Internet Engineering Task Force
INTERNET DRAFT
File: draft-sarolahti-tsvwg-tcp-frto-04.txt

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

June, 2003
Expires: December, 2003

F-RTO: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP

Status of this Memo

Abstract

Spurious retransmission timeouts (RTOs) cause suboptimal TCP performance, because they often result in unnecessary retransmission of the last window of data. This document describes the "Forward RTO Recovery" (F-RTO) algorithm for detecting spurious TCP RTOs. 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 an RTO, the F-RTO algorithm at a TCP sender monitors the incoming acknowledgements to determine whether the timeout was spurious and to decide whether to send new segments or retransmit unacknowledged segments. The algorithm effectively helps to avoid additional unnecessary retransmissions and thereby improves TCP performance in case of a spurious timeout. The F-RTO algorithm can

also be applied with the SCTP protocol.

1. Introduction

The TCP protocol [Pos81] has two methods for triggering retransmissions. Primarily, the TCP sender relies on incoming duplicate ACKs, which indicate that the receiver is missing some of the data. After a required amount of successive duplicate ACKs have arrived at the sender, it retransmits the first unacknowledged segment [APS99]. Secondarily, the TCP sender maintains a retransmission timer which triggers retransmission of segments, if they have not been acknowledged within the retransmission timer expiration period. When the retransmission timer expires, the TCP sender enters the RTO recovery where 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 trigger unnecessary retransmissions when no segments have been lost [GL02]. After a spurious RTO the late acknowledgements of original segments arrive at the sender, usually triggering unnecessary retransmissions of whole window of segments during the RTO recovery. Furthermore, after a spurious RTO a conventional TCP sender increases the congestion window on each late acknowledgement in slow start, injecting a large number of data segments to the network within one round-trip time.

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

This document describes an alternative RTO recovery algorithm called "Forward RTO-Recovery" (F-RTO) to be used for detecting spurious RTOs

and thus avoiding unnecessary retransmissions following the RTO. When the RTO is not spurious, the F-RTO algorithm reverts back to the conventional RTO recovery algorithm and should have similar behavior and performance. F-RTO does not require any TCP options in its operation, and it can be implemented by modifying only the TCP

sender. This is different from alternative algorithms (Eifel [LK00], [LM03] and DSACK-based algorithms [BA02]) that have been suggested for detecting unnecessary retransmissions. The Eifel algorithm uses TCP timestamps for detecting a spurious timeout and the DSACK-based algorithms require that the TCP Selective Acknowledgment Option [MMFR96] with DSACK extension [FMMP00] is in use. With DSACK, the TCP receiver can report if it has received a duplicate segment, making it possible for the sender to detect afterwards whether it has retransmitted segments unnecessarily.

When an RTO occurs, the F-RTO sender retransmits the first unacknowledged segment normally, but tries to transmit new, previously unsent data after that. If the next two acknowledgements cover segments that were not retransmitted, the F-RTO sender can declare the RTO spurious and exit the RTO recovery. However, if either of the next two acknowledgements is a duplicate ACK, there was no sufficient evidence of spurious RTO; 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 RTOs may be detected even if duplicate ACKs arrive after an RTO. The F-RTO algorithm only attempts to detect and avoid unnecessary retransmissions after an RTO. Eifel and DSACK can also be used in detecting unnecessary retransmissions in other events, for example due to packet reordering.

The F-RTO algorithm can also be applied with the SCTP protocol [Ste00], because SCTP has similar acknowledgement and packet retransmission concepts as TCP. When a SCTP retransmission timeout occurs, the SCTP sender is required to retransmit the outstanding data similarly to TCP, thus being prone to unnecessary retransmissions and congestion control adjustments, if delay spikes occur in the network. The SACK-enhanced version of F-RTO should be directly applicable to SCTP, which has selective acknowledgements as a built-in feature. For simplicity, this document mostly refers to TCP, but the algorithms and other discussion should be applicable also to SCTP.

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 leverage on the TCP SACK option. Section 4 discusses the possible actions to be taken after detecting a spurious RTO, and Section 5 discusses the security considerations.

2. F-RTO Algorithm

The F-RTO algorithm affects the TCP sender behavior only after a retransmission timeout. Otherwise the TCP behavior remains

unmodified. This section describes a basic version of the F-RTO algorithm that does not require TCP options to work. The actions taken in response to spurious RTO are not specified in this document, but we discuss the different alternatives for congestion control in Section 4.

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 could then be used as an input for a related response algorithm. With F-RTO, the outcome of SpuriousRecovery can either be SPUR_TO, indicating a spurious retransmission timeout; or FALSE, when the RTO is not declared spurious, and the TCP sender should follow the conventional RTO recovery algorithm.

When the retransmission timer expires, the F-RTO algorithm takes the following steps at the TCP sender.

1) When RTO expires, the TCP sender SHOULD retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE.

   The highest sequence number transmitted so far is stored in variable "send_high". The TCP sender MAY postpone adjusting the congestion control parameters for the next two incoming ACKs, until it has got more input on whether the RTO was spurious or not. If the TCP sender adjusts the congestion control parameters at this point, it may store the earlier values of the parameters to be able to restore the values in case it detects that the RTO was spurious.

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

   a) If the acknowledgement is a duplicate ACK OR it is acknowledging a sequence number equal to (or above) the value of send_high, the TCP sender MUST revert to the conventional RTO recovery, and continue by transmitting unacknowledged data in slow start. The TCP sender does not enter step 3 of this algorithm, and the SpuriousRecovery variable remains as FALSE.

      If the acknowledgement covers the send_high point, there is not enough evidence that a non-retransmitted segment has arrived at the receiver after the RTO. This is a common case when a fast retransmission is lost and it has been retransmitted again after an RTO.

      If the acknowledgement is a duplicate ACK, it was triggered by a segment that was sent before the RTO retransmission. This can occur, for example, if the TCP sender is under fast recovery when the RTO occurs. In this situation reliably declaring a spurious RTO is difficult, hence the safest alternative is to follow the conventional RTO recovery.

   b) If the acknowledgement advances the window AND it is below the value of send_high, the TCP sender transmits up to two new (previously unsent) segments and enters step 3 of this algorithm.

      Sending two new segments at this point is equally aggressive to the conventional RTO recovery algorithm, which would increase cwnd to 2 * MSS when the first valid ACK arrives after RTO. It is possible that the sender can transmit only one new segment at this time, because the receiver window limits it, or because the TCP sender does not have more data to send. This does not prevent the algorithm from working. In any case, the TCP sender SHOULD transmit at least one segment, either new data or from the retransmission queue. If the sender retransmits the next unacknowledged segment, it MUST NOT enter the step 3 of this

algorithm, but continue retransmitting similarly to the conventional RTO recovery algorithm.

If the first acknowledgement after RTO does not acknowledge all of the data that was retransmitted in step 1, the TCP sender MUST NOT enter step 3 of this algorithm. Otherwise, a malicious receiver acknowledging partial segments could cause the sender to declare the RTO spurious in a case where data was lost. When receiving an acknowledgement for a partial segment, the TCP sender SHOULD revert to the conventional RTO recovery.

3) When the second acknowledgement after the RTO arrives at the sender, either declare the RTO spurious, or start retransmitting the unacknowledged segments.

   a) If the acknowledgement is a duplicate ACK, the TCP sender MUST set congestion window to no more than 3 * MSS, and continue with the slow start algorithm retransmitting unacknowledged segments. The sender leaves SpuriousRecovery to FALSE.

      The duplicate ACK indicates that at least one segment other than the segment which triggered RTO is lost in the last window of data. There is no sufficient evidence to assume that the RTO was spurious. Therefore, the sender proceeds with retransmissions similarly to the conventional RTO recovery algorithm, with the send_high variable stored when the

retransmission timer expired to avoid unnecessary fast retransmits.

   b) If the acknowledgement advances the window and acknowledges data beyond the highest sequence number that was retransmitted on RTO, the TCP sender SHOULD declare the RTO spurious and set SpuriousRecovery to SPUR_TO.

      Because the TCP sender has retransmitted only one segment after the RTO, this acknowledgement indicates that an originally transmitted segment has arrived at the receiver. This is regarded as a strong indication of a spurious RTO. The TCP sender should assume that the unacknowledged segments are not lost and continue by sending new previously unsent segments.

If this algorithm branch is taken, the TCP sender SHOULD set the value of send_high variable to SND.UNA in order to disable the NewReno "bugfix" [FH99]. The send_high variable was proposed for avoiding unnecessary multiple fast retransmits when RTO expires during fast recovery with NewReno TCP. As the sender has not retransmitted other segments but the one that triggered RTO, the problem addressed by the bugfix cannot occur. Therefore, if there are duplicate ACKs arriving at the sender after the RTO, they are likely to indicate a packet loss, hence 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 another time and the sender enters step 1 of this algorithm.

If the TCP sender does not have any new data to send in algorithm branch (2b), or the receiver window limits the transmission, it has to send something in order to prevent the transmission from stalling. In that case the following options are available for the sender.

- Continue with the conventional RTO recovery algorithm and do not try to detect the spurious RTO. The disadvantage is that the sender may do unnecessary retransmissions due to possible spurious RTO. On the other hand, we believe that the benefits of detecting spurious RTO in an application limited or receiver limited situations are not very remarkable.

- Use additional information if available, e.g. TCP timestamps with the Eifel Detection algorithm, for detecting a spurious RTO. However, Eifel detection may yield different results from F-RTO when ACK losses and a RTO occur within the same round-trip time [SKR02].

- 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 is unnecessarily made, hence this option is not encouraged, except for hosts that are known to operate in an environment that is highly likely to have spurious RTOs. On the other hand, with this method it is possible to avoid several unnecessary retransmissions due to spurious RTO by doing only one retransmission that may be unnecessary.

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

After the RTO is declared spurious, the TCP sender cannot detect if the unnecessary RTO retransmission was lost. In principle the loss of the RTO retransmission should be taken as a congestion signal, and thus there is a small possibility that the F-RTO sender violates the congestion control rules, if it chooses to fully revert congestion control parameters after detecting a spurious RTO. 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 avoids most of the unnecessary retransmissions after a spurious RTO, 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 retransmission ambiguity. As a result, the RTO estimator is likely have more accurate and larger values with F-RTO than with the regular TCP after a spurious RTO that was triggered due to delayed segments. We believe this is an advantage in the networks that are prone to delay spikes.

It is possible that the F-RTO algorithm does not always avoid unnecessary retransmissions after spurious RTO. If packet reordering

or packet duplication occurs on the segment that triggered the

spurious RTO, the F-RTO algorithm may not detect the spurious RTO. Additionally, if a spurious RTO occurs during fast recovery, the F-RTO algorithm often cannot detect the spurious RTO. However, we consider these cases relatively rare, and note that in cases where F-RTO fails to detect the spurious RTO, it performs similarly to the regular RTO recovery.

3. A SACK-enhanced version of the F-RTO algorithm

This section describes an alternative version of the F-RTO algorithm, that makes use of TCP Selective Acknowledgement Option [MMFR96]. By using the SACK option the TCP sender can detect spurious RTOs in most of the cases when packet reordering or packet duplication is present, or when the TCP sender is under loss recovery. The difference to the basic F-RTO algorithm is that the sender may declare RTO spurious even when duplicate ACKs follow the RTO, if the SACK blocks acknowledge new data that was not transmitted after RTO. The algorithm presented in this section is also applicable to be used with the SCTP protocol.

The SACK-enhanced F-RTO algorithm takes the following steps:

1) When RTO expires, the TCP sender SHOULD retransmit first unacknowledged segment and set SpuriousRecovery to FALSE.

   A variable "SpuriousThreshold" is set to indicate the highest acknowledgement that is accepted for declaring the RTO spurious. If there was loss recovery ongoing at the time an RTO occurs, SpuriousThreshold is set to the value of RecoveryPoint that was valid at the time when RTO occurred. Otherwise, SpuriousThreshold is set to HighData, that indicates the highest sequence number transmitted. The SACK-based loss recovery algorithm describes RecoveryPoint and HighData in more detail [BAFW03].

2) The first acknowledgement after RTO arrives at the sender.

   a) if the cumulative ACK acknowledges all segments up to SpuriousThreshold stored in algorithm step 1, the TCP sender SHOULD revert to the conventional RTO recovery and it MUST set congestion window to no more than 2 * MSS. The sender does not enter step 3 of this algorithm.

   b) otherwise, the TCP sender transmits up to two new (previously unsent) segments, within the limitations of the congestion window. If the TCP sender is not able to transmit previously unsent data due to receiver window limitation or because it

does not have new data to send, it may follow one of the options presented in Section 2.

However, if the TCP sender chooses to retransmit a data segment here, SACK of that segment cannot be used for declaring a spurious RTO in step (3b).

3) The second acknowledgement after RTO arrives at the sender.

a) if the ACK acknowledges data above SpuriousThreshold, either in SACK blocks or as a cumulative ACK, the sender MUST set congestion window to no more than 3 * MSS and proceed with slow start, retransmitting unacknowledged segments. The sender SHOULD take this branch also when the acknowledgement is a duplicate ACK and it does not contain any new SACK blocks for previously unacknowledged data below SpuriousThreshold.

b) if the ACK does not acknowledge data above SpuriousThreshold AND it acknowledges some previously unacknowledged data below SpuriousThreshold, the TCP sender SHOULD declare the RTO spurious and set SpuriousRecovery to SPUR_TO.

If there are unacknowledged holes between the received SACK blocks, those segments SHOULD be retransmitted similarly to the conventional SACK recovery algorithm. In addition, RecoveryPoint should be set to its earlier value, since no loss recovery was needed due to the RTO.

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 RTO is detected to be spurious, the TCP sender should not start retransmitting based on the RTO. For example, if the sender was in congestion avoidance phase transmitting new previously unsent segments, it should continue transmitting previously unsent segments after detecting spurious RTO. In addition, it is suggested that the RTO estimation is reinitialized and the RTO timer is adjusted to a more conservative value in order to avoid subsequent spurious RTOs [LG02].

Different approaches have been suggested for adjusting the congestion control state after a spurious RTO. This document does not specifically recommend any of the alternatives below, but considers the response to spurious RTO as a subject of further research.

1) Revert the congestion control parameters to the state before the

RTO [LG02]. This appears to be a justified decision, because it is similar to the situation in which the RTO did not expire spuriously. However, two concerns exists with this approach: First, some detection mechanisms, such as F-RTO or the Eifel Detection algorithm, do not notice the loss of the spurious retransmission, thus introducing a small risk of violation of the congestion control principles. Second, a spurious RTO indicates that some part of the network was unable to deliver packets for a while, which can be considered as a potential indication of congestion.

2) Reduce congestion window to half of its earlier value but revert slow start threshold to its earlier value [SL03]. This alternative takes measures to validate the congestion window after a period during which no data has been transmitted. This would be a justified action to take if the spurious RTO is assumed to be caused due to changes in the network conditions, such as a change in the available bandwidth or a wireless handoff to another point of attachment in the network.

3) Reduce ssthresh and congestion window when detecting a spurious RTO [SKR02]. For example, ssthresh and cwnd could be set to half of their earlier values, as done with the other congestion notification events. This alternative would be conservative enough considering the possibility of not detecting a packet loss of the RTO-triggered retransmission, but the TCP sender should avoid reducing the congestion window more than once in a round-trip time. Furthermore, if a spurious RTO occurs in the beginning of a TCP connection, this alternative causes the slow start to be canceled, which may sacrifice TCP performance.

5. Security Considerations

The main security threat regarding F-RTO is the possibility of receiver misleading the sender to set 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

acknowledgement 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 lead to declare RTO spurious in F-RTO algorithm step 3. However, since this causes the sender to have 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 of an RTO 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 RTO 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 RTO was declared spurious, but reduce the congestion window to 1. However, the sender can take actions to avoid unnecessary retransmissions normally. If a TCP sender implements a burst avoidance algorithm that limits the sending rate to be no higher than in slow start, this precaution is not needed, and the sender may apply F-RTO normally.

If there are more than one segments missing at the time when an RTO occurs, the receiver does not benefit from misleading the sender to declare a spurious RTO, because the sender would then have to go through another recovery period to retransmit the missing segments, usually after an RTO.

Acknowledgements

We are grateful to Reiner Ludwig, Andrei Gurtov, Josh Blanton, Mark Allman, Sally Floyd, Yogesh Swami, Mika Liljeberg, and Ivan Arias Rodriguez for the discussion and feedback contributed to this text.

Normative References

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

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

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

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

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

[Ste00] R. Stewart, et. al. Stream Control Transmission Protocol. RFC 2960, October 2000.

Informative References

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

[BA02] E. Blanton and M. Allman. On Making TCP more Robust to Packet Reordering. ACM Computer Communication Review, 32(1), January 2002.

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

[FH99] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option to 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, Flo-

rence, Italy, February 2002

[LG02] R. Ludwig and A. Gurtov. The Eifel Response Algorithm for TCP. Internet draft "draft-ietf-tsvwg-tcp-eifel-response-02.txt". December 2002. Work in progress.

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

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

[SKR02] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts. University of Helsinki, Dept. of Computer Science. Series of Publications C, No. C-2002-07. February 2002. Available at: http://www.cs.helsinki.fi/research/iwtcp/papers/f-rto.ps

[SL03] Y. Swami and K. Le. DCLOR: De-correlated Loss Recovery using SACK option for spurious timeouts. Internet draft "draft-swami-tsvwg-tcp-dclor-01.txt". April 2003. Work in progress.

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 RTO, 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 [SKR02].

A.1. Sudden delay

An unexpectedly long delay can trigger an RTO, should it occur on a single packet blocking the following packets, or appear as increased RTTs for several successive packets. The example below illustrates the sequence of packets and acknowledgements seen by the TCP sender that follows the F-RTO algorithm, when a sudden delay occurs triggering RTO but no packets are lost. For simplicity, delayed

acknowledgements are not used in the example.

```
...                 (cwnd = 6, ssthresh < 6, FlightSize = 5)
1.  SEND(10)
2.  ACK(6)
3.  SEND(11)
4.  <delay + RTO>   (set ssthresh <- 3)
5.  SEND(6)
6.  ACK(7)
7.  SEND(12)
8.  SEND(13)
9.  ACK(8)          (set cwnd <- 3, FlightSize = 6)
10. ACK(9)          (cwnd = 3,  FlightSize = 5)
11. ACK(10)         (cwnd = 3,  FlightSize = 4)
12. ACK(11)         (cwnd = 4,  FlightSize = 3)
13. SEND(14)
...
```

When a sudden delay long enough to trigger RTO occurs at step 4, the TCP sender retransmits the first unacknowledged segment (step 5). Because the next ACK advances the cumulative ACK point, the TCP sender continues by sending two new data segments (steps 7, 8) and adjusts cwnd to 3 MSS. Because the second acknowledgement arriving after the RTO also advances the cumulative ACK point, the TCP sender exits the recovery and continues with the congestion avoidance. From this point on the retransmissions are invoked either by fast retransmit or when triggered by the retransmission timer. Because the TCP sender reduces cwnd when receiving the first ACK after RTO and sends the two new data segments at steps 7 and 8, it has to wait until the FlightSize is reduced to the level of congestion window

before it can continue transmitting again at step 13.

A.2. Loss of a retransmission

If a retransmitted segment is lost, the only way to retransmit it again is to wait for the RTO to trigger the retransmission. Once the segment is successfully received, the receiver usually acknowledges several segments cumulatively. 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 = 5)
    <segment 6 lost>
1.  SEND(10)
2.  ACK(6)
3.  SEND(11)
4.  ACK(6)
5.  ACK(6)
6.  ACK(6)
7.  SEND(6)         (set cwnd <- 6, set ssthresh <- 3)
    <segment 6 lost>
8.  ACK(6)
9.  <RTO>           (set ssthresh <- 2)
10. SEND(6)
11. ACK(9)
12. SEND(12)
13. SEND(13)
14. ACK(9)          (set cwnd <- 3)
15. SEND(9)
16. SEND(10)
17. SEND(11)
18. ACK(11)
...
```

In the example above, segment 6 is lost and the sender retransmits it after three duplicate ACKs in step 7. 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 advances the cumulative ACK point (step 11), the sender transmits two new segments. The second ACK in step 14 does not advance the cumulative ACK point, and the sender enters the slow start, sets cwnd to 3 * MSS, and retransmits the next three unacknowledged segments, as per the F-RTO algorithm description given in Section 2. After this 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

A performance study shows that F-RTO performs similarly to the regular recovery when consecutive packets are lost both up- and

downstream as a result of link outage, triggering an RTO [SKR02]. If the RTO was not spurious but some data was actually lost, one of the next two ACKs after RTO does not advance the cumulative ACK point when RTO was caused by data loss, because the basic F-RTO retransmits only one segment after RTO. As a result, F-RTO sender continues by retransmitting unacknowledged segments similarly to the conventional RTO recovery.

A.4. Packet reordering

Since F-RTO modifies the TCP sender behavior only after a retransmission timeout and it is intended to avoid unnecessary retransmits only after spurious RTO, we limit the discussion on the effects of packet reordering in F-RTO behavior to the cases where packet reordering occurs immediately after RTO. We consider the retransmission timeout due to packet reordering to be very rare case, since reordering often triggers fast retransmit due to duplicate ACKs caused by out-of-order segments. Should packet reordering occur after an RTO, duplicate ACKs arrive to the sender, taking the F-RTO algorithm to retransmit in slow start as a regular RTO recovery would do. Although this might not be the correct action, it is similar to the behavior of the regular TCP, making F-RTO a safe modification also in the presence of reordering.

Appendix B: On using the TCP timestamps with F-RTO

The basic F-RTO algorithm suggests applying the conventional RTO recovery if the receiver window or application limits the transmission of new previously unsent data, and in such a case it is possible that the F-RTO algorithm cannot be used to detect a spurious RTO. The F-RTO sender can avoid the need of transmitting new previously unsent segments after RTO, if it has TCP timestamps [BBJ92] available. The Eifel detection algorithm [LK00] describes how the TCP timestamps can be used to avoid unnecessary retransmissions after a spurious RTO. However, if the RTO is declared spurious based on the timestamp echoed with the first acceptable ACK following the RTO, the TCP sender may falsely declare the RTO spurious and continue by transmitting new data when the RTO was caused by loss of acknowledgements. The Eifel algorithm may signal spurious RTO falsely, if the first data segment retransmitted after RTO was not lost, but the corresponding acknowledgement was, and the acknowledgement does not include DSACK option [FMMP00]. If sender and receiver implement DSACK, this problem can be avoided.

An alternative algorithm for detecting spurious RTOs by using TCP timestamps without DSACK is described below. When TCP timestamps are available, the F-RTO sender MAY apply the following algorithm.

1) When RTO expires, retransmit first unacknowledged segment and store the timestamp of retransmitted segment in variable "RetransmitTS". Store the highest sequence number transmitted so far in variable "send_high".

2) Wait until the first ACK that acknowledges previously unacknowledged data arrives at the sender. If duplicate ACKs arrive, they are processed normally while the sender stays in this step of the algorithm.

   a) if the timestamp echoed with the ACK is later or equal than what is stored in "RetransmitTS", the TCP sender SHOULD revert to the conventional RTO recovery and it MUST NOT enter step 3 of this algorithm. The sender should adjust the congestion window according to the standard congestion control rules.

   b) if the timestamp echoed with the first ACK is earlier than what is stored in "RetransmitTS", the TCP sender SHOULD transmit the first unacknowledged segment and enter step 3 of this algorithm.

3) When the next acknowledgement arrives at the sender, it SHOULD apply one of the following branches of the algorithm.

   a) if the timestamp echoed with the ACK is later or equal than what is stored in "RetransmitTS", or if the acknowledgement is duplicate ACK, the TCP sender SHOULD revert to the conventional RTO recovery. The TCP sender MUST set the congestion window to no more than 2 * MSS.

   b) if the timestamp echoed with the ACK is earlier than what is stored in "RetransmitTS", the TCP sender SHOULD declare the RTO spurious. send_high SHOULD be set to the value of SND.UNA to cancel the NewReno bugfix, as described in Section 2.

The drawback of this algorithm compared to the original Eifel detection is that the above-presented algorithm can make two unnecessary retransmissions instead of one. In addition, packet reordering, packet duplication, or packet loss for the next segment after the one that triggered RTO may prevent the detection of spurious RTO. Therefore, it may be desirable to apply the basic F-RTO or the SACK-enhanced version of the F-RTO algorithm whenever the sender is able to transmit previously unsent data when the first ACK

after RTO arrives. However, we believe the algorithm above

effectively avoids false spurious RTO signals.

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 26
FIN-00014 UNIVERSITY OF HELSINKI
Finland

Phone: +358 9 1914 4179
EMail: markku.kojo@cs.helsinki.fi