

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 3, 2010

Y. Nishida
WIDE Project
March 2, 2010

NewReno Modification for Smooth Recovery After Fast Retransmission
draft-nishida-newreno-modification-02

Abstract

This memo describes a feeble point in Fast Recovery algorithm in NewReno defined in [RFC3782](#) and proposes a simple modification to solve the problem.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on September 3, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

Table of Contents

1.	Introduction	3
2.	Conventions and Terminology	4
3.	Problem Description	5
4.	Possible Scenarios	6
4.1.	Case 1: Small Sending Window Size at Sender	6
4.2.	Case 2: Zero Window Advertisement from Receiver	6
4.3.	Case 3: Lost of ACK segments	7
5.	Discussion	9
6.	Proposed Fix	10
7.	Simulation Results	11
8.	Security Considerations	13
9.	IANA Considerations	14
10.	Normative References	15
	Author's Address	16

1. Introduction

There are some situations that NewReno cannot recover quickly after the success of fast retransmission. This issue is resulted from a feeble point in Fast Recovery algorithm in NewReno defined in [RFC3782](#) [[RFC3782](#)]. This document describes the point in Fast Recovery and presents possible scenarios. This memo also propose a simple modification to fix this problem.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC2119](#)].

Since this document describes a potential risk in NewReno, it uses the same terminology and definitions in [RFC3782](#) [[RFC3782](#)]. Which means this documents assumes that the reader is familiar with the terms SENDER MAXIMUM SEGMENT SIZE (SMSS), CONGESTION WINDOW (cwnd), and FLIGHT SIZE (FlightSize) defined in [[RFC2581](#)].

3. Problem Description

This section describes a potential risk in Fast Retransmit and Fast Recovery Algorithm in [RFC3782](#).

[Section 3 in RFC3782](#) describes the Fast Retransmit and Fast Recovery Algorithm in NewReno. The algorithm consists of 6 steps. The following lines are the description of the fifth steps which describes the behavior for the arrival of the first Full ACK after first retransmission.

- 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) $\min(\text{ssthresh}, \text{FlightSize} + \text{SMSS})$ 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.)

According to this description, the cwnd after the first FULL ACK reception will be one of the followings.

- (1) $\min(\text{ssthresh}, \text{FlightSize} + \text{SMSS})$
- (2) ssthresh

However, there is a risk in (1) which can cause performance degradation. In (1), if FlightSize is zero, the result of (1) will be 1 SMSS. (ssthresh should be bigger than 1) This means TCP can transmit only 1 segment in this case. This can cause the delay in ACK transmission at the receiver side if the receiver use delayed ACK algorithm. The FlightSize in (1) represents the amount of data outstanding in the fifth step: the moment when the new Full ACK arrives. The next section describes several scenarios where the FlightSize becomes zero.

[4.](#) Possible Scenarios

There are several possible situations that FlightSize becomes zero when the first new full ACK arrives after fast retransmission. This section describe several possible cases.

[4.1.](#) Case 1: Small Sending Window Size at Sender

This is the tcpdump example of the case. This log is recorded at A.

```
1 10:41:00.000001 A > B: . 1000:2000(1000) ack 1 win 32768
2 10:41:00.001001 A > B: . 2000:3000(1000) ack 1 win 32768
3 10:41:00.002001 A > B: . 3000:4000(1000) ack 1 win 32768
4 10:41:00.003001 A > B: . 4000:5000(1000) ack 1 win 32768
```

```

5  10:41:00.010001 B > A: . ack 1000 win 16384
6  10:41:00.011001 B > A: . ack 1000 win 16384
7  10:41:00.012001 B > A: . ack 1000 win 16384
8  10:41:00.013001 A > B: . 1000:2000(1000) ack 1 win 32768
9  10:41:00.014001 A > B: . 5000:6000(1000) ack 1 win 32768
10 10:41:00.024001 B > A: . ack 6000 win 16384
11 10:41:00.025001 A > B: . 6000:7000(1000) ack 1 win 32768

```

In this example, A sends data segments to B. At the beginning of the log, the cwnd of A is 4 SMSS, hence A sends 4 segments to B (line 1-4). Here, if the segment sent in line 1 (segment 1000:2000) is lost, B sends 3 duplicated ACKs for the lost segment (line 5-7) to ask retransmission. At line 8, A receives 3 duplicated ACKs then it transmits the lost segment. At line 9, A sets cwnd to ssthresh plus 3*SMSS (as defined in the second steps in NewReno algorithm) and cwnd becomes 5 SMSS as the result. This window inflation allows A to transmit one new segment.

Since the two segments in line 8 and 9 are usually transmitted almost at the same time, the receiver may send back only one ACK for these two segments (line 10) The ACK received in line 10 is the first Full ACK and there is no out-standing data in this moment. Hence, new cwnd is set to 1 SMSS and only one new segment is sent (line 11)

[4.2.](#) Case 2: Zero Window Advertisement from Receiver

This is the tcpdump example of the case. This log is recorded at A.

```

1  11:42:00.000001 A > B: . 1000:2000(1000) ack 1 win 32768
2  11:42:00.001001 A > B: . 2000:3000(1000) ack 1 win 32768
3  11:42:00.002001 A > B: . 3000:4000(1000) ack 1 win 32768
4  11:42:00.003001 A > B: . 4000:5000(1000) ack 1 win 32768
5  11:42:00.004001 A > B: . 5000:6000(1000) ack 1 win 32768
6  11:42:00.005001 A > B: . 6000:7000(1000) ack 1 win 32768
7  11:42:00.010001 B > A: . ack 1000 win 0
8  11:42:00.011001 B > A: . ack 1000 win 0

```

```

 9  11:42:00.012001 B > A: . ack 1000 win 0
10  11:42:00.012201 A > B: . 1000:2000(1000) ack 1 win 32768
11  11:42:00.013001 B > A: . ack 1000 win 0
12  11:42:00.014001 B > A: . ack 1000 win 0
13  11:42:00.022001 B > A: . ack 7000 win 16384
14  11:42:00.023001 A > B: . 7000:8000(1000) ack 1 win 32768

```

In this example, A sends data segments to B. At the beginning of the log, the cwnd of A is 6 SMSS, hence A sends 6 segments to B (line 1-6). Here, if the segment sent in line 1 (segment 1000:2000) is lost, B sends duplicated ACKs for the lost segment (line 7-9 and 11-12) to ask retransmission. However, these duplicated ACKs sent from B have zero advertised window because of buffer overflow. In this case, although the cwnd at A is inflated at the reception of the duplicated ACKs, it cannot transmit new segments. Hence, only the lost segment is retransmitted (line 10). When B receives retransmitted segment, the buffer becomes empty, then B sends a Full ACK with non-zero advertised window. The ACK received in line 13 is the first Full ACK and there is no out-standing data in this moment. Hence, new cwnd is set to 1 SMSS and only one new segment is sent (line 14)

[4.3.](#) Case 3: Lost of ACK segments

This is the tcpdump example of the case. This log is recorded at A.

```

1  12:43:00.000001 A > B: . 1000:2000(1000) ack 1 win 32768

```



```

2 12:43:00.001001 A > B: . 2000:3000(1000) ack 1 win 32768
3 12:43:00.002001 A > B: . 3000:4000(1000) ack 1 win 32768
4 12:43:00.003001 A > B: . 4000:5000(1000) ack 1 win 32768
5 12:43:00.004001 A > B: . 5000:6000(1000) ack 1 win 32768
6 12:43:00.005001 A > B: . 6000:7000(1000) ack 1 win 32768
7 12:43:00.010001 B > A: . ack 1000 win 16384
8 12:43:00.011001 B > A: . ack 1000 win 16384
9 12:43:00.012001 B > A: . ack 1000 win 16384
10 12:43:00.012201 A > B: . 1000:2000(1000) ack 1 win 32768
11 12:43:00.022001 B > A: . ack 7000 win 16384
12 12:43:00.023001 A > B: . 7000:8000(1000) ack 1 win 32768

```

In this example, A sends data segments to B. At the beginning of the log, the cwnd of A is 6 SMSS, hence A sends 6 segments to B (line 1-6). Here, if the segment sent in line 1 (segment 1000:2000) is lost, B generates 5 duplicated ACKS, however 2 ACK segments are lost in this case. Then, only 3 duplicated ACKs arrives at A (line 7-9). At line 10, A transmits the lost segment and sets cwnd to ssthresh plus 3*SMSS. As the result, the cwnd becomes 6 SMSS. However, this cwnd does not allow A to transmit new segments. At line 11, A receives the first Full ACK and there is no out-standing data in this moment. Hence, new cwnd is set to 1 SMSS and only one new segment is sent (line 12)

5. Discussion

Some TCP implementations such as Linux, NS-2 Network simulator do not have this issue. This is because these implementations always transmit more than 1 MSS right after fast recovery. In these implementations, when TCP exits Fast Recovery (when the first FULL ACK is received) it also calls "open cwnd" function at the same time and performs Slow Start or Congestion Avoidance algorithm. Hence, even though cwnd is set to 1 MSS after Fast Recovery as described in [Section 3](#), the cwnd will be increased by 1 MSS by Slow Start. (Since ssthresh should be bigger than 1 MSS at this moment, Slow Start is always used to increase cwnd)

However, this behavior can be controversial because it enters Slow-Start after Fast Recovery without receiving any packets. Although this point is unclear in [RFC3782](#), we believe that this is rather aggressive behavior and TCP should not open cwnd after Fast Recovery without receiving another ACKs. In fact, several implementation do not perform Slow Start right after Fast Recovery. With these implementations, severe performance degradations can be observed over lossy networks.

6. Proposed Fix

To solve the problem mentioned above, we propose a simple fix to the fifth step in NewReno.

The proposed solution is modifying the current cwnd adjustment:

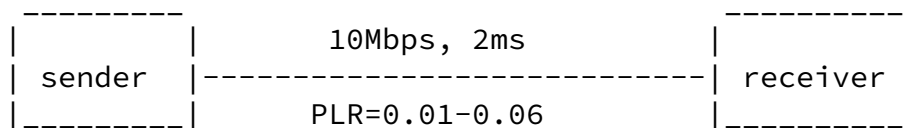
to

$$(1) \min (\text{ssthresh}, \text{FlightSize} + \text{SMSS})$$
$$(1) \min (\text{ssthresh}, \max(\text{FlightSize}, \text{SMSS}) + \text{SMSS})$$

This fix ensures that cwnd is always larger than 1 SMSS. Hence, sender TCP can always transmit at least two segments right after the first Full ACK reception. This can avoid the delay of ACK transmissions caused by delayed ACK algorithm. The new algorithm increases 1 SMSS only when FlightSize becomes zero and behaves completely the same as the previous algorithm does in other situations. The new algorithm might add slight burstness since it requires additional increase of cwnd. However, we believe this burstness can be almost negligible.

7. Simulation Results

In order to verify the effect of the issue described in this document, we implemented our algorithm in the TCP/Newreno agent in ns-2.34 and conducted several simulations. We used a simple network configuration as depicted in the below figure for our simulations. There is one 10Mbps link between the sender and the receiver and link delay is set to 2ms. The PLR on the link is set to 0.01 - 0.06 for the traffic towards the receiver. The sender transmits 100000 packets to the receiver with one TCP connection. (FTP application attached to TCP/Newreno agent is used) The receiver uses TCPSink/DelAck agent and delayed ack interval is set to 200ms.



With this configuration, we measured the performance of TCP by using the following three algorithms. alg1 is the algorithm adopted in the original NS-2 code or linux. alg2 is the algorithm that seems to be adopted in some other OSs. alg3 is the algorithm proposed in this document.

alg1 ... always do slow start after fast recovery without receiving ACKs

alg2 ... don't do slow start after fast recovery without receiving ACKs

alg3 ... don't do slow start after fast recovery without receiving ACKs. but, adjust cwnd to be always bigger than 1.

At first, we measured the number of events where flightsize becomes zero after fast recovery. As showed in the below table, when PLR=0.01, the ratio of the event is around 0.1% while it is around 2.0% when PLR=0.06. This means that the ratio of this event cannot be negligible under congested situations.

number of events where flightsize becomes zero after fast recovery

	PLR=0.01	PLR=0.02	PLR=0.03	PLR=0.04	PLR=0.05	PLR=0.06
alg1	108	333	687	1140	1537	1916
alg2	113	365	724	1182	1615	1939
alg3	107	371	717	1186	1587	1936

Next, we measured the throughput of each algorithm. As showed in the below table, alg2 exhibits serious performance degradation compared to the other two. alg1 maintains the best performance in all cases. This is because it has a bit aggressive natures. Although alg3 is a less aggressive algorithm than alg1, it attains mostly the same performance as alg1.

throughput (kbps)

	PLR=0.01	PLR=0.02	PLR=0.03	PLR=0.04	PLR=0.05	PLR=0.06
alg1	1028.49	697.87	491.29	356.36	257.71	198.65
alg2	825.57	451.96	284.25	190.13	137.99	107.05
alg3	1006.64	671.86	470.39	344.28	248.71	193.30

From these results, we recommend not to adopt alg2 and to use alg1 or alg3. We also believe that alg3 is the best algorithm since it can attain good performance while it keeps conservative nature as we discuss in this draft.

[8.](#) Security Considerations

This document only propose simple modification in [RFC3782](#). There are no known additional security concerns for this algorithm.

[9.](#) IANA Considerations

This document does not create any new registries or modify the rules for any existing registries managed by IANA.

[10](#). Normative References

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

- [RFC2581] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", [RFC 2581](#), April 1999.
- [RFC3782] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", [RFC 3782](#), April 2004.

Author's Address

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

Email: nishida@wide.ad.jp

