

16. Acknowledgements

Many thanks to Anil Agarwal, Mark Allman, Armando Caro, Jeffrey Hsu, Vern Paxson, Kacheong Poon, Keyur Shah, and Bernie Volz for detailed feedback on this document or on its precursor, RFC 2582. Jeffrey Hsu provided clarifications on the handling of the recover variable that were applied to RFC 3782 as errata, and now are in Section 8 of this document. Yoshifumi Nishida contributed a modification to the fast recovery algorithm to account for the case in which flightsize is 0 when the TCP sender leaves fast recovery, and the TCP receiver uses delayed acknowledgments.

17. References

17.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

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

17.2. Informative References

[C98] Cardwell, N., "delayed ACKs for retransmitted packets: ouch!". November 1998, Email to the tcpimpl mailing list, Message-ID "Pine.LNX.4.02A.
9811021421340.26785-100000@sake.cs.washington.edu", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[F98] Floyd, S., Revisions to RFC 2001, "Presentation to the TCPIMPL Working Group", August 1998. URLs "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.ps" and "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.pdf".

[F03] Floyd, S., "Moving NewReno from Experimental to Proposed Standard? Presentation to the TSVWG Working Group", March 2003. URLs "http://www.icir.org/floyd/talks/newreno-Mar03.ps" and "http://www.icir.org/floyd/talks/newreno-Mar03.pdf".

[FF96] Fall, K. and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, July 1996. URL "ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z".

[F94] Floyd, S., "TCP and Successive Fast Retransmits", Technical report, October 1994. URL

"ftp://ftp.ee.lbl.gov/papers/fastretrans.ps".

[GF04] Gurtov, A. and S. Floyd, "Resolving Acknowledgment Ambiguity in non-SACK TCP", Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04), February 2004. URL "http://www.cs.helsinki.fi/u/gurtov/papers/heuristics.html".

[Gur03] Gurtov, A., "[Tsvwg] resolving the problem of unnecessary fast retransmits in go-back-N", email to the tsvwg mailing list, message ID <3F25B467.9020609@cs.helsinki.fi>, July 28, 2003. URL "http://www1.ietf.org/mail-archive/working-groups/tsvwg/current/msg04334.html".

[Hen98] Henderson, T., Re: NewReno and the 2001 Revision. September 1998. Email to the tcpimpl mailing list, Message ID "Pine.BSI.3.95.980923224136.26134A-100000@raptor.CS.Berkeley.EDU", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[Hoe95] Hoe, J., "Startup Dynamics of TCP's Congestion Control and Avoidance Schemes", Master's Thesis, MIT, 1995.

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

[LM97] Lin, D. and R. Morris, "Dynamics of Random Early Detection", SIGCOMM 97, September 1997. URL "http://www.acm.org/sigcomm/sigcomm97/program.html".

[NS] The Network Simulator (NS). URL "http://www.isi.edu/nsnam/ns/".

[PF01] Padhye, J. and S. Floyd, "Identifying the TCP Behavior of Web Servers", June 2001, SIGCOMM 2001.

[RFC1323] Jacobson, V., Braden, R. and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC2582] Floyd, S. and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[RFC2883] Floyd, S., J. Mahdavi, M. Mathis, and M. Podolsky, "The Selective Acknowledgment (SACK) Option for TCP, RFC 2883, July 2000.

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

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

[RFC3782] Floyd, S., T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004.

Appendix A. Document Revision History

To be removed upon publication

Revision	Comments
draft-00	RFC3782 errata applied, and changes applied from draft-nishida-newreno-modification-02

Authors' Addresses

Thomas R. Henderson
The Boeing Company
P.O. Box 3707
Seattle, WA 98124

EMail: thomas.r.henderson@boeing.com

Sally Floyd
International Computer Science Institute

Phone: +1 (510) 666-2989
EMail: floyd@acm.org
URL: http://www.icir.org/floyd/

Andrei Gurtov
HIIT
Helsinki Institute for Information Technology
Aalto University
P.O. Box 19800
Helsinki FIN-00076
FINLAND

EMail: gurtov@hiit.fi

Yoshifumi Nishida
WIDE Project
Endo 5322
Fujisawa, Kanagawa 252-8520
Japan

Email: nishida@wide.ad.jp