Network Working Group Reiner Ludwig
INTERNET-DRAFT Ericsson Research
Expires: September 2003 Andrei Gurtov
Sonera Corporation
March, 2003

The Eifel Response Algorithm for TCP
<draft-ietf-tsvwg-tcp-eifel-response-03.txt>

Status of this memo

Abstract

The Eifel response algorithm requires a detection algorithm to detect a posteriori whether the TCP sender has entered loss recovery unnecessarily. In response to a 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. 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', 'FlightSize', and 'Initial Window (IW)' 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. 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 [RFC***B]. That document discusses the relevant background and motivation that also applies to this document. Hence, the reader is expected to be familiar with [RFC***B]. Note that alternative response algorithms have been proposed [BDA03] that could also rely on the Eifel detection algorithm, and vice versa alternative detection algorithms have been proposed [BA02b], [SK03] that could work together with the Eifel response algorithm.

The Eifel response algorithm requires a detection algorithm to detect a posteriori whether the TCP sender has entered loss recovery unnecessarily. In response to a 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. 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. 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 to offer the possibility that their detection algorithms can work together with the Eifel response algorithm MUST reuse the variable SpuriousRecovery with the semantics and defined values as specified in [RFC***B]. In addition, we define LATE_SPUR_TO (equal -1) as another possible value of the variable SpuriousRecovery. Detection algorithms must set the value of SpuriousRecovery to LATE_SPUR_TO if the detection is based upon receiving the ACK for the retransmit. For example, this applies to detection algorithms that are based on the DSACK option.

3. 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.

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 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.

(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)
```

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

(RESP) If SpuriousRecovery equals FALSE, then proceed to step (DONE),

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

(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 change the calculation of SRTT to

```
SRTT <- SRTT + 1/FlightSize * (RTT-SAMPLE - SRTT)
```

and set

```
SRTT <- RTT-SAMPLE
RTTVAR <- RTT-SAMPLE/2,
```

recalculate the RTO, and restart the retransmission timer,

Note: Even after changing the calculation of SRTT, the retransmission timer is considered as being implemented according to [RFC2988].

else adapt the conservativeness of 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 <- min (pipe_prev, (FlightSize + IW))
    ssthresh <- pipe_prev

Proceed to step (DONE).

(DONE) No further processing.

3.2 Responding to Spurious Timeouts

3.2.1 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 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. 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.

3.2.2 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. However, since timestamps are being 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. 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. 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.

When the path's RTT varies largely, it is recommended to take RTT samples more frequently than only once per RTT. This allows the TCP sender to track changes in the RTT more closely. In particular, a TCP sender can react more quickly to sudden increases of the RTT by sooner updating the RTO to a more conservative value. The TCP Timestamps option [RFC1323] provides this capability, allowing the TCP sender to sample the RTT from every segment that is acknowledged. Using timestamps across such paths leads to a more conservative TCP retransmission timer and reduces the risk of triggering spurious timeouts [IMLGK02].

On the other hand, it is known that executing the RTO calculation defined in [RFC2988] more often than once per RTT leads to an RTO that decays too quickly, i.e., that converges to the RTT too quickly. This is because of the fixed gains (1/8 and 1/4) of RFC2988's RTT estimators. When timing every segment these gains are increasingly too large with an increasing FlightSize. This leads to the effect that the RTT estimators "lose" their memory too soon. This is a known conflict between [RFC2988] and [RFC1323]. Especially, a large RTO resulting from an RTT spike will decay within one or two RTTs (e.g., see [LS00]). Hence, simply reinitializing RFC2988's RTT estimators from RTT-SAMPLE is probably not enough to make the retransmission timer sufficiently conservative for at least the next couple of RTTs. A solution for the case when every segment is timed according to [RFC1323] is to make the gains adaptive to the FlightSize [LS00]. We suggest to adopt this solution for at least the SRTT.

3.3 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 may lead to fast timeouts [RFC***B], i.e., an event where the retransmission timer expires before the TCP sender receives the duplicate ACK that would trigger a fast retransmit of the oldest outstanding segment.

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].

3.4 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.

We suggest to restore cwnd to the minimum of the previous FlightSize, and the current FlightSize plus IW. The latter avoids large packet bursts that may occur with less careful variants for restoring congestion control state. For example, the original proposal [LK00] typically causes large bursts after packet reordering. The current proposal limits a potential packet burst to IW, which 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.

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.

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]. Note that we refer to the case were the oldest outstanding segment does arrive at the TCP receiver but one or more packets from the remaining outstanding flight are lost. We found that in such a case TCP-Reno's performance can even suffer if the Eifel response algorithm is operated without an advanced loss recovery scheme such as NewReno [RFC2582], or SACK-based schemes [2018], [RFC***A]. 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 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.

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 [RFC***A] 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.

> [Making TCP (even) more conservative by fixing a misbehavior in the name of 'packet conservation' would probably at most result in credits in the academic world.]

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.

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights

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 [RFC***B], 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, and Alexey Kuznetsov 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.

[RFC***B] R. Ludwig, M. Meyer, The Eifel Detection Algorithm for TCP, RFC***B, March 2003.

[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, Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions, draft-blanton-dsack-use-02.txt (work in progress), October 2002.

[BDA03] E. Blanton, R. Dimond, M. Allman. Practices for TCP Senders in the Face of Segment Reordering, draft-blanton-tcp-reordering-00.txt (work in progress), February 2003..

[RFC***A] E. Blanton, M. Allman, K. Fall, L. Wang, A Conservative SACK-based Loss Recovery Algorithm for TCP, RFC***A, March 2003.

[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.

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

[RFC3481] H. Inamura, G. Montenegro, R. Ludwig, A. Gurtov, F. Khafizov, TCP over Second (2.5G) and Third (3G) Generation Wireless Networks, RFC3481, February 2003.

[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.

[LS00] R. Ludwig, K. Sklower, The Eifel Retransmission Timer, ACM Computer Communication Review, Vol. 30, No. 3, July 2000.

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

[SK03] P. Sarolahti, M. Kojo, F-RTO: A TCP RTO Recovery Algorithm for Avoiding Unnecessary Retransmissions, draft-sarolahti-tsvwg-tcp-frto-03.txt (work in progress), January 2003.

[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 September 2003.