Network Working Group
INTERNET-DRAFT
Expires: April 2004

Reiner Ludwig
Ericsson Research
Andrei Gurtov
TeliaSonera

October, 2003

The Eifel Response Algorithm for TCP

<draft-ietf-tsvwg-tcp-eifel-response-04.txt>

Status of this memo

Abstract

Based on an appropriate detection algorithm, the Eifel response algorithm provides a way for a TCP sender to respond to a detected spurious timeout. It adapts the retransmission timer to avoid further spurious timeouts, and can avoid - depending on the detection algorithm - the often unnecessary go-back-N retransmits that would otherwise be sent. In addition, 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 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 'FlightSize', and 'Initial Window (IW)' as defined in [RFC2581]. FlightSize is the amount of outstanding data at a given point in time. The IW is the size of the sender's congestion window after the three-way handshake is completed. 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 a detection algorithm such as the Eifel detection algorithm defined in [RFC3522]. That document discusses the relevant background and motivation that also applies to this document. Hence, the reader is expected to be familiar with

[RFC3522]. Note that alternative response algorithms have been proposed [BA02] that could also rely on the Eifel detection algorithm, and vice versa alternative detection algorithms have been proposed [BA03], [SK03] that could work together with the Eifel response algorithm.

Based on an appropriate detection algorithm, the Eifel response algorithm provides a way for a TCP sender to respond to a detected spurious timeout. It adapts the retransmission timer to avoid further spurious timeouts, and can avoid - depending on the detection algorithm - the often unnecessary go-back-N retransmits that would otherwise be sent. In addition, the Eifel response algorithm restores the congestion control state in such a way that packet bursts are avoided.

Note: A previous version of the Eifel Response algorithm also included a response to a detected spurious fast retransmit. However, since a consensus was not reached about how to adapt the duplicate acknowledgement threshold in that case, that part of the algorithm was removed for the time being.

2. Interworking with Detection Algorithms

If the Eifel response algorithm is implemented at the TCP sender, it MUST be implemented together with a detection algorithm that is specified in an RFC.

Designers of detection algorithms who want their algorithms to work together with the Eifel response algorithm should reuse the variable SpuriousRecovery with the semantics and defined values as specified in [RFC3522]. In addition, we define LATE_SPUR_TO (equal -1) as another possible value of the variable SpuriousRecovery. Detection algorithms should set the value of SpuriousRecovery to LATE_SPUR_TO if the detection of a spurious retransmit is based upon receiving the ACK for the retransmit (as opposed to the ACK for the original transmit). For example, this applies to detection algorithms that are based on the DSACK option [BA03].

3. The Eifel Response Algorithm

The complete algorithm is specified in section 3.1. In sections 3.2 to 3.5, we motivate the different steps of the algorithm.

3.1. The Algorithm

Given that a TCP sender has enabled a detection algorithm that complies with the requirements set in Section 2, a TCP sender MAY use the Eifel response algorithm as defined in this subsection.

If the Eifel response algorithm is used, the following steps MUST be taken by the TCP sender, but only upon initiation of loss recovery, i.e., when the timeout-based 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.

(INIT) 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)

   (DET)   This is a placeholder for a detection algorithm that must
           be executed at this point. In case [RFC3522] is used as
           the detection algorithm, steps (1) - (6) of that algorithm
           go here.

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

           else if SpuriousRecovery equals LATE_SPUR_TO, then
               proceed to step (STO.2),

           else
               proceed to step (DONE).

   (STO.1) Resume transmission off the top:

           Set
               SND.NXT <- SND.MAX

   (STO.2) Adapt the Conservativeness of the Retransmission Timer:

           If the retransmission timer is implemented according to
           [RFC2988], then
               if the TCP Timestamps option [RFC1323] is enabled for
               this connection, then set
                   SRTT <- RTT-SAMPLE
                   RTTVAR <- RTT-SAMPLE/2

               else set
                   RTTVAR <- max (2 * RTTVAR, SRTT)
                   SRTT <- 2 * SRTT

               Set
                   RTO <- SRTT + max (G, 4*RTTVAR)
               Restart the retransmission timer
```

else
    appropriately adapt the conservativeness of the
    retransmission timer that is implemented.

Proceed to step (ReCC).

(ReCC) Reversing the congestion control state:

If the acceptable ACK has the ECN-Echo flag [RFC3168] set, then
    proceed to step (DONE),

else set

```
cwnd <- FlightSize + min (bytes_acked, IW)
ssthresh <- pipe_prev
```

Proceed to step (DONE).

(CWV) Interworking with Congestion Window Validation (the variables 'T_last' and 'tcpnow' are defined in [RFC2861]):

If congestion window validation is implemented according to [RFC2861], then set

```
T_last <- tcpnow
```

(DONE) No further processing.

3.2 Storing the Current Congestion Control State (step INIT)

The TCP sender stores in pipe_prev what is considered a "safe" slow-start threshold (ssthresh) before loss recovery is initiated, i.e., before the loss indication is taken into account. This is either the current FlightSize if the TCP sender is in congestion avoidance or the current ssthresh if the TCP sender is in slow-start. If the TCP sender later detects that it has entered loss recovery unnecessarily, then pipe_prev is used in step (ReCC) to reverse the congestion control state. Thus, until the loss recovery phase is terminated, pipe_prev maintains a memory of the congestion control state of the time right before the loss recovery phase was initiated. A similar approach is proposed in [RFC2861], where this state is stored in ssthresh directly after a TCP sender has become application-limited.

There had been debates about whether the value of pipe_prev should be decayed over time, e.g., upon subsequent timeouts for the same outstanding segment. We do not require the decaying of pipe_prev for the Eifel response algorithm, and do not believe that such a conservative approach would be in place. Instead, we follow the idea of revalidating the congestion window through slow-start as suggested in [RFC2861]. That is, in step (ReCC), the cwnd is reset to a value that avoids large packet bursts, while ssthresh is reset to the value of pipe_prev. Note that [RFC2581] and [RFC2861] also do not require a decaying of ssthresh after it has been reset in response to a loss indication, or after a TCP sender has become application-limited.

3.3 Responding to Spurious Timeouts

3.3.1 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]. Hence, when the first acceptable ACK arrives after a spurious timeout, the TCP sender must assume that this 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 further assume that all other segments outstanding at that point were lost.

Note: Except for certain cases where original ACKs were lost, the 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. From that point on 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 transmit 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. Once the Eifel detection algorithm has 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.

3.3.2 Adapting the Retransmission Timer (step STO.2)

There is currently only one retransmission timer standardized for TCP [RFC2988]. We therefore only address that timer explicitly. Future standards that might define alternatives to [RFC2988] should propose similar measures to adapt the conservativeness of the retransmission timer.

Since the timeout was spurious, the TCP sender's RTT estimators are likely to be off. If timestamps are enabled for this connection, 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 according to rule (2.2) of RFC2988. Note that this RTT-SAMPLE will be relatively large since it will include the delay spike that caused the spurious timeout in the first place. If timestamps are not enabled for this connection, the TCP sender should instead double SRTT and also make RTTVAR more conservative.

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.

3.4 Reversing the 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 indication has been received. We therefore believe that it is safe to revert to the previous congestion control state following the approach of revalidating the congestion window as outlined below. This is unless the acceptable ACK signals congestion through the ECN-Echo flag [RFC3168]. In that case, the TCP sender MUST refrain from reversing congestion control state.

If the ECN-Echo flag is not set, cwnd is reset to the sum of the current FlightSize and the minimum of IW and the number of bytes that have been acknowledged by the acceptable ACK. Note that the value of cwnd must not be changed any further for that ACK, and that the value of FlightSize at this point in time may be different from the value of FlightSize in step (INIT). The value of IW puts a limit on the size of the packet burst that the TCP sender may send into the network after the Eifel response algorithm has terminated. The value of IW is considered an acceptable burst size. It is the amount of data that a TCP sender may send into a yet "unprobed" network at the beginning of a connection.

The TCP sender is then forced into slow-start by resetting ssthresh to the value of pipe_prev. As a result, the TCP sender either immediately resumes probing the network for more bandwidth in congestion avoidance, or it first slow-starts to what is considered a "safe" operating point for the congestion window. In some cases, this can mean that the first few acceptable ACKs that arrive will not clock out any data segments.

3.5 Interworking with the Congestion Window Validation Algorithm

An implementation of the Congestion Window Validation (CWV) algorithm [RFC2861] could potentially misinterpret a delay spike that caused a spurious timeout as a phase where the TCP sender had been application-limited. To prevent the triggering of CWV algorithm in this case, the variable 'T_last' defined in [RFC2861] is reset.

4. Non-Conservative Advanced Loss Recovery after Spurious Timeouts

A TCP sender MAY implement an optimistic form of advanced loss recovery after a spurious timeout has been detected as motivated in this section. Such a scheme MUST be terminated after the highest sequence number outstanding when the spurious timeout was detected has been acknowledged.

We have studied environments where spurious timeouts and multiple losses from the same flight of packets often coincide [GL02]. In such a case the oldest outstanding segment does arrive at the TCP receiver, but one or more packets from the remaining outstanding flight are lost. In those environments, TCP-Reno's performance suffers if the Eifel response algorithm is operated without an advanced loss recovery scheme such as NewReno [RFC2582], or SACK-based schemes [RFC2018], [RFC3517]. The reason is TCP-Reno's 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 all packets lost from that flight within a single round-trip time. On the contrary, the more conservative TCP-Reno/Eifel is often forced into another (backed-off) timeout.

However, in a more recent study [GL03], we found that the mentioned advanced loss recovery schemes are often too conservative to compete against TCP-Reno's blind go-back-N in terms of quickly recovering multiple losses after a spurious timeout. The problem with the NewReno scheme is that it does not exploit knowledge (e.g., provided through SACK options) about which segments were lost. The problem with the conservative SACK-based scheme [RFC3517] is that it waits for three SACKs before it retransmits a lost segment. This may often lead to a second - and in this case genuine - (potentially backed-off) timeout. In those cases TCP-Reno's loss recovery is often quicker due the blind go-back-N. This could be viewed as a disincentive to the deployment of the Eifel response algorithm.

We therefore suggest that a TCP sender MAY implement an optimistic (non-conservative) form of advanced loss recovery after a spurious timeout has been detected, if the following guidelines are met:

- Packet Conservation: The TCP sender may not have more segments (counting both original transmits and retransmits) in flight than indicated by the congestion window.

- A retransmit may only be sent when a potential loss has been indicated. For example, a single duplicate ACK is such an indication; potentially with the corresponding SACK info in case the SACK option is enabled for the connection.

We have developed and evaluated such a scheme (a variant of NewReno that exploits SACK info) in [GL03] that shows good results.

5. IPR Considerations

The IETF has been notified of intellectual property rights claimed in regard to some or all of the specification contained in this document. For more information consult the online list of claimed rights at http://www.ietf.org/ipr.

6. Security Considerations

There is a risk that a detection algorithm is fooled by spoofed ACKs that make genuine retransmits appear to the TCP sender as spurious retransmits. When such a detection algorithm is run together with the Eifel response algorithm, this could effectively disable congestion control at the TCP sender. Should this become a concern, the Eifel response algorithm SHOULD only be run together with detection algorithms that are known to be safe against such "ACK spoofing attacks".

For example, the safe variant of the Eifel detection algorithm [RFC3522], is a reliable method to protect against this risk.

Acknowledgments

Many thanks to Keith Sklower, Randy Katz, Michael Meyer, Stephan Baucke, Sally Floyd, Vern Paxson, Mark Allman, Ethan Blanton, Pasi Sarolahti, Alexey Kuznetsov, and Yogesh Swami for many discussions that contributed to this work.

Normative References

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

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

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

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

[RFC2861] Handley, M., Padhye, J. and S. Floyd, TCP Congestion Window

Validation, RFC 2861, June 2000.

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

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

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

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

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

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

Informative References

[BA02] Blanton, E. and M. Allman, On Making TCP More Robust to Packet Reordering, ACM Computer Communication Review, Vol. 32, No. 1, January 2002.

[BA03] Blanton, E. and M. Allman, Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions, draft-ietf-tsvwg-dsack-use-02.txt (work in progress), October 2003.

[RFC3517] Blanton, E., Allman, M., Fall, K. and L. Wang, A Conservative SACK-based Loss Recovery Algorithm for TCP, RFC3517, April 2003.

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

Wireless Conference, February 2002.

[GL03] Gurtov, A. and R. Ludwig, Responding to Spurious Timeouts in TCP, In Proceedings of IEEE INFOCOM 03, .

[Jac88] Jacobson, V., Congestion Avoidance and Control, In Proceedings of ACM SIGCOMM 88.

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

[LK00] Ludwig, R. and R. H. Katz, The Eifel Algorithm: Making TCP

Robust Against Spurious Retransmissions, ACM Computer Communication Review, Vol. 30, No. 1, January 2000.

[SK03] Sarolahti, P. and M. Kojo, F-RTO: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP, draft-ietf-tsvwg-tcp-frto-00.txt (work in progress), October 2003.

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

[Zh86] Zhang, L., 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
TeliaSonera Finland
P.O. Box 970, FIN-00051 Sonera
Helsinki, Finland
Email: andrei.gurtov@teliasonera.com
Homepage: http://www.cs.helsinki.fi/u/gurtov

This Internet-Draft expires in April 2004.