Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-tcpm-rfc4138bis-01.ps
Expires: May 2008

P. Sarolahti
Nokia Research Center
M. Kojo
University of Helsinki
K. Yamamoto
M. Hata
NTT Docomo

18 November 2007

# Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP

## Status of this Memo



## 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.

## 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, a hand-off may take place from a low latency path to a high latency path, suddenly increasing the round-trip time beyond the current

RTO value. Third, on a low-bandwidth link the arrival of competing traffic (possibly with higher priority), or some other change in available bandwidth, can cause a sudden increase of the 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. Even though this document only specifies F-RTO algorithm for TCP, the algorithm can also be applied to the Stream Control Transmission Protocol (SCTP) [Ste00] that has acknowledgment and packet retransmission concepts similar to TCP. Considerations on applying F-RTO for SCTP are discussed in RFC 4138 [SK05].

This document is organized as follows. Section 2 describes the basic F-RTO algorithm, and the SACK-enhanced F-RTO algorithm is given in Section 3. Section 4 discusses the possible actions to be taken after detecting a spurious RTO and Section 5 discusses the security considerations.

### 1.1. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC2119] and indicate requirement levels for protocols.

## 2. Basic 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 implementing the basic F-RTO algorithm MUST take the following steps after the retransmission timer expires. If the retransmission timer expires again during the execution of the F-RTO algorithm, the TCP sender MUST re-start the algorithm processing from step 1. 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.

The F-RTO algorithm takes different actions based on whether an incoming acknowledgement advances the cumulative acknowledgement point for an received in-order segment, or whether it is a duplicate acknowledgement to indicate an out-of-order segment. Duplicate acknowledgement is defined in [APB07]. The F-RTO algorithm does not specify actions for receiving a segment that does not acknowledge new data but is not a duplicate acknowledgement. The TCP sender SHOULD ignore such segments and wait for a segment that either acknowledges new data or is a duplicate acknowledgment.

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 TCP sender, the TCP 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 the Acknowledgement field covers "recover" but not more than "recover" OR the acknowledgment 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 the Acknowledgement field does not cover "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 A.

3) When the second acknowledgment after the RTO retransmission arrives at the TCP 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 A 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 F-RTO 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 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.

The steps of the SACK-enhanced version of the F-RTO algorithm are as follows. If the retransmission timer expires again during the execution of the SACK-enhanced F-RTO algorithm, the TCP sender MUST re-start the algorithm processing from step 1.

1) When the RTO expires, retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Set variable "RecoveryPoint" 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 the Cumulative Acknowledgement field covers "RecoveryPoint" but not more than "RecoveryPoint", 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 the Cumulative Acknowledgement field does not cover "RecoveryPoint" but is 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 A.

3) The next acknowledgment arrives at the sender. Either a duplicate ACK or a new cumulative ACK (advancing the window) applies in this step. Other types of ACKs are ignored without any action.

a) if the Cumulative Acknowledgement field or a SACK block covers more than "RecoveryPoint", 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 "RecoveryPoint" in the SACK blocks. Leave SpuriousRecovery set to FALSE.

b) if the Cumulative Acknowledgement field or a SACK block in the ACK does not cover more than "RecoveryPoint" 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, "RecoveryPoint" is set to SND.UNA, thus allowing fast recovery on incoming duplicate acknowledgments.

The SACK enhanced algorithm works on the same principle as the basic algorithm, but by utilizing the additional information from the SACK option. When a genuine retransmission timeout occurs during a steady state of a connection, it can be assumed that there are no segments left in the pipe. Otherwise, the acknowledgments triggered by these segments would have triggered the SACK loss recovery or transmission of new segments. Therefore, if the F-RTO sender receives acknowledgements for segments transmitted before the retransmission timeout in response to the two new segments sent at the algorithm step 2, the normal operation of TCP has been just delayed, and the retransmission timeout is considered spurious. Note that this reasoning works only when the TCP sender is not in SACK loss recovery at the time the retransmission timeout occurs.

## 4. Taking Actions after Detecting Spurious RTO

Upon a 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 in congestion avoidance.

There are currently two alternatives specified for a spurious timeout response algorithm, the Eifel Response Algorithm [LG04], and an algorithm for adapting the retransmission timeout after a spurious RTO [BBA06]. If no specific response algorithm is implemented, the TCP SHOULD respond to spurious timeout conservatively, applying the TCP congestion control specification [APS99]. Different response algorithms for spurious retransmission timeouts have been analyzed in some research papers [GL03, Sar03] and IETF documents [SL03].

## 5. Evaluation of RFC 4138 and Differences to this Document

F-RTO was first specified in an Experimental RFC 4138 that has been implemented in a number of operating systems since it was published. Gained experience has been documented in a separate document [KYHS07], and can be summarized as follows.

If the TCP sender employs F-RTO, it is able to detect spurious RTOs and avoid the unnecessary retransmission of the whole window of data. Because F-RTO avoids the unnecessary retransmissions after a spurious RTO, it is able to adhere to the packet conservation principle, unlike a regular TCP that enters the slow-start recovery unnecessarily an inappropriately restarts the ACK clock while there are segments outstanding in the network. When a spurious RTO has been detected, a sender can select an appropriate congestion control response instead of setting the congestion window to one segment. Because F-RTO avoids unnecessary retransmissions, it is able to take the RTT of the delayed segments into account when calculating the RTO estimate, which may help in avoiding further spurious retransmission timeouts.

Experimental results with the basic F-RTO have been reported in an emulated network using a Linux implementation [SKR03]. Also different congestion control responses along with the SACK-enhanced version of F-RTO were tested in a similar environment [Sar03]. There are publications analyzing F-RTO performance over commercial W-CDMA networks, and in an emulated HSDPA network [Yam05, Hok05]. Also Microsoft reported positive experiences with their implementation of F-RTO in the IETF-68 meeting.

It is known that some spurious RTOs may remain undetected by F-RTO if duplicate acknowledgements arrive at the sender immediately after the spurious RTO, for example due to packet reordering or packet loss. There are rare corner cases where F-RTO could "hide" a packet loss and therefore lead to inappropriate behavior with non-conservative congestion control response: first, if a massive packet reordering occurred so that the acknowledgement of RTO retransmission arrived at the sender before the acknowledgments of original transmissions, the sender might not detect the loss of the segment that triggered the RTO. Second, a malicious receiver could lead F-RTO to make a wrong conclusion after an RTO by acknowledging segments it has not received. Such receiver would, however, risk breaking the consistency of the TCP state between the sender and receiver, causing the connection to become unusable, which cannot be of any benefit to the receiver. Therefore we believe it is not likely that receivers would start employing such tricks in a significant scale. Finally, loss of the unnecessary RTO retransmission cannot be detected without using some explicit acknowledgement scheme such as DSACK. This is common to the other mechanisms for detecting spurious RTO, as well as to regular TCP that does not use DSACK. We note that if the congestion control response to spurious RTO is conservative enough, the above corner cases do not cause problems due to increased congestion.

## 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 the 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

The authors would like to thank Alfred Hoenes and Ilpo Jarvinen for the comments on this document.

We are also thankful 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 who gave valuable feedback during the preparation of RFC 4138, the precursor of this document.

# Appendix

## A. 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 receiver 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. Furthermore, the operation of the SACK-based F-RTO algorithm would need to consider this case separately, to not use the retransmitted segment to indicate spurious timeout. Given these considerations, this option is not recommended.

- 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.

## B. List of Changes

Changes between different document versions are summarized below, apart from minor editing and language improvements.

Changes from draft-ietf-tcpm-rfc4138bis-00:

* Added back the original SACK-algorithm from RFC 4138 after the common feedback to have the SACK-algorithm in the document. Clarified the algorithm a bit, and added one paragraph of description of the basic idea of the algorithm.

* Clarified behavior on multiple timeouts.

* Added a paragraph on acknowledgements that do not acknowledge new data but are not duplicate acknowledgements

Changes from RFC 4138:

* Removed description of the SACK-enhanced algorithm

* Removed SCTP considerations

* Removed earlier Appendix sections, except Appendix C from RFC 4138, which is now Appendix A

* Clarified text about the possible response algorithms

* Added section that summarizes the evaluation of RFC 4138

## References

## Normative References

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

[APB07] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", Internet-Draft "draft-ietf-tcpm-rfc2581bis-03.txt", September 2007.

[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.

## 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.

[BBA06] J. Blanton, E. Blanton, and M. Allman. Using Spurious Retransmissions to Adapt the Retransmission Timeout, Internet-Draft "draft-allman-rto-backoff-04.txt", December 2006. Work in progress.

[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.

[Hok05] A. Hokamura, et al. "Performance Evaluation of F-RTO and Eifel Response Algorithms over W-CDMA packet network". Wireless Personal Multimedia Communications (WPMC'05), Sept. 2005.

[KYHS07] M. Kojo, K. Yamamoto, M. Hata, and P. Sarolahti. Evaluation of RFC 4138. Internet-draft "draft-kojo-tcpm-frto-eval-00.txt", June 2007. Work in progress.

[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.

[SK05] P. Sarolahti and M. Kojo, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP), RFC 4138, August 2005.

[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", Expired Internet-Draft, September 2003.

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

[Yam05] K. Yamamoto, et al. "Effects of F-RTO and Eifel Response Algorithms for W-CDMA and HSDPA networks". Wireless Personal Multimedia Communications (WPMC'05), Sept. 2005.

## AUTHORS' ADDRESSES

Pasi Sarolahti
Nokia Research Center
P.O. Box 407
FI-00045 NOKIA GROUP
Finland
Phone: +358 50 4876607
Email: pasi.sarolahti@nokia.com

Markku Kojo
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland
Email: kojo@cs.helsinki.fi

Kazunori Yamamoto
NTT Docomo, Inc.
3-5 Hikarinooka, Yokosuka, Kanagawa, 239-8536, Japan
Phone: +81-46-840-3812
Email: yamamotokaz@nttdocomo.co.jp

Max Hata
NTT Docomo, Inc.
3-5 Hikarinooka, Yokosuka, Kanagawa, 239-8536, Japan
Phone: +81-46-840-3812
Email: hatama@s1.nttdocomo.co.jp