Network Working Group
Request for Comments: 3522
Category: Experimental

R. Ludwig
M. Meyer
Ericsson Research

April 2003

The Eifel Detection Algorithm for TCP

Status of this Memo

This memo defines an Experimental Protocol for the Internet community. It does not specify an Internet standard of any kind. Discussion and suggestions for improvement are requested. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The Eifel detection algorithm allows a TCP sender to detect a posteriori whether it has entered loss recovery unnecessarily. It requires that the TCP Timestamps option defined in RFC 1323 be enabled for a connection. The Eifel detection algorithm makes use of the fact that the TCP Timestamps option eliminates the retransmission ambiguity in TCP. Based on the timestamp of the first acceptable ACK that arrives during loss recovery, it decides whether loss recovery was entered unnecessarily. The Eifel detection algorithm provides a basis for future TCP enhancements. This includes response algorithms to back out of loss recovery by restoring a TCP sender's congestion control state.

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, with repacketization, 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 algorithm, this makes no difference as it also operates 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 a 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 a TCP sender to trigger a fast retransmit. Currently, DupThresh is specified as a fixed value of three [RFC2581]. Future TCPs might implement an adaptive DupThresh.

1. Introduction

The retransmission ambiguity problem [Zh86], [KP87] is a TCP sender's inability to distinguish whether the first acceptable ACK that arrives after a retransmit was sent in response to the original transmit or the retransmit. This problem occurs after a timeout-based retransmit and after a fast retransmit. The Eifel detection algorithm uses the TCP Timestamps option defined in [RFC1323] to eliminate the retransmission ambiguity. It thereby allows a TCP sender to detect a posteriori whether it has entered loss recovery unnecessarily.

This added capability of a TCP sender is useful in environments where TCP's loss recovery and congestion control algorithms may often get falsely triggered. This can be caused by packet reordering, packet duplication, or a sudden delay increase in the data or the ACK path that results in a spurious timeout. For example, such sudden delay increases can often occur in wide-area wireless access networks due to handovers, resource preemption due to higher priority traffic (e.g., voice), or because the mobile transmitter traverses through a radio coverage hole (e.g., see [Gu01]). In such wireless networks, the often unnecessary go-back-N retransmits that typically occur after a spurious timeout create a serious problem. They decrease end-to-end throughput, are useless load upon the network, and waste transmission (battery) power. Note that across such networks the use of timestamps is recommended anyway [RFC3481].

Based on the Eifel detection algorithm, a TCP sender may then choose to implement dedicated response algorithms. One goal of such a response algorithm would be to alleviate the consequences of a falsely triggered loss recovery. This may include restoring the TCP sender's congestion control state, and avoiding the mentioned unnecessary go-back-N retransmits. Another goal would be to adapt protocol parameters such as the duplicate acknowledgement threshold [RFC2581], and the RTT estimators [RFC2988]. This is to reduce the risk of falsely triggering TCP's loss recovery again as the connection progresses. However, such response algorithms are outside the scope of this document. Note: The original proposal, the "Eifel algorithm" [LK00], comprises both a detection and a response algorithm. This document only defines the detection part. The response part is defined in [LG03].

A key feature of the Eifel detection algorithm is that it already detects, upon the first acceptable ACK that arrives during loss recovery, whether a fast retransmit or a timeout was spurious. This is crucial to be able to avoid the mentioned go-back-N retransmits. Another feature is that the Eifel detection algorithm is fairly robust against the loss of ACKs.

Also the DSACK option [RFC2883] can be used to detect a posteriori whether a TCP sender has entered loss recovery unnecessarily [BA02]. However, the first ACK carrying a DSACK option usually arrives at a TCP sender only after loss recovery has already terminated. Thus, the DSACK option cannot be used to eliminate the retransmission ambiguity. Consequently, it cannot be used to avoid the mentioned unnecessary go-back-N retransmits. Moreover, a DSACK-based detection algorithm is less robust against ACK losses. A recent proposal based on neither the TCP timestamps nor the DSACK option does not have the limitation of DSACK-based schemes, but only addresses the case of spurious timeouts [SK03].

2. Events that Falsely Trigger TCP Loss Recovery

The following events may falsely trigger a TCP sender’s loss recovery and congestion control algorithms. This causes a so-called spurious retransmit, and an unnecessary reduction of the TCP sender’s congestion window and slow start threshold [RFC2581].

- Spurious timeout

- Packet reordering

- Packet duplication

A spurious timeout is a timeout that would not have occurred had the sender "waited longer". This may be caused by increased delay that suddenly occurs in the data and/or the ACK path. That in turn might cause an acceptable ACK to arrive too late, i.e., only after a TCP sender’s retransmission timer has expired. For the purpose of specifying the algorithm in Section 3, we define this case as SPUR_TO (equal 1).

> Note: There is another case where a timeout would not have occurred had the sender "waited longer": 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. We call this a 'fast timeout', since in competition with the fast retransmit algorithm the timeout was faster. However, a fast timeout is not spurious since apparently a segment was in fact lost, i.e., loss recovery was initiated rightfully. In this document, we do not consider fast timeouts.

Packet reordering in the network may occur because IP [RFC791] does not guarantee in-order delivery of packets. Additionally, a TCP receiver generates a duplicate ACK for each segment that arrives out-of-order. This results in a spurious fast retransmit if three or more data segments arrive out-of-order at a TCP receiver, and at least three of the resulting duplicate ACKs arrive at the TCP sender. This assumes that the duplicate acknowledgement threshold is set to three as defined in [RFC2581].

Packet duplication may occur because a receiving IP does not (cannot) remove packets that have been duplicated in the network. A TCP receiver in turn also generates a duplicate ACK for each duplicate segment. As with packet reordering, this results in a spurious fast retransmit if duplication of data segments or ACKs results in three or more duplicate ACKs to arrive at a TCP sender. Again, this assumes that the duplicate acknowledgement threshold is set to three.

The negative impact on TCP performance caused by packet reordering and packet duplication is commonly the same: a single spurious retransmit (the fast retransmit), and the unnecessary halving of a TCP sender's congestion window as a result of the subsequent fast recovery phase [RFC2581].

The negative impact on TCP performance caused by a spurious timeout is more severe. First, the timeout event itself causes a single spurious retransmit, and unnecessarily forces a TCP sender into slow start [RFC2581]. Then, as the connection progresses, a chain reaction gets triggered that further decreases TCP's performance. Since the timeout was spurious, at least some ACKs for original transmits typically arrive at the TCP sender before the ACK for the retransmit arrives. (This is unless severe packet reordering coincided with the spurious timeout in such a way that the ACK for the retransmit is the first acceptable ACK to arrive at the TCP sender.) Those ACKs for original transmits then trigger an implicit go-back-N loss recovery at the TCP sender [LK00]. Assuming that none of the outstanding segments and none of the corresponding ACKs were lost, all outstanding segments get retransmitted unnecessarily. In fact, during this phase, a TCP sender violates the packet conservation principle [Jac88]. This is because the unnecessary go-back-N retransmits are sent during slow start. Thus, for each packet that leaves the network and that belongs to the first half of the original flight, two useless retransmits are sent into the network. In addition, some TCPs suffer from a spurious fast retransmit. This is because the unnecessary go-back-N retransmits arrive as duplicates at the TCP receiver, which in turn triggers a series of duplicate ACKs. Note that this last spurious fast retransmit could be avoided with the careful variant of 'bugfix' [RFC2582].

More detailed explanations, including TCP trace plots that visualize the effects of spurious timeouts and packet reordering, can be found in the original proposal [LK00].

3. The Eifel Detection Algorithm

3.1 The Idea

The goal of the Eifel detection algorithm is to allow a TCP sender to detect a posteriori whether it has entered loss recovery unnecessarily. Furthermore, the TCP sender should be able to make this decision upon the first acceptable ACK that arrives after the timeout-based retransmit or the fast retransmit has been sent. This in turn requires extra information in ACKs by which the TCP sender can unambiguously distinguish whether that first acceptable ACK was sent in response to the original transmit or the retransmit. Such extra information is provided by the TCP Timestamps option [RFC1323]. Generally speaking, timestamps are monotonously increasing "serial numbers" added into every segment that are then echoed within the corresponding ACKs. This is exploited by the Eifel detection algorithm in the following way.

Given that timestamps are enabled for a connection, a TCP sender always stores the timestamp of the retransmit sent in the beginning of loss recovery, i.e., the timestamp of the timeout-based retransmit or the fast retransmit. If the timestamp of the first acceptable ACK, that arrives after the retransmit was sent, is smaller then the stored timestamp of that retransmit, then that ACK must have been sent in response to an original transmit. Hence, the TCP sender must have entered loss recovery unnecessarily.

The fact that the Eifel detection algorithm decides upon the first acceptable ACK is crucial to allow future response algorithms to avoid the unnecessary go-back-N retransmits that typically occur after a spurious timeout. Also, if loss recovery was entered unnecessarily, a window worth of ACKs are outstanding that all carry a timestamp that is smaller than the stored timestamp of the retransmit. The arrival of any one of those ACKs is sufficient for the Eifel detection algorithm to work. Hence, the solution is fairly robust against ACK losses. Even the ACK sent in response to the retransmit, i.e., the one that carries the stored timestamp, may get lost without compromising the algorithm.

3.2 The Algorithm

Given that the TCP Timestamps option [RFC1323] is enabled for a connection, a TCP sender MAY use the Eifel detection algorithm as defined in this subsection.

If the Eifel detection algorithm is used, the following steps MUST be taken by a TCP sender, but only upon initiation of loss recovery, i.e., when either the timeout-based retransmit or the fast retransmit is sent. The Eifel detection algorithm MUST NOT be reinitiated after loss recovery has already started. In particular, it must not be reinitiated upon subsequent timeouts for the same segment, and not upon retransmitting segments other than the oldest outstanding segment, e.g., during selective loss recovery.

(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 RetransmitTS, then proceed to step (5),

else proceed to step (DONE).

(5) If the acceptable ACK carries a DSACK option [RFC2883], then proceed to step (DONE),

else if during the lifetime of the TCP connection the TCP sender has previously received an ACK with a DSACK option, or the acceptable ACK does not acknowledge all outstanding data, 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)  Do nothing (Placeholder for a response algorithm).

   (DONE)  No further processing.
```

The comparison "smaller than" in step (4) is conservative. In theory, if the timestamp clock is slow or the network is fast, RetransmitTS could at most be equal to the timestamp echoed by an ACK sent in response to an original transmit. In that case, it is assumed that the loss recovery was not falsely triggered.

Note that the condition "if during the lifetime of the TCP connection the TCP sender has previously received an ACK with a DSACK option" in step (5) would be true in case the TCP receiver would signal in the SYN that it is DSACK-enabled. But unfortunately, this is not required by [RFC2883].

3.3 A Corner Case: "Timeout due to loss of all ACKs" (step 5)

Even though the oldest outstanding segment arrived at a TCP receiver, the TCP sender is forced into a timeout if all ACKs are lost. Although the resulting retransmit is unnecessary, such a timeout is unavoidable. It should therefore not be considered spurious. Moreover, the subsequent reduction of the congestion window is an appropriate response to the potentially heavy congestion in the ACK path. The original proposal [LK00] does not handle this case well. It effectively disables this implicit form of congestion control for the ACK path, which otherwise does not exist in TCP. This problem is fixed by step (5) of the Eifel detection algorithm as explained in the remainder of this section.

If all ACKs are lost while the oldest outstanding segment arrived at the TCP receiver, the retransmit arrives as a duplicate. In response to duplicates, RFC 1323 mandates that the timestamp of the last segment that arrived in-sequence should be echoed. That timestamp is carried by the first acceptable ACK that arrives at the TCP sender after loss recovery was entered, and is commonly smaller than the timestamp carried by the retransmit. Consequently, the Eifel detection algorithm misinterprets such a timeout as being spurious, unless the TCP receiver is DSACK-enabled [RFC2883]. In that case, the acceptable ACK carries a DSACK option, and the Eifel algorithm is terminated through the first part of step (5).

Note: Not all TCP implementations strictly follow RFC 1323. In response to a duplicate data segment, some TCP receivers echo the timestamp of the duplicate. With such TCP receivers, the corner case discussed in this section does not apply. The timestamp carried by the retransmit would be echoed in the first acceptable ACK, and the Eifel detection algorithm would be terminated through step (4). Thus, even though all ACKs were lost and independent of whether the DSACK option was enabled for a connection, the Eifel detection algorithm would have no effect.

With TCP receivers that are not DSACK-enabled, disabling the mentioned implicit congestion control for the ACK path is not a problem as long as data segments are lost, in addition to the entire flight of ACKs. The Eifel detection algorithm misinterprets such a timeout as being spurious, and the Eifel response algorithm would reverse the congestion control state. Still, the TCP sender would respond to congestion (in the data path) as soon as it finds out about the first loss in the outstanding flight. I.e., the TCP sender would still halve its congestion window for that flight of packets. If no data segment is lost while the entire flight of ACKs is lost, the first acceptable ACK that arrives at the TCP sender after loss recovery was entered acknowledges all outstanding data. In that case, the Eifel algorithm is terminated through the second part of step (5).

Note that there is little concern about violating the packet conservation principle when entering slow start after an unavoidable timeout caused by the loss of an entire flight of ACKs, i.e., when the Eifel detection algorithm was terminated through step (5). This is because in that case, the acceptable ACK corresponds to the retransmit, which is a strong indication that the pipe has drained entirely, i.e., that no more original transmits are in the network. This is different with spurious timeouts as discussed in Section 2.

3.4 Protecting Against Misbehaving TCP Receivers (the Safe Variant)

A TCP receiver can easily make a genuine retransmit appear to the TCP sender as a spurious retransmit by forging echoed timestamps. This may pose a security concern.

Fortunately, there is a way to modify the Eifel detection algorithm in a way that makes it robust against lying TCP receivers. The idea is to use timestamps as a segment’s "secret" that a TCP receiver only gets to know if it receives the segment. Conversely, a TCP receiver will not know the timestamp of a segment that was lost. Hence, to "prove" that it received the original transmit of a segment that a TCP sender retransmitted, the TCP receiver would need to return the timestamp of that original transmit. The Eifel detection algorithm could then be modified to only decide that loss recovery has been unnecessarily entered if the first acceptable ACK echoes the timestamp of the original transmit.

Hence, implementers may choose to implement the algorithm with the following modifications.

Step (2) is replaced with step (2’):

(2’) Set a "RetransmitTS" variable to the value of the Timestamp Value field of the Timestamps option that was included in the original transmit corresponding to the retransmit. Note: This step requires that the TCP sender stores the timestamps of all outstanding original transmits.

Step (4) is replaced with step (4’):

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

else proceed to step (DONE).

These modifications come at a cost: the modified algorithm is fairly sensitive against ACK losses since it relies on the arrival of the acceptable ACK that corresponds to the original transmit.

Note: The first acceptable ACK that arrives after loss recovery has been unnecessarily entered should echo the timestamp of the original transmit. This assumes that the ACK corresponding to the original transmit was not lost, that that ACK was not reordered in the network, and that the TCP receiver does not forge timestamps but complies with RFC 1323. In case of a spurious fast retransmit, this is implied by the rules for generating ACKs for data segments that fill in all or part of a gap in the sequence space (see section 4.2 of [RFC2581]) and by the rules for echoing timestamps in that case (see rule (C) in section 3.4 of [RFC1323]). In case of a spurious timeout, it is likely that the delay that has caused the spurious timeout has also caused the TCP receiver’s delayed ACK timer [RFC1122] to expire before the original transmit arrives. Also, in this case the rules for generating ACKs and the rules for echoing timestamps (see rule (A) in section 3.4 of [RFC1323]) ensure that the original transmit’s timestamp is echoed.

A remaining problem is that a TCP receiver might guess a lost segment's timestamp from observing the timestamps of recently received segments. For example, if segment N was lost while segment N-1 and N+1 have arrived, a TCP receiver could guess the timestamp that lies in the middle of the timestamps of segments N-1 and N+1, and echo it in the ACK sent in response to the retransmit of segment N. Especially if the TCP sender implements timestamps with a coarse granularity, a misbehaving TCP receiver is likely to be successful with such an approach. In fact, with the 500 ms granularity suggested in [WS95], it even becomes quite likely that the timestamps of segments N-1, N, N+1 are identical.

One way to reduce this risk is to implement fine grained timestamps. Note that the granularity of the timestamps is independent of the granularity of the retransmission timer. For example, some TCP implementations run a timestamp clock that ticks every millisecond. This should make it more difficult for a TCP receiver to guess the timestamp of a lost segment. Alternatively, it might be possible to combine the timestamps with a nonce, as is done for the Explicit Congestion Notification (ECN) [RFC3168]. One would need to take care, though, that the timestamps of consecutive segments remain monotonously increasing and do not interfere with the RTT timing defined in [RFC1323].

4. 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 might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

5. Security Considerations

There do not seem to be any security considerations associated with the Eifel detection algorithm. This is because the Eifel detection algorithm does not alter the existing protocol state at a TCP sender. Note that the Eifel detection algorithm only requires changes to the implementation of a TCP sender.

Moreover, a variant of the Eifel detection algorithm has been proposed in Section 3.4 that makes it robust against lying TCP receivers. This may become relevant when the Eifel detection algorithm is combined with a response algorithm such as the Eifel response algorithm [LG03].

Acknowledgments

Many thanks to Keith Sklower, Randy Katz, Stephan Baucke, Sally Floyd, Vern Paxson, Mark Allman, Ethan Blanton, Andrei Gurtov, Pasi Sarolahti, and Alexey Kuznetsov for useful 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", BCP 14, RFC 2119, March 1997.

[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.

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

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

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

Informative References

[BA02] Blanton, E. and M. Allman, "Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions", Work in Progress.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

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

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

[RFC3481] Inamura, H., Montenegro, G., Ludwig, R., Gurtov, A. and F. Khafizov, "TCP over Second (2.5G) and Third (3G) Generation Wireless Networks", RFC 3481, February 2003.

[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.

[LG03] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", Work in Progress.

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

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

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

[SK03] Sarolahti, P. and M. Kojo, "F-RTO: A TCP RTO Recovery Algorithm for Avoiding Unnecessary Retransmissions", Work in Progress.

[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 86.

Authors' Addresses

Reiner Ludwig
Ericsson Research
Ericsson Allee 1
52134 Herzogenrath, Germany

EMail: Reiner.Ludwig@eed.ericsson.se

Michael Meyer
Ericsson Research
Ericsson Allee 1
52134 Herzogenrath, Germany

EMail: Michael.Meyer@eed.ericsson.se

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.