Network Working Group
INTERNET-DRAFT
Expires: April 2003

Reiner Ludwig
Ericsson Research
Andrei Gurtov
Sonera Corporation
October, 2002

The Eifel Response Algorithm for TCP
<draft-ietf-tsvwg-tcp-eifel-response-01.txt>

Status of this memo

Abstract

The Eifel response algorithm uses the Eifel detection algorithm to detect a posteriori whether the TCP sender has entered loss recovery unnecessarily. In response to a spurious timeout it avoids the often unnecessary go-back-N retransmits that would otherwise be sent, and reinitializes the RTT estimators to avoid further spurious timeouts. Likewise, it adapts the duplicate acknowledgement threshold in response to a spurious fast retransmit. In both cases, the Eifel response algorithm restores the congestion control state in such a way that packet bursts are avoided.

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 refer to the first-time transmission of an octet as the 'original transmit'. A subsequent transmission of the same octet is referred to as a 'retransmit'. In most cases this terminology can likewise be applied to data segments as opposed to octets. However, when repacketization occurs, a segment can contain both first-time transmissions and retransmissions of octets. In that case this terminology is only consistent when applied to octets. For the Eifel detection and response algorithms this makes no difference as they also operate correctly when repacketization occurs.

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

The Eifel response algorithm relies on the Eifel detection algorithm defined in [LM02]. That document discusses the relevant background

and motivation that also applies to this document. Hence, the reader is expected to be familiar with [LM02]. Note that alternative response algorithms are conceivable that could also rely on the Eifel detection algorithm.

The Eifel response algorithm uses the Eifel detection algorithm to detect a posteriori whether the TCP sender has entered loss recovery unnecessarily. In response to a spurious timeout it avoids the often unnecessary go-back-N retransmits that would otherwise be sent, and reinitializes the RTT estimators to avoid further spurious timeouts. Likewise, it adapts the duplicate acknowledgement threshold in response to a spurious fast retransmit. In both cases, the Eifel response algorithm restores the congestion control state in such a way that packet bursts are avoided.

2. The Eifel Response Algorithm

The complete algorithm is specified in section 2.1. In sections 2.2 to 2.4, we motivate the different steps of the algorithm.

2.1. The Algorithm

Given that a TCP sender has enabled the Eifel detection algorithm [LM02] for a connection, a TCP sender MAY use the Eifel response algorithm as defined in this subsection. Note that this implies that the TCP Timestamps option [RFC1323] is used for that connection. Since the Eifel response algorithm is dependent on the Eifel detection algorithm, we describe it as an extension of the latter.

If the combined Eifel detection and response algorithm is used, the

following steps MUST be taken by the TCP sender, but only upon initiation of loss recovery, i.e., when either the timeout-based retransmit or the fast retransmit is sent. 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, and not upon retransmitting segments other than the oldest outstanding segment.

Note that steps (1)-(6) are an one-to-one copy of the Eifel detection algorithm specified in [LM02], step (0) has been added, and step (RESP) from [LM02] has been replaced by steps (RESP)-(ReCC) given below.

(0) Before the variables cwnd and ssthresh get updated when loss recovery is initiated, set a "pipe_prev" variable as follows:
pipe_prev <- max (FlightSize, ssthresh)

(1) Set a "SpuriousRecovery" variable to FALSE (equal 0).

(2) Set a "RetransmitTS" variable to the value of the Timestamp Value field of the Timestamps option included in the retransmit sent when loss recovery is initiated. A TCP sender must ensure that RetransmitTS does not get overwritten as loss recovery progresses, e.g., in case of a second timeout and subsequent second retransmit of the same octet.

(3) Wait for the arrival of an acceptable ACK. When an acceptable ACK has arrived proceed to step (4).

(4) If the value of the Timestamp Echo Reply field of the acceptable ACK's Timestamps option is smaller than the value of the variable RetransmitTS, then proceed to step (5),

else proceed to step (DONE).

(5) If the acceptable ACK does not carry a DSACK option [RFC2883], then proceed to step (6),

else proceed to step (DONE).

(6) If the loss recovery has been initiated with a timeout-based retransmit, then set
SpuriousRecovery <- SPUR_TO (equal 1),

else set
SpuriousRecovery <- dupacks+1

(RESP) If SpuriousRecovery equals SPUR_TO, then proceed to step (STO.1),

else (spurious fast retransmit) proceed to step (SFR).

(STO.1) Resume transmission off the top:

Set
SND.NXT <- SND.MAX

(STO.2) Reinitialize the RTT estimators:

Set
SRTT <- RTT-SAMPLE
RTTVAR <- RTT-SAMPLE/2,
recalculate the RTO, and restart the retransmission timer.

Proceed to step (ReCC).

(SFR) Adapt the duplicate acknowledgement threshold:

Set
DupThresh <- max (DupThresh, SpuriousRecovery)

Proceed to step (ReCC).

(ReCC) Revert the congestion control state:

If the acceptable ACK has the ECN-Echo flag [RFC3168] set OR the TCP sender has already taken more than three timeouts for the oldest outstanding segment, then proceed to step (DONE),

```
            else set
                cwnd <- FlightSize + SMSS
                ssthresh <- pipe_prev

            Note: At this point in the algorithm, the value of
            FlightSize might be different from the value of FlightSize
            in step (0).

            Proceed to step (DONE).

    (DONE)  No further processing.
```

2.2 Responding to Spurious Timeouts

2.2.1 Suppressing the Unnecessary go-back-N Retransmits (step STO.1)

Without the use of the TCP timestamps option, the TCP sender suffers from the retransmission ambiguity problem [Zh86], [KP87]. This means that when the first acceptable ACK arrives after a spurious timeout, the TCP sender must believe that that ACK was sent in response to the retransmit when in fact it was sent in response to the original transmit. Furthermore, the TCP sender must also believe that all other segments outstanding at that point were lost.

> Note: Except for certain cases where original ACKs were lost, that first acceptable ACK cannot carry any DSACK option [RFC2883].

Consequently, once the TCP sender's state has been updated after the first acceptable ACK has arrived, SND.NXT equals SND.UNA. This is what causes the often unnecessary go-back-N retransmits. Now every arriving acceptable ACK that was sent in response to an original transmit will advance SND.NXT. But as long as SND.NXT is smaller than the value that SND.MAX had when the timeout occurred, those ACKs will clock out retransmits; whether those segments were lost or not.

In fact, during this phase the TCP sender breaks 'packet conservation' [Jac88]. This is because the go-back-N retransmits are sent during slow start. I.e., for each original packet leaving the

network, two retransmits are sent into the network as long as SND.NXT

does not equal SND.MAX (see [LK00] for more detail).

The use of the TCP timestamps option reliably eliminates the retransmission ambiguity problem. Thus, once the Eifel detection algorithm detected that a timeout was spurious, it is therefore safe to let the TCP sender resume the transmission with new data. Thus, the Eifel response algorithm changes the TCP sender's state by setting SND.NXT to SND.MAX in that case.

2.2.2 Re-Initializing the RTT Estimators (step STO.2)

Since the timeout was spurious, the TCP sender's RTT estimators are likely to be off. On the other hand, since timestamps are used, a new and valid RTT measurement (RTT-SAMPLE) can be derived from the acceptable ACK. It is therefore suggested to reinitialize the RTT estimators from RTT-SAMPLE.

To have the new RTO become effective, the retransmission timer needs to be restarted. This is consistent with [RFC2988] which recommends restarting the retransmission timer with the arrival of an acceptable ACK.

2.3 Responding to Spurious Fast Retransmits (step SFR)

The assumption behind the fast retransmit algorithm [RFC2581] is that a segment was lost if as many duplicate ACKs have arrived at the TCP sender as indicated by DupThresh. Currently, DupThresh is specified as a fixed value of three [RFC2581]. That value is assumed to be sufficiently conservative so that packet reordering and/or packet duplication does not falsely trigger the fast retransmit algorithm. Clearly, this assumption does not hold for a particular TCP connection once the TCP sender detects that the last fast retransmit was spurious. It is therefore suggested to dynamically adapt DupThresh to the reordering characteristics observed over the course of a particular connection.

At the beginning of a connection DupThresh is initialized with three. Then for each spurious fast retransmit that is detected, DupThresh is set to the maximum of the previous DupThresh, and the lowest value that would have avoided that last spurious fast retransmit. Note that the Eifel detection algorithm records the latter value in SpuriousRecovery. This strategy ensures that the TCP sender is able to cope with the longest reordering length seen on a particular connection so far.

However, the strategy bears the risk that the retransmission timer expires before the TCP sender receives the duplicate ACK that would trigger a fast retransmit of the oldest outstanding segment. To

alleviate that potential problem the TCP sender may implement the Fast Timeout algorithm proposed in [Lu02].

Also, we believe that this strategy should be implemented together with an advanced version of the Limited Transmit algorithm [RFC3042]. That is for each duplicate ACK that arrives until DupThresh is reached, the TCP sender should sent a new data segment if allowed by the TCP receiver's advertised window, and if new data is available. Although, the current Limited Transmit algorithm only allows this for the first two duplicate ACKs, we believe that such an advanced limited transmit strategy is safe. It is already implemented in widely deployed TCPs [SK02].

Other alternatives for responding to spurious fast retransmits are discussed in [BA02a].

2.4 Reverting Congestion Control State (step ReCC)

When a TCP sender enters loss recovery, it also assumes that is has received a congestion indication. In response to that it reduces cwnd, and ssthresh. However, once the TCP sender detects that the loss recovery has been falsely triggered, this reduction was unnecessary. In fact, no congestion signal has been received. We therefore believe that it is safe to revert to the previous congestion control state.

To avoid packet bursts, we suggest to restore cwnd to the amount of data currently outstanding in the network plus one SMSS. That will allow no more than a single packet to be clocked out by the first acceptable ACK. In addition, we suggest to restore ssthresh to pipe_prev, i.e., the maximum of the previous value of ssthresh and the value that FlightSize had when loss recovery was unnecessarily entered. As a result, the TCP sender either immediately resumes probing the network for more bandwidth in congestion avoidance, or it first slow starts until it has reached its previous share of the available bandwidth.

Clearly, when the acceptable ACK signals congestion through the ECN-Echo flag [RFC3168], the TCP sender MUST refrain from reverting congestion control state. The same is true if the TCP sender has already taken more than three timeouts for the oldest outstanding

segment. Allowing three timeouts while still reverting congestion control state goes beyond [RFC2581]. That standard recommends setting cwnd to no more than the restart window (one SMSS) if the TCP sender has not sent data in an interval exceeding the current RTO. That is done to restart the ACK clock which is believed to be lost. The case in step (ReCC) of the Eifel response algorithm is different. Since, an acceptable ACK corresponding to an original transmit has finally returned, the TCP has reason to believe that the ACK clock was merely interrupted but has now resumed "ticking" again.

3. Interoperability with Advanced Loss Recovery Schemes

We believe that there are no problems concerning interoperability with advanced loss recovery schemes such as NewReno [RFC2582], or SACK-based schemes [2018], [BA02b]. This is because in case loss recovery has been initiated unnecessarily, the Eifel response algorithm makes the TCP sender back out of loss recovery before those schemes would have a chance to kick in.

In fact, we recommend that the Eifel response algorithm is implemented together with one of those advanced loss recovery schemes; ideally a SACK-based alternative. In an environment where spurious timeouts and back-to-back packet losses often coincide, we have found that TCP's performance can even suffer if the Eifel response algorithm is operated without an advanced loss recovery scheme [GL02].

In that study, we among other variants compared TCP-Reno with and without the Eifel response algorithm (TCP-Reno/Eifel vs. TCP-Reno), and without an advanced loss recovery scheme for both variants. The reason that TCP-Reno performed better in the mentioned scenario, is its aggressiveness after a spurious timeout. Even though it breaks 'packet conservation' (see Section 2.2.1) when blindly retransmitting all outstanding segments, it usually recovers the back-to-back packet losses within a single round-trip time. On the contrary, the more conservative TCP-Reno/Eifel was forced into another (backed-off) timeout in that case. In the study, we found that the best end-to-end performance was achieved when the TCP sender implemented both the Eifel response algorithm and SACK-based loss recovery. In case NewReno is chosen as the advanced loss recovery scheme, we found that

it performs better if the 'bugfix' feature is disabled. That feature often leads the TCP sender to the wrong decision.

4. Security Considerations

There is a risk that TCP receivers make genuine retransmits appear to the TCP sender as spurious retransmits by forging echoed timestamps. This could effectively disable congestion control at the TCP sender. A reliable method to protect against that risk is to implement the safe variant of the Eifel detection algorithm specified in [LM02].

Acknowledgments

Many thanks to Keith Sklower, Randy Katz, Michael Meyer, Stephan Baucke, Sally Floyd, Vern Paxson, Mark Allman, and Ethan Blanton for very useful discussions that contributed to this work.

Normative References

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

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

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

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

[RFC2883] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, A. Romanow, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, July 2000.

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

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

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

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

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

[RFC3168] K. Ramakrishnan, S. Floyd, D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, September 2001

Informative References

[BA02a] E. Blanton, M. Allman, On Making TCP More Robust to Packet Reordering, ACM Computer Communication Review, Vol. 32, No. 1, January 2002.

[BA02b] E. Blanton, M. Allman, A Conservative SACK-based Loss Recovery Algorithm for TCP, work in progress, October 2002.

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

[GL02] A. Gurtov, R. Ludwig, Evaluating the Eifel Algorithm for TCP in a GPRS Network, In Proceedings of the European Wireless Conference, February 2002.

[KP87] P. Karn, C. Partridge, Improving Round-Trip Time Estimates in Reliable Transport Protocols, In Proceedings of ACM SIGCOMM 87.

[LK00] R. Ludwig, R. H. Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, ACM Computer Communication Review, Vol. 30, No. 1, January 2000.

[Lu02] R. Ludwig, Responding to Fast Timeouts in TCP, work in progress, July 2002.

[SK02] P. Sarolahti, A. Kuznetsov, Congestion Control in Linux TCP, In Proceedings of USENIX, June 2002.

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

[Zh86] L. Zhang, Why TCP Timers Don't Work Well, In Proceedings of ACM SIGCOMM 88.

Author's Address

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

Andrei Gurtov
Cellular Systems Development
P.O. Box 970, FIN-00051 Sonera
Helsinki, Finland
Phone: +358(0)20401
Fax: +358(0)204064365
Email: andrei.gurtov@sonera.com
Homepage: http://www.cs.helsinki.fi/u/gurtov

This Internet-Draft expires in April 2003.