Network Working Group
INTERNET-DRAFT
Expires: January 2003

Reiner Ludwig
Ericsson Research
July, 2002

Responding to Fast Timeouts in TCP

<draft-ludwig-tsvwg-tcp-fast-timeouts-00.txt>

Status of this memo

Abstract

A "fast timeout" occurs if the retransmission timer expires, and afterwards the TCP sender receives the duplicate ACK that would have triggered a fast retransmit of the oldest outstanding segment. In this case, staying in slow start is an unnecessarily drastic response to the congestion indication. Instead, we believe it is safe to back out of the slow start phase but instead go into the fast recovery phase. One benefit of this approach is that the potentially following duplicate ACKs can be exploited for advanced loss recovery algorithms.

Terminology

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

We use the term 'acceptable ACK' as defined in [RFC793]. That is an ACK that acknowledges previously unacknowledged data. We use the term 'duplicate ACK', and the variable 'dupacks' as defined in [WS95]. The variable 'dupacks' is a counter of duplicate ACKs that have already been received by the TCP sender before the fast retransmit is sent. We use the variable 'DupThresh' to refer to the so-called duplicate acknowledgement threshold, i.e., the number of duplicate ACKs that need to arrive at the TCP sender to trigger a fast retransmit. Currently, DupThresh is specified as a fixed value of three [RFC2581].

Furthermore, we use the TCP sender state variables 'SND.UNA' and 'SND.NXT' as defined in [RFC793]. SND.UNA holds the segment sequence number of the oldest outstanding segment. SND.NXT holds the segment sequence number of the next segment the TCP sender will (re-)transmit. In addition, we define as 'SND.MAX' the segment sequence number of the next original transmit to be sent. The definition of SND.MAX is equivalent to the definition of snd_max in [WS95].

We use the TCP sender state variables 'cwnd' (congestion window), and 'ssthresh' (slow start threshold), and the terms 'SMSS', and 'FlightSize' as defined in [RFC2581]. FlightSize is the amount of outstanding data in the network, or alternatively, the difference between SND.MAX and SND.UNA at a given point in time. We use the TCP sender state variables 'SRTT' and 'RTTVAR', and the term 'RTO' as defined in [RFC2988]. In addition, we assume that the TCP sender maintains in the variable 'RTT-SAMPLE' the value of the latest round-trip time (RTT) measurement.

1. Introduction

We call a timeout a "fast timeout" if the retransmission timer expires, and afterwards the TCP sender receives the duplicate ACK that would have triggered a fast retransmit of the oldest outstanding segment [RFC2581]. We name this a fast timeout since in competition with the fast retransmit algorithm the timeout was faster.

As with the common case of a spurious timeout (see definition in [LM02]), a fast timeout would not have occurred had the sender "waited longer". However, a fast timeout is not spurious since apparently a segment was in fact lost, i.e., loss recovery was

entered rightfully.

While we have no data that indicates how frequent TCP fast timeouts occur in the general Internet, they do occur when wireless. They have been observed on paths that span across wide-area wireless links [Gu01]. In those experiments TCP fast timeouts occurred due to drastic bit rate reductions of the dedicated (non-shared with other hosts) wireless link that also happened to be the path's bottleneck link. Such rate changes may, e.g., occur in current wide-area wireless links due to a host roaming into a more congested radio cell, or due to preemption of radio resources in favor of higher priority traffic.

The fast timeout algorithm is a spin-off that had originally been proposed as part of the Eifel detection and response algorithms [LM02], [LG02]. There are two main reason why we have separated it into an independent document.

First, the Eifel detection and response algorithms only kick in if a loss recovery has been initiated unnecessarily, i.e., when in fact no congestion indication has been given to the TCP sender. On the contrary, the fast timeout algorithm kicks in even though the TCP has received a congestion indication.

Second, the Eifel response algorithm relies on the Eifel detection algorithm that in turn relies on the use of the TCP Timestamps option [RFC1323]. On the contrary, the fast timeout algorithm defines its own detection scheme that does not rely on the use of the TCP Timestamps option, nor on the use of any other TCP option.

2. Responding to Fast Timeouts

2.1 The Fast Timeout Algorithm

A TCP sender MAY use the fast timeout algorithm as defined in this subsection.

If the fast timeout algorithm is used, the following steps MUST be taken by the TCP sender, but only upon initiation of loss recovery, and only if that was triggered by a timeout. Note: The algorithm MUST NOT be reinitiated after loss recovery has already started. In particular, it may not be reinitiated upon subsequent timeouts for the same segment.

(1) Before the variables cwnd and ssthresh have been updated when loss recovery is initiated, set a "cwnd_prev" variable to the current value of FlightSize, and set a "ssthresh_prev" variable to the value of ssthresh.

Note: The value of the variable dupacks might be greater

than zero at this point. For example, when one or two duplicate ACKs have already been received when the

timeout occurs.

(2) Wait for the arrival of either an acceptable or a duplicate ACK. If such an ACK arrives, then update the variable dupacks and proceed to step (3).

(3) If an acceptable ACK has arrived, then proceed to step (DONE),

else if the value of the variable dupacks is smaller than the value of the variable DupThresh, then return to step (2)

else (dupacks equals DupThresh) proceed to step (4).

(4) Resume transmission off the top:

Suppress the fast retransmit, and set
SND.NXT <- SND.MAX

(5) Make the RTT estimator more conservative:

Set
SRTT <- 2*SRTT,
recalcualte the RTO, and restart the retransmission timer.

(6) Leave slow start and move to congestion avoidance:

Set
ssthresh <- max (cwnd_prev/2, 2*SMSS)
cwnd <- ssthresh + SMSS * DupThresh

(DONE) No further processing.

2.2 Motivating the Response Steps

2.2.1 Suppressing the Fast Retransmit (step 4)

Since the timeout already triggered a retransmit of the oldest outstanding segment, another (fast) retransmit of that segment should be suppressed. Instead, transmission should be resumed with new data as done in the fast recovery algorithm [RFC2581].

2.2.2 Making the RTO more Conservative (step 5)

Given that a fast timeout occurred, the TCP sender's RTT estimators

are likely to be off. However, the TCP sender cannot derive a new and valid RTT measurement from the duplicate ACK [RFC1323]. It is

therefore suggested to double SRTT to make the RTO more conservative for future segment transmissions.

To have the new RTO become effective, the retransmission timer needs to be restarted. This is a more conservative management of the retransmission timer than recommended in [RFC2988].

2.2.3 Moving from Slow Start to Congestion Avoidance(step 6)

Given that the TCP sender has received the duplicate ACK that would have triggered a fast retransmit, staying in slow start [RFC2581] is an unnecessarily drastic response to the congestion indication. Instead, we believe it is safe to back out of the slow start phase but instead go into the fast recovery phase.

A benefit of this approach is that the potentially following duplicate ACKs can be exploited for advanced SACK-based loss recovery algorithms [RFC2018], [BA02].

3. Security Considerations

As with standard TCP there is a risk that misbehaving TCP receivers spoof duplicate ACKs to "tune" a TCP sender's send rate [SCWA99]. A TCP sender that implements the fast timeout algorithm is slightly more vulnerable to such attacks than a standard TCP sender. This is because the TCP sender gets "upgraded" from a smaller congestion window during slow start to a larger congestion window during congestion avoidance.

Still, a TCP sender that implements the fast timeout algorithm remains responsive to congestion indications in such cases. Hence, the mentioned risk does not pose any threat to the stability of the Internet, and should only result in minor unfairness against less capable TCP senders. We believe that the unfairness is not any larger than the unfairness caused, e.g., by newer TCPs that start with a larger initial congestion window.

References

[RFC2581] M. Allman, V. Paxson, W. Stevens, TCP Congestion Control, RFC 2581, April 1999.

[BA02] E. Blanton, M. Allman, A Conservative SACK-based Loss Recovery Algorithm for TCP, work in progress, July 2002.

[RFC2119] S. Bradner, Key words for use in RFCs to Indicate

Requirement Levels, RFC 2119, March 1997.

[Gu01] A. Gurtov, Effect of Delays on TCP Performance, In Proceedings of IFIP Personal Wireless Conference, August 2001.

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

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

[LM02] R. Ludwig, M. Meyer, The Eifel Detection Algorithm for TCP, work in progress, July 2002.

[LG02] R. Ludwig, A. Gurtov, The Eifel Response Algorithm for TCP, work in progress, July 2002.

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

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

[SCWA99] S. Savage, N. Cardwell, D. Wetherall, T. Anderson, TCP Congestion Control with a Misbehaving Receiver, ACM Computer Communications Review, October 1999.

[WS95] G. R. Wright, W. R. Stevens, TCP/IP Illustrated, Volume 2 (The Implementation), Addison Wesley, January 1995.

Author's Address

Reiner Ludwig
Ericsson Research
Ericsson Allee 1
52134 Herzogenrath, Germany
Email: Reiner.Ludwig@ericsson.com

This Internet-Draft expires in January 2003.