Network Working Group
Request for Comments: 4138
Category: Experimental

P. Sarolahti
Nokia Research Center
M. Kojo
University of Helsinki

August 2005

Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)

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

Spurious retransmission timeouts cause suboptimal TCP performance because they often result in unnecessary retransmission of the last window of data. This document describes the F-RTO detection algorithm for detecting spurious TCP retransmission timeouts. F-RTO is a TCP sender-only algorithm that does not require any TCP options to operate. After retransmitting the first unacknowledged segment triggered by a timeout, the F-RTO algorithm of the TCP sender monitors the incoming acknowledgments to determine whether the timeout was spurious. It then decides whether to send new segments or retransmit unacknowledged segments. The algorithm effectively helps to avoid additional unnecessary retransmissions and thereby improves TCP performance in the case of a spurious timeout. The F-RTO algorithm can also be applied to the Stream Control Transmission Protocol (SCTP).

Table of Contents

1. Introduction

The Transmission Control Protocol (TCP) [Pos81] has two methods for triggering retransmissions. First, the TCP sender relies on incoming duplicate ACKs, which indicate that the receiver is missing some of the data. After a required number of successive duplicate ACKs have arrived at the sender, it retransmits the first unacknowledged segment [APS99] and continues with a loss recovery algorithm such as NewReno [FHG04] or SACK-based loss recovery [BAFW03]. Second, the TCP sender maintains a retransmission timer which triggers retransmission of segments, if they have not been acknowledged before the retransmission timeout (RTO) expires. When the retransmission timeout occurs, the TCP sender enters the RTO recovery where the congestion window is initialized to one segment and unacknowledged segments are retransmitted using the slow-start algorithm. The retransmission timer is adjusted dynamically, based on the measured round-trip times [PA00].

It has been pointed out that the retransmission timer can expire spuriously and cause unnecessary retransmissions when no segments have been lost [LK00, GL02, LM03]. After a spurious retransmission timeout, the late acknowledgments of the original segments arrive at the sender, usually triggering unnecessary retransmissions of a whole window of segments during the RTO recovery. Furthermore, after a spurious retransmission timeout, a conventional TCP sender increases the congestion window on each late acknowledgment in slow start. This injects a large number of data segments into the network within one round-trip time, thus violating the packet conservation principle [Jac88].

There are a number of potential reasons for spurious retransmission timeouts. First, some mobile networking technologies involve sudden delay spikes on transmission because of actions taken during a hand-off. Second, given a low-bandwidth link or some other change in available bandwidth, arrival of competing traffic (possibly with higher priority) can cause a sudden increase of round-trip time. This may trigger a spurious retransmission timeout. A persistently reliable link layer can also cause a sudden delay when a data frame and several retransmissions of it are lost for some reason. This document does not distinguish between the different causes of such a delay spike. Rather, it discusses the spurious retransmission timeouts caused by a delay spike in general.

This document describes the F-RTO detection algorithm. It is based on the detection mechanism of the "Forward RTO-Recovery" (F-RTO) algorithm [SKR03] that is used for detecting spurious retransmission timeouts and thus avoids unnecessary retransmissions following the retransmission timeout. When the timeout is not spurious, the F-RTO algorithm reverts back to the conventional RTO recovery algorithm, and therefore has similar behavior and performance. In contrast to alternative algorithms proposed for detecting unnecessary retransmissions (Eifel [LK00], [LM03] and DSACK-based algorithms [BA04]), F-RTO does not require any TCP options for its operation, and it can be implemented by modifying only the TCP sender. The Eifel algorithm uses TCP timestamps [BBJ92] for detecting a spurious timeout upon arrival of the first acknowledgment after the retransmission. The DSACK-based algorithms require that the TCP Selective Acknowledgment Option [MMFR96], with the DSACK extension [FMMP00], is in use. With DSACK, the TCP receiver can report if it has received a duplicate segment, enabling the sender to detect afterwards whether it has retransmitted segments unnecessarily. The F-RTO algorithm only attempts to detect and avoid unnecessary retransmissions after an RTO. Eifel and DSACK can also be used for detecting unnecessary retransmissions caused by other events, such as packet reordering.

When an RTO expires, the F-RTO sender retransmits the first unacknowledged segment as usual [APS99]. Deviating from the normal operation after a timeout, it then tries to transmit new, previously unsent data, for the first acknowledgment that arrives after the timeout, given that the acknowledgment advances the window. If the second acknowledgment that arrives after the timeout advances the window (i.e., acknowledges data that was not retransmitted), the F-RTO sender declares the timeout spurious and exits the RTO recovery. However, if either of these two acknowledgments is a duplicate ACK, there will not be sufficient evidence of a spurious timeout. Therefore, the F-RTO sender retransmits the unacknowledged segments in slow start similarly to the traditional algorithm. With a

SACK-enhanced version of the F-RTO algorithm, spurious timeouts may be detected even if duplicate ACKs arrive after an RTO retransmission.

The F-RTO algorithm can also be applied to the Stream Control Transmission Protocol (SCTP) [Ste00], because SCTP has acknowledgment and packet retransmission concepts similar to TCP. For convenience, this document mostly refers to TCP, but the algorithms and other discussion are valid for SCTP as well.

This document is organized as follows. Section 2 describes the basic F-RTO algorithm. Section 3 outlines an optional enhancement to the F-RTO algorithm that takes advantage of the TCP SACK option. Section 4 discusses the possible actions to be taken after detecting a spurious RTO. Section 5 gives considerations on applying F-RTO with SCTP, and Section 6 discusses the security considerations.

1.1. 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].

2. F-RTO Algorithm

A timeout is considered spurious if it would have been avoided had the sender waited longer for an acknowledgment to arrive [LM03]. F-RTO affects the TCP sender behavior only after a retransmission timeout. Otherwise, the TCP behavior remains the same. When the RTO expires, the F-RTO algorithm monitors incoming acknowledgments and if the TCP sender gets an acknowledgment for a segment that was not retransmitted due to timeout, the F-RTO algorithm declares a timeout spurious. The actions taken in response to a spurious timeout are not specified in this document, but we discuss some alternatives in Section 4. This section introduces the algorithm and then discusses the different steps of the algorithm in more detail.

Following the practice used with the Eifel Detection algorithm [LM03], we use the "SpuriousRecovery" variable to indicate whether the retransmission is declared spurious by the sender. This variable can be used as an input for a corresponding response algorithm. With F-RTO, the value of SpuriousRecovery can be either SPUR_TO (indicating a spurious retransmission timeout) or FALSE (indicating that the timeout is not declared spurious), and the TCP sender should follow the conventional RTO recovery algorithm.

2.1. The Algorithm

A TCP sender MAY implement the basic F-RTO algorithm. If it chooses to apply the algorithm, the following steps MUST be taken after the retransmission timer expires. If the sender implements some loss recovery algorithm other than Reno or NewReno [FHG04], the F-RTO algorithm SHOULD NOT be entered when earlier fast recovery is underway.

1) When RTO expires, retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Also, store the highest sequence number transmitted so far in variable "recover".

2) When the first acknowledgment after the RTO retransmission arrives at the sender, the sender chooses one of the following actions, depending on whether the ACK advances the window or whether it is a duplicate ACK.

   a) If the acknowledgment is a duplicate ACK OR it acknowledges a sequence number equal to the value of "recover" OR it does not acknowledge all of the data that was retransmitted in step 1, revert to the conventional RTO recovery and continue by retransmitting unacknowledged data in slow start. Do not enter step 3 of this algorithm. The SpuriousRecovery variable remains as FALSE.

   b) Else, if the acknowledgment advances the window AND it is below the value of "recover", transmit up to two new (previously unsent) segments and enter step 3 of this algorithm. If the TCP sender does not have enough unsent data, it can send only one segment. In addition, the TCP sender MAY override the Nagle algorithm [Nag84] and immediately send a segment if needed. Note that sending two segments in this step is allowed by TCP congestion control requirements [APS99]: An F-RTO TCP sender simply chooses different segments to transmit.

      If the TCP sender does not have any new data to send, or the advertised window prohibits new transmissions, the recommended action is to skip step 3 of this algorithm and continue with slow start retransmissions, following the conventional RTO recovery algorithm. However, alternative ways of handling the window-limited cases that could result in better performance are discussed in Appendix C.

3) When the second acknowledgment after the RTO retransmission arrives at the sender, the TCP sender either declares the timeout spurious, or starts retransmitting the unacknowledged segments.

a) If the acknowledgment is a duplicate ACK, set the congestion window to no more than 3 * MSS, and continue with the slow start algorithm retransmitting unacknowledged segments. The congestion window can be set to 3 * MSS, because two round-trip times have elapsed since the RTO, and a conventional TCP sender would have increased cwnd to 3 during the same time. Leave SpuriousRecovery set to FALSE.

b) If the acknowledgment advances the window (i.e., if it acknowledges data that was not retransmitted after the timeout), declare the timeout spurious, set SpuriousRecovery to SPUR_TO, and set the value of the "recover" variable to SND.UNA (the oldest unacknowledged sequence number [Pos81]).

2.2. Discussion

The F-RTO sender takes cautious actions when it receives duplicate acknowledgments after a retransmission timeout. Because duplicate ACKs may indicate that segments have been lost, reliably detecting a spurious timeout is difficult due to the lack of additional information. Therefore, it is prudent to follow the conventional TCP recovery in those cases.

If the first acknowledgment after the RTO retransmission covers the "recover" point at algorithm step (2a), there is not enough evidence that a non-retransmitted segment has arrived at the receiver after the timeout. This is a common case when a fast retransmission is lost and has been retransmitted again after an RTO, while the rest of the unacknowledged segments were successfully delivered to the TCP receiver before the retransmission timeout. Therefore, the timeout cannot be declared spurious in this case.

If the first acknowledgment after the RTO retransmission does not acknowledge all of the data that was retransmitted in step 1, the TCP sender reverts to the conventional RTO recovery. Otherwise, a malicious receiver acknowledging partial segments could cause the sender to declare the timeout spurious in a case where data was lost.

The TCP sender is allowed to send two new segments in algorithm branch (2b) because the conventional TCP sender would transmit two segments when the first new ACK arrives after the RTO retransmission. If sending new data is not possible in algorithm branch (2b), or if the receiver window limits the transmission, the TCP sender has to send something in order to prevent the TCP transfer from stalling. If no segments were sent, the pipe between sender and receiver might run out of segments, and no further acknowledgments would arrive. Therefore, in the window-limited case, the recommendation is to revert to the conventional RTO recovery with slow start retransmissions. Appendix C discusses some alternative solutions for window-limited situations.

If the retransmission timeout is declared spurious, the TCP sender sets the value of the "recover" variable to SND.UNA in order to allow fast retransmit [FHG04]. The "recover" variable was proposed for avoiding unnecessary, multiple fast retransmits when RTO expires during fast recovery with NewReno TCP. Because the sender retransmits only the segment that triggered the timeout, the problem of unnecessary multiple fast retransmits [FHG04] cannot occur. Therefore, if three duplicate ACKs arrive at the sender after the timeout, they probably indicate a packet loss, and thus fast retransmit should be used to allow efficient recovery. If there are not enough duplicate ACKs arriving at the sender after a packet loss, the retransmission timer expires again and the sender enters step 1 of this algorithm.

When the timeout is declared spurious, the TCP sender cannot detect whether the unnecessary RTO retransmission was lost. In principle, the loss of the RTO retransmission should be taken as a congestion signal. Thus, there is a small possibility that the F-RTO sender will violate the congestion control rules, if it chooses to fully revert congestion control parameters after detecting a spurious timeout. The Eifel detection algorithm has a similar property, while the DSACK option can be used to detect whether the retransmitted segment was successfully delivered to the receiver.

The F-RTO algorithm has a side-effect on the TCP round-trip time measurement. Because the TCP sender can avoid most of the unnecessary retransmissions after detecting a spurious timeout, the sender is able to take round-trip time samples on the delayed segments. If the regular RTO recovery was used without TCP timestamps, this would not be possible due to the retransmission ambiguity. As a result, the RTO is likely to have more accurate and larger values with F-RTO than with the regular TCP after a spurious timeout that was triggered due to delayed segments. We believe this is an advantage in the networks that are prone to delay spikes.

There are some situations where the F-RTO algorithm may not avoid unnecessary retransmissions after a spurious timeout. If packet reordering or packet duplication occurs on the segment that triggered the spurious timeout, the F-RTO algorithm may not detect the spurious timeout due to incoming duplicate ACKs. Additionally, if a spurious timeout occurs during fast recovery, the F-RTO algorithm often cannot detect the spurious timeout because the segments that were transmitted before the fast recovery trigger duplicate ACKs. However, we consider these cases rare, and note that in cases where

F-RTO fails to detect the spurious timeout, it retransmits the unacknowledged segments in slow start, and thus performs similarly to the regular RTO recovery.

3. SACK-Enhanced Version of the F-RTO Algorithm

This section describes an alternative version of the F-RTO algorithm that uses the TCP Selective Acknowledgment Option [MMFR96]. By using the SACK option, the TCP sender detects spurious timeouts in most of the cases when packet reordering or packet duplication is present. If the SACK blocks acknowledge new data that was not transmitted after the RTO retransmission, the sender may declare the timeout spurious, even when duplicate ACKs follow the RTO.

Given that the TCP Selective Acknowledgment Option [MMFR96] is enabled for a TCP connection, a TCP sender MAY implement the SACK-enhanced F-RTO algorithm. If the sender applies the SACK-enhanced F-RTO algorithm, it MUST follow the steps below. This algorithm SHOULD NOT be applied if the TCP sender is already in SACK loss recovery when retransmission timeout occurs. However, when retransmission timeout occurs during existing loss recovery, it should be possible to apply the principle of F-RTO within certain limitations. This is a topic for further research. Appendix B briefly discusses the related issues.

The steps of the SACK-enhanced version of the F-RTO algorithm are as follows.

1) When the RTO expires, retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Set variable "recover" to indicate the highest segment transmitted so far. Following the recommendation in SACK specification [MMFR96], reset the SACK scoreboard.

2) Wait until the acknowledgment of the data retransmitted due to the timeout arrives at the sender. If duplicate ACKs arrive before the cumulative acknowledgment for retransmitted data, adjust the scoreboard according to the incoming SACK information. Stay in step 2 and wait for the next new acknowledgment. If RTO expires again, go to step 1 of the algorithm.

   a) if a cumulative ACK acknowledges a sequence number equal to "recover", revert to the conventional RTO recovery and set the congestion window to no more than 2 * MSS, like a regular TCP would do. Do not enter step 3 of this algorithm.

b) else, if a cumulative ACK acknowledges a sequence number (smaller than "recover", but larger than SND.UNA) transmit up to two new (previously unsent) segments and proceed to step 3. If the TCP sender is not able to transmit any previously unsent data -- either due to receiver window limitation, or because it does not have any new data to send -- the recommended action is to refrain from entering step 3 of this algorithm. Rather, continue with slow start retransmissions following the conventional RTO recovery algorithm.

It is also possible to apply some of the alternatives for handling window-limited cases discussed in Appendix C. In this case, the TCP sender should follow the recommendations concerning acknowledgments of retransmitted segments given in Appendix B.

3) The next acknowledgment arrives at the sender. Either a duplicate ACK or a new cumulative ACK (advancing the window) applies in this step.

a) if the ACK acknowledges a sequence number above "recover", either in SACK blocks or as a cumulative ACK, set the congestion window to no more than 3 * MSS and proceed with the conventional RTO recovery, retransmitting unacknowledged segments. Take this branch also when the acknowledgment is a duplicate ACK and it does not acknowledge any new, previously unacknowledged data below "recover" in the SACK blocks. Leave SpuriousRecovery set to FALSE.

b) if the ACK does not acknowledge sequence numbers above "recover" AND it acknowledges data that was not acknowledged earlier (either with cumulative acknowledgment or using SACK blocks), declare the timeout spurious and set SpuriousRecovery to SPUR_TO. The retransmission timeout can be declared spurious, because the segment acknowledged with this ACK was transmitted before the timeout.

If there are unacknowledged holes between the received SACK blocks, those segments are retransmitted similarly to the conventional SACK recovery algorithm [BAFW03]. If the algorithm exits with SpuriousRecovery set to SPUR_TO, "recover" is set to SND.UNA, thus allowing fast recovery on incoming duplicate acknowledgments.

4. Taking Actions after Detecting Spurious RTO

Upon retransmission timeout, a conventional TCP sender assumes that outstanding segments are lost and starts retransmitting the unacknowledged segments. When the retransmission timeout is detected to be spurious, the TCP sender should not continue retransmitting based on the timeout. For example, if the sender was in congestion avoidance phase transmitting new, previously unsent segments, it should continue transmitting previously unsent segments after detecting a spurious RTO. This document does not describe the response to spurious timeouts, but a response algorithm is described in RFC 4015 [LG04].

Additionally, different response variants to spurious retransmission timeout have been discussed in various research papers [SKR03, GL03, Sar03] and IETF documents [SL03]. The different response alternatives vary in whether the spurious retransmission timeout should be taken as a congestion signal, thus causing the congestion window or slow start threshold to be reduced at the sender, or whether the congestion control state should be fully reverted to the state valid prior to the retransmission timeout.

5. SCTP Considerations

SCTP has similar retransmission algorithms and congestion control to TCP. The SCTP T3-rtx timer for one destination address is maintained in the same way as the TCP retransmission timer, and after a T3-rtx expires, an SCTP sender retransmits unacknowledged data chunks in slow start like TCP does. Therefore, SCTP is vulnerable to the negative effects of the spurious retransmission timeouts similarly to TCP. Due to similar RTO recovery algorithms, F-RTO algorithm logic can be applied also to SCTP. Since SCTP uses selective acknowledgments, the SACK-based variant of the algorithm is recommended, although the basic version can also be applied to SCTP. However, SCTP contains features that are not present with TCP that need to be discussed when applying the F-RTO algorithm.

SCTP associations can be multi-homed. The current retransmission policy states that retransmissions should go to alternative addresses. If the retransmission was due to spurious timeout caused by a delay spike, it is possible that the acknowledgment for the retransmission arrives back at the sender before the acknowledgments of the original transmissions arrive. If this happens, a possible loss of the original transmission of the data chunk that was retransmitted due to the spurious timeout may remain undetected when applying the F-RTO algorithm. Because the timeout was caused by a delay spike, and it was spurious in that respect, a suitable response is to continue by sending new data. However, if the original transmission was lost, fully reverting the congestion control parameters is too aggressive. Therefore, taking conservative actions on congestion control is recommended, if the SCTP association is multi-homed and retransmissions go to alternative addresses. The information in duplicate TSNs can be then used for reverting congestion control, if desired [BA04].

Note that the forward transmissions made in F-RTO algorithm step (2b) should be destined to the primary address, since they are not retransmissions.

When making a retransmission, an SCTP sender can bundle a number of unacknowledged data chunks and include them in the same packet. This needs to be considered when implementing F-RTO for SCTP. The basic principle of F-RTO still holds: in order to declare the timeout spurious, the sender must get an acknowledgment for a data chunk that was not retransmitted after the retransmission timeout. In other words, acknowledgments of data chunks that were bundled in RTO retransmission must not be used for declaring the timeout spurious.

6. Security Considerations

The main security threat regarding F-RTO is the possibility that a receiver could mislead the sender into setting too large a congestion window after an RTO. There are two possible ways a malicious receiver could trigger a wrong output from the F-RTO algorithm. First, the receiver can acknowledge data that it has not received. Second, it can delay acknowledgment of a segment it has received earlier, and acknowledge the segment after the TCP sender has been deluded to enter algorithm step 3.

If the receiver acknowledges a segment it has not really received, the sender can be led to declare spurious timeout in the F-RTO algorithm, step 3. However, because the sender will have an incorrect state, it cannot retransmit the segment that has never reached the receiver. Therefore, this attack is unlikely to be useful for the receiver to maliciously gain a larger congestion window.

A common case for a retransmission timeout is that a fast retransmission of a segment is lost. If all other segments have been received, the RTO retransmission causes the whole window to be acknowledged at once. This case is recognized in F-RTO algorithm branch (2a). However, if the receiver only acknowledges one segment after receiving the RTO retransmission, and then the rest of the segments, it could cause the timeout to be declared spurious when it is not. Therefore, it is suggested that, when an RTO expires during fast recovery phase, the sender would not fully revert the congestion window even if the timeout was declared spurious. Instead, the sender would reduce the congestion window to 1.

If there is more than one segment missing at the time of a retransmission timeout, the receiver does not benefit from misleading the sender to declare a spurious timeout because the sender would have to go through another recovery period to retransmit the missing segments, usually after an RTO has elapsed.

7. Acknowledgements

We are grateful to Reiner Ludwig, Andrei Gurtov, Josh Blanton, Mark Allman, Sally Floyd, Yogesh Swami, Mika Liljeberg, Ivan Arias Rodriguez, Sourabh Ladha, Martin Duke, Motoharu Miyake, Ted Faber, Samu Kontinen, and Kostas Pentikousis for the discussion and feedback contributed to this text.

8. References

8.1. Normative References

[APS99] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.

[BAFW03] Blanton, E., Allman, M., Fall, K., and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC 3517, April 2003.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[FHG04] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno Modification to TCP’s Fast Recovery Algorithm", RFC 3782, April 2004.

[MMFR96] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgement Options", RFC 2018, October 1996.

[PA00] Paxson, V. and M. Allman, "Computing TCP’s Retransmission Timer", RFC 2988, November 2000.

[Pos81] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[Ste00] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

8.2. Informative References

[ABF01] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP’s Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[BA04] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, February 2004.

[BBJ92] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[FMMP00] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

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

[GL03] A. Gurtov and R. Ludwig, Responding to Spurious Timeouts in TCP. In Proceedings of IEEE INFOCOM 03, San Francisco, CA, USA, March 2003.

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

[LG04] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, February 2005.

[LK00] R. Ludwig and R.H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. ACM SIGCOMM Computer Communication Review, 30(1), January 2000.

[LM03] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

[Nag84] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, January 1984.

[SKR03] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts. ACM SIGCOMM Computer Communication Review, 33(2), April 2003.

[Sar03] P. Sarolahti. Congestion Control on Spurious TCP Retransmission Timeouts. In Proceedings of IEEE Globecom 2003, San Francisco, CA, USA. December 2003.

[SL03] Y. Swami and K. Le, "DCLOR: De-correlated Loss Recovery using SACK Option for Spurious Timeouts", work in progress, September 2003.

Appendix A: Scenarios

This section discusses different scenarios where RTOs occur and how the basic F-RTO algorithm performs in those scenarios. The interesting scenarios are: a sudden delay triggering retransmission timeout, loss of a retransmitted packet during fast recovery, link outage causing the loss of several packets, and packet reordering. A performance evaluation with a more thorough analysis on a real implementation of F-RTO is given in [SKR03].

A.1. Sudden Delay

The main motivation behind the F-RTO algorithm is to improve TCP performance when a delay spike triggers a spurious retransmission timeout. The example below illustrates the segments and acknowledgments transmitted by the TCP end hosts when a spurious timeout occurs, but no packets are lost. For simplicity, delayed acknowledgments are not used in the example. The example below applies the Eifel Response Algorithm [LG04] after detecting a spurious timeout.

```
         ...
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         1.          <---------------------------- ACK 5
         2.  SEND 10 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         3.          <---------------------------- ACK 6
         4.  SEND 11 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         5.                       |
                               [delay]
                                  |
             [RTO]
             [F-RTO step (1)]
         6.  SEND 6  ---------------------------->
          (cwnd = 6, ssthresh = 3, FlightSize = 6)
                     <earlier xmitted SEG 6>  --->
         7.          <---------------------------- ACK 7
             [F-RTO step (2b)]
         8.  SEND 12 ---------------------------->
         9.  SEND 13 ---------------------------->
          (cwnd = 7, ssthresh = 3, FlightSize = 7)
                     <earlier xmitted SEG 7>  --->
         10.         <---------------------------- ACK 8
             [F-RTO step (3b)]
             [SpuriousRecovery <- SPUR_TO]
           (cwnd = 7, ssthresh = 6, FlightSize = 6)
         11. SEND 14 ---------------------------->
           (cwnd = 7, ssthresh = 6, FlightSize = 7)
         12.         <---------------------------- ACK 9
         13. SEND 15 ---------------------------->
           (cwnd = 7, ssthresh = 6, FlightSize = 7)
         14.         <---------------------------- ACK 10
         15. SEND 16 ---------------------------->
           (cwnd = 7, ssthresh = 6, FlightSize = 7)
         ...
```

When a sudden delay (long enough to trigger timeout) occurs at step 5, the TCP sender retransmits the first unacknowledged segment (step 6). The next ACK covers the RTO retransmission because the originally transmitted segment 6 arrived at the receiver, and the TCP sender continues by sending two new data segments (steps 8, 9). Note that on F-RTO steps (1) and (2b), congestion window and FlightSize are not yet reset because in the case of spurious timeout, the segments sent before the timeout are still in the network. However, the sender should still be equally aggressive toward conventional TCP. Because the second acknowledgment arriving after the RTO retransmission acknowledges data that was not retransmitted due to timeout (step 10), the TCP sender declares the timeout to be spurious and continues by sending new data on the next acknowledgments. Also, the congestion control state is reversed, as required by the Eifel Response Algorithm.

A.2. Loss of a Retransmission

If a retransmitted segment is lost, the only way to retransmit it is to wait for the timeout to trigger the retransmission. Once the segment is successfully received, the receiver usually acknowledges several segments at once, because other segments in the same window have been successfully delivered before the retransmission arrives at the receiver. The example below shows a scenario where retransmission (of segment 6) is lost, as well as a later segment (segment 9) in the same window. The limited transmit [ABF01] or SACK TCP [MMFR96] enhancements are not in use in this example.

```
         ...
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
             <segment 6 lost>
             <segment 9 lost>
         1.          <---------------------------- ACK 5
         2.  SEND 10 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         3.          <---------------------------- ACK 6
         4.  SEND 11 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         5.          <---------------------------- ACK 6
         6.          <---------------------------- ACK 6
         7.          <---------------------------- ACK 6
         8.  SEND 6  --------------X
          (cwnd = 6, ssthresh = 3, FlightSize = 6)
             <segment 6 lost>
         9.          <---------------------------- ACK 6
         10. SEND 12 ---------------------------->
          (cwnd = 7, ssthresh = 3, FlightSize = 7)
         11.         <---------------------------- ACK 6
         12. SEND 13 ---------------------------->
          (cwnd = 8, ssthresh = 3, FlightSize = 8)
             [RTO]
         13. SEND 6  ---------------------------->
          (cwnd = 8, ssthresh = 2, FlightSize = 8)
         14.         <---------------------------- ACK 9
             [F-RTO step (2b)]
         15. SEND 14 ---------------------------->
         16. SEND 15 ---------------------------->
          (cwnd = 7, ssthresh = 2, FlightSize = 7)
         17.         <---------------------------- ACK 9
             [F-RTO step (3a)]
             [SpuriousRecovery <- FALSE]
          (cwnd = 3, ssthresh = 2, FlightSize = 7)
         18. SEND 9  ---------------------------->
         19. SEND 10 ---------------------------->
         20. SEND 11 ---------------------------->
         ...
```

In the example above, segment 6 is lost and the sender retransmits it after three duplicate ACKs in step 8. However, the retransmission is also lost, and the sender has to wait for the RTO to expire before retransmitting it again. Because the first ACK following the RTO retransmission acknowledges the RTO retransmission (step 14), the sender transmits two new segments. The second ACK in step 17 does not acknowledge any previously unacknowledged data. Therefore, the F-RTO sender enters the slow start and sets cwnd to 3 * MSS. The congestion window can be set to three segments, because two round-trips have elapsed after the retransmission timeout. Finally, the receiver acknowledges all segments transmitted prior to entering recovery and the sender can continue transmitting new data in congestion avoidance.

A.3. Link Outage

The example below illustrates the F-RTO behavior when 4 consecutive packets are lost in the network causing the TCP sender to fall back to RTO recovery. Limited transmit and SACK are not used in this example.

```
         ...
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
             <segments 6-9 lost>
         1.          <---------------------------- ACK 5
         2.  SEND 10 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         3.          <---------------------------- ACK 6
         4.  SEND 11 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         5.          <---------------------------- ACK 6
                                  |
                                  |
             [RTO]
         6.  SEND 6  ---------------------------->
          (cwnd = 6, ssthresh = 3, FlightSize = 6)
         7.          <---------------------------- ACK 7
             [F-RTO step (2b)]
         8.  SEND 12 ---------------------------->
         9.  SEND 13 ---------------------------->
          (cwnd = 7, ssthresh = 3, FlightSize = 7)
         10.         <---------------------------- ACK 7
             [F-RTO step (3a)]
             [SpuriousRecovery <- FALSE]
          (cwnd = 3, ssthresh = 3, FlightSize = 7)
         11. SEND 7  ---------------------------->
         12. SEND 8  ---------------------------->
         13. SEND 9  ---------------------------->
```

Again, F-RTO sender transmits two new segments (steps 8 and 9) after the RTO retransmission is acknowledged. Because the next ACK does not acknowledge any data that was not retransmitted after the retransmission timeout (step 10), the F-RTO sender proceeds with conventional recovery and slow start retransmissions.

A.4. Packet Reordering

Because F-RTO modifies the TCP sender behavior only after a retransmission timeout and it is intended to avoid unnecessary retransmissions only after spurious timeout, we limit the discussion on the effects of packet reordering on F-RTO behavior to the cases where it occurs immediately after the retransmission timeout. When the TCP receiver gets an out-of-order segment, it generates a duplicate ACK. If the TCP sender implements the basic F-RTO algorithm, this may prevent the sender from detecting a spurious timeout.

However, if the TCP sender applies the SACK-enhanced F-RTO, it is possible to detect a spurious timeout when packet reordering occurs. Below, we illustrate the behavior of SACK-enhanced F-RTO when segment 8 arrives before segments 6 and 7, and segments starting from segment 6 are delayed in the network. In this example the TCP sender reduces the congestion window and slow start threshold in response to spurious timeout.

```
         ...
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         1.          <---------------------------- ACK 5
         2.  SEND 10 ---------------------------->
          (cwnd = 6, ssthresh < 6, FlightSize = 6)
         3.          <---------------------------- ACK 6
         4.  SEND 11 ---------------------------->
         5.                       |
                               [delay]
                                  |
             [RTO]
         6.  SEND 6  ---------------------------->
          (cwnd = 6, ssthresh = 3, FlightSize = 6)
                     <earlier xmitted SEG 8>  --->
         7.          <---------------------------- ACK 6
                                                   [SACK 8]
             [SACK F-RTO stays in step 2]
         8.          <earlier xmitted SEG 6>  --->
         9.          <---------------------------- ACK 7
                                                   [SACK 8]
             [SACK F-RTO step (2b)]
         10. SEND 12 ---------------------------->
         11. SEND 13 ---------------------------->
           (cwnd = 7, ssthresh = 3, FlightSize = 7)
         12.         <earlier xmitted SEG 7>  --->
         13.         <---------------------------- ACK 9
             [SACK F-RTO step (3b)]
             [SpuriousRecovery <- SPUR_TO]
           (cwnd = 7, ssthresh = 6, FlightSize = 6)
         14. SEND 14 ---------------------------->
           (cwnd = 7, ssthresh = 6, FlightSize = 7)
         15.         <---------------------------- ACK 10
         16. SEND 15 ---------------------------->
         ...
```

After RTO expires and the sender retransmits segment 6 (step 6), the receiver gets segment 8 and generates duplicate ACK with SACK for segment 8. In response to the acknowledgment, the TCP sender does not send anything but stays in F-RTO step 2. Because the next acknowledgment advances the cumulative ACK point (step 9), the sender can transmit two new segments according to SACK-enhanced F-RTO. The next segment acknowledges new data between 7 and 11 that was not acknowledged earlier (segment 7), so the F-RTO sender declares the timeout spurious.

Appendix B: SACK-enhanced F-RTO and Fast Recovery

We believe that a slightly modified, SACK-enhanced F-RTO algorithm can be used to detect spurious timeouts also when RTO expires while an earlier loss recovery is underway. However, there are issues that need to be considered if F-RTO is applied in this case.

In step 3, the original SACK-based F-RTO algorithm requires that an ACK acknowledges previously unacknowledged non-retransmitted data between SND.UNA and send_high. If RTO expires during earlier (SACK-based) loss recovery, the F-RTO sender must use only acknowledgments for non-retransmitted segments transmitted before the SACK-based loss recovery started. This means that in order to declare timeout spurious, the TCP sender must receive an acknowledgment for non-retransmitted segment between SND.UNA and RecoveryPoint in algorithm step 3. RecoveryPoint is defined in conservative SACK-recovery algorithm [BAFW03], and it is set to indicate the highest segment transmitted so far when SACK-based loss recovery begins. In other words, if the TCP sender receives acknowledgment for a segment that was transmitted more than one RTO ago, it can declare the timeout spurious. Defining an efficient algorithm for checking these conditions remains a future work item.

When spurious timeout is detected according to the rules given above, it may be possible that the response algorithm needs to consider this case separately, for example, in terms of which segments to retransmit after RTO expires, and whether it is safe to revert the congestion control parameters. This is considered a topic for future research.

Appendix C: Discussion of Window-Limited Cases

When the advertised window limits the transmission of two new previously unsent segments, or there are no new data to send, it is recommended in F-RTO algorithm step (2b) that the TCP sender continue with the conventional RTO recovery algorithm. The disadvantage is that the sender may continue unnecessary retransmissions due to possible spurious timeout. This section briefly discusses the options that can potentially improve performance when transmitting previously unsent data is not possible.

- The TCP sender could reserve an unused space of a size of one or two segments in the advertised window to ensure the use of algorithms such as F-RTO or Limited Transmit [ABF01] in window-limited situations. On the other hand, while doing this, the TCP sender should ensure that the window of outstanding segments is large enough for proper utilization of the available pipe.

- Use additional information if available, e.g., TCP timestamps with the Eifel Detection algorithm, for detecting a spurious timeout. However, Eifel detection may yield different results from F-RTO when ACK losses and an RTO occur within the same round-trip time [SKR03].

- Retransmit data from the tail of the retransmission queue and continue with step 3 of the F-RTO algorithm. It is possible that the retransmission will be made unnecessarily. Thus, this option is not encouraged, except for hosts that are known to operate in an environment that is prone to spurious timeouts. On the other hand, with this method it is possible to limit unnecessary retransmissions due to spurious timeout to one retransmission.

- Send a zero-sized segment below SND.UNA, similar to TCP Keep-Alive probe, and continue with step 3 of the F-RTO algorithm. Because the receiver replies with a duplicate ACK, the sender is able to detect whether the timeout was spurious from the incoming acknowledgment. This method does not send data unnecessarily, but it delays the recovery by one round-trip time in cases where the timeout was not spurious. Therefore, this method is not encouraged.

- In receiver-limited cases, send one octet of new data, regardless of the advertised window limit, and continue with step 3 of the F-RTO algorithm. It is possible that the receiver will have free buffer space to receive the data by the time the segment has propagated through the network, in which case no harm is done. If the receiver is not capable of receiving the segment, it rejects the segment and sends a duplicate ACK.

Authors' Addresses

Pasi Sarolahti
Nokia Research Center
P.O. Box 407
FIN-00045 NOKIA GROUP
Finland

Phone: +358 50 4876607
EMail: pasi.sarolahti@nokia.com
http://www.cs.helsinki.fi/u/sarolaht/

Markku Kojo
University of Helsinki
Department of Computer Science
P.O. Box 68
FIN-00014 UNIVERSITY OF HELSINKI
Finland

Phone: +358 9 191 51305
EMail: kojo@cs.helsinki.fi

Full Copyright Statement

Intellectual Property

Acknowledgement

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