

Internet Engineering Task Force  
INTERNET DRAFT  
File: [draft-ietf-tcpm-frto-00.txt](#)

P. Sarolahti  
Nokia Research Center  
M. Kojo  
University of Helsinki  
May, 2004  
Expires: November, 2004

## **F-RT0: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP**

### Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of [Section 10 of \[RFC2026\]](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

### Copyright Notice

Copyright (C) The Internet Society (2004). All Rights Reserved.

### 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-RT0 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 at a TCP sender monitors the incoming acknowledgments 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 to SCTP.

## 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](#)].

## **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 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 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, injecting a large number of data segments to 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, 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 a data frame and several retransmissions of it are lost for some reason. This document does not distinguish the different causes of such a delay spike, but discusses the spurious retransmission timeouts caused by a delay spike in general.

This document describes the F-RTT detection algorithm. It is based on the detection mechanism of the "Forward RTT-Recovery" (F-RTT) algorithm [[SKR03](#)] that is used for detecting spurious retransmission timeouts and thus avoiding unnecessary retransmissions following the retransmission timeout. When the timeout is not spurious, the F-RTT algorithm reverts back to the conventional RTT recovery algorithm and therefore has similar behavior and performance. F-RTT 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 [[BA04](#)]) that have been suggested for detecting unnecessary retransmissions. 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, making it possible for the sender to detect afterwards whether it has retransmitted segments unnecessarily. The F-RTT algorithm only attempts to detect and avoid unnecessary retransmissions after an RTT. Eifel and DSACK can also be used for detecting unnecessary retransmissions caused by other events, for example packet reordering.

When an RTT expires, the F-RTT 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 also advances the window, i.e., acknowledges data that was not retransmitted, the F-RTT sender declares the timeout spurious and exits the RTT recovery. However, if either of these two acknowledgments is a duplicate ACK, there is no sufficient evidence of a spurious timeout; therefore the F-RTT sender retransmits the unacknowledged segments in slow start similarly to the traditional algorithm. With a SACK-enhanced version of the F-RTT algorithm, spurious timeouts may be

detected even if duplicate ACKs arrive after an RTO retransmission.

The F-RTO algorithm can also be applied with the Stream Control

Transmission Protocol (SCTP) [[Ste00](#)], because SCTP has similar acknowledgment and packet retransmission concepts as TCP. For convenience, this document mostly refers to TCP, but the algorithms and other discussion are valid also with 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. [Section 5](#) gives considerations on applying F-RTO with SCTP, and [Section 6](#) discusses the security considerations.

## **[2.](#) F-RTO Algorithm**

A spurious timeout is a timeout that would not have had to occur if the sender had 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 unmodified. When the RTO expires the F-RTO algorithm monitors incoming acknowledgments and declares a timeout spurious, if the TCP sender gets an acknowledgment for a segment that was not retransmitted due to timeout. The actions taken in response to a spurious timeout are not specified in this document, but we discuss the different alternatives in [Section 4](#). This section first describes 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 either be SPUR\_TO, indicating a spurious retransmission timeout; or FALSE, when 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, and if it chooses to apply the algorithm, the following steps MUST be taken after the retransmission timer expires. If the sender implements some

other loss recovery algorithm than Reno or NewReno [[FHG04](#)], F-RTT algorithm SHOULD NOT be entered when earlier fast recovery is underway.

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



the TCP SHOULD 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 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, the TCP sender MUST revert to the conventional RTO recovery and continue by retransmitting unacknowledged data in slow start. The TCP sender MUST NOT enter step 3 of this algorithm, and the SpuriousRecovery variable remains as FALSE.
  - b) Else, if the acknowledgment advances the window AND it is below the value of "recover", the TCP sender SHOULD 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 SHOULD 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](#)], but F-RTO changes which segments are transmitted.

If the TCP sender does not have any new data to send, or the advertised window limits the transmission, the recommended action is to not enter step 3 of this algorithm but 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, the TCP sender MUST set congestion window to no more than  $3 * MSS$ , and continue with the slow start algorithm retransmitting unacknowledged segments. Congestion window can be set to  $3 * MSS$ , because two

round-trip times have elapsed since the RT0, and a conventional TCP sender would have increased cwnd to 3 during the same time. The sender leaves SpuriousRecovery set to FALSE.

- b) If the acknowledgment advances the window, i.e. it acknowledges data that was not retransmitted after the timeout, the TCP

sender SHOULD declare the timeout spurious, set SpuriousRecovery to SPUR\_TO and set the value of "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. Since duplicate ACKs may indicate that segments have been lost, reliably detecting a spurious timeout is difficult in the lack of additional information. Therefore the safest alternative is 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 it has been retransmitted again after an RTO, while the rest of the unacknowledged segments have successfully been 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 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 may run out of segments, and no further acknowledgments would arrive. In this case the recommendation is to revert to the conventional RTO recovery with slow start retransmissions, but [Appendix C](#) discusses some alternative solutions for window limited situations.

If the RTO 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. As the sender does not retransmit other segments but the one that triggered timeout, the problem addressed by the RFC cannot

occur. Therefore, if there are three duplicate ACKs arriving at the sender after the timeout, 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.

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, 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 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.

It is possible that the F-RTO algorithm does not always 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 transmitted before the fast recovery trigger duplicate ACKs. However, we consider these cases relatively rare, and note that in cases where F-RTO fails to detect the spurious timeout, 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-RTT algorithm, that makes use of TCP Selective Acknowledgment Option [[MMFR96](#)]. By using the SACK option the TCP sender can detect spurious timeouts in most of the cases when packet reordering or packet duplication is present. The difference to the basic F-RTT algorithm is that the

sender may declare timeout spurious even when duplicate ACKs follow the RTO, if the SACK blocks acknowledge new data that was not transmitted after the RTO retransmission.

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 loss recovery when retransmission timeout occurs. However, it should be possible to apply the principle of F-RTO within certain limitations also when retransmission timeout occurs during existing loss recovery. While this is a topic of further research, [Appendix B](#) briefly discusses the related issues.

- 1) When the RTO expires, the TCP sender SHOULD retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Variable "recover" is set to indicate the highest segment transmitted so far. Following the recommendation in SACK specification [[MMFR96](#)], the SACK scoreboard SHOULD be reset.
- 2) Wait until the acknowledgment for 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 but stay in step 2 waiting for the next new acknowledgment. If RTO expires again, restart the algorithm.
  - a) if a cumulative ACK acknowledges a sequence number equal to "recover", the TCP sender SHOULD revert to the conventional RTO recovery and it MUST set congestion window to no more than 2 \* MSS. The sender MUST 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, the TCP sender SHOULD 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 due to receiver window limitation or because it does not have any new data to send, the recommended action is to not enter step 3 of this algorithm but 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 also follow the recommendations concerning acknowledgments of retransmitted segments given in [Appendix B](#).



- 3) The next acknowledgment arrives at the sender. Either duplicate ACK or a new cumulative ACK advancing the window applies in this step.
  - a) if the ACK acknowledges sequence number above "recover", either in SACK blocks or as a cumulative ACK, the sender MUST set congestion window to no more than  $3 * MSS$  and proceed with the conventional RTO recovery, retransmitting unacknowledged segments. The sender SHOULD 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. The sender leaves 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, the TCP sender SHOULD 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 SHOULD be retransmitted similarly to the conventional SACK recovery algorithm [[BAFW03](#)]. If the algorithm exits with SpuriousRecovery set to SPUR\_TO, "recover" SHOULD be 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 spurious RTO. This document does not describe the response to spurious timeout, but a response algorithm is described in another IETF document [[LG04](#)].

Additionally, different response variants to spurious retransmission

timeout have been discussed in various research papers [SKR03, GL03, Sar03] and Internet-Drafts [[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 than 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 association 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 address. 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, a 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 spu-

rious, 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-RTT is the possibility of a receiver misleading the sender to set too large a congestion window after an RTT. There are two possible ways a malicious receiver could trigger a wrong output from the F-RTT 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 lead to declare spurious timeout in F-RTT 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 for a retransmission timeout is that a fast retransmission of a segment is lost. If all other segments have been received, the RTT retransmission causes the whole window to be acknowledged at once. This case is recognized in F-RTT algorithm branch (2a). However, if the receiver only acknowledges one segment after receiving the RTT 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 RTT expires during fast recovery phase, the sender would not fully revert the congestion window even if the timeout 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-RTT normally.

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

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, and Samu Kontinen 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.
- [FHG04] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. [RFC 3782](#), April 2004.
- [MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment 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.
- [BA04] E. Blanton and M. Allman. Using TCP Duplicate Selective Acknowledgment (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions. [RFC 3708](#), February 2004.
- [BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP Extensions for

High Performance. [RFC 1323](#), May 1992.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgment (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, Florence, Italy, February 2002

Expires: November 2004

[Page 12]



- [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] R. Ludwig and A. Gurtov. The Eifel Response Algorithm for TCP. Internet draft "[draft-ietf-tsvwg-tcp-eifel-response-05.txt](#)". March 2004. Work in progress.
- [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] R. Ludwig and M. Meyer. The Eifel Detection Algorithm for TCP. [RFC 3522](#), April 2003.
- [Nag84] J. Nagle. Congestion Control in IP/TCP Internetworks. [RFC 896](#), January 1984.
- [SKR03] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RT0: 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. Internet draft "[draft-swami-tsvwg-tcp-dclor-02.txt](#)". September 2003. Work in progress.

## Appendix A: Scenarios

This section discusses different scenarios where RTOs occur and how

the basic F-RTT 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-RTT is given in [[SKR03](#)].

#### **[A.1.](#) Sudden delay**

The main motivation of F-RTT 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-RTT step (1)]
6.  SEND 6 ----->
(cwnd = 6, ssthresh = 3, FlightSize = 6)
  <earlier xmitted SEG 6> --->
7.      <----- ACK 7
      [F-RTT step (2b)]
8.  SEND 12 ----->
9.  SEND 13 ----->
(cwnd = 7, ssthresh = 3, FlightSize = 7)
  <earlier xmitted SEG 7> --->
10.     <----- ACK 8
      [F-RTT step (3b)]
      [SpuriousRecovery <- SPUR_T0]
(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 RT0 retransmission because originally  
transmitted segment 6 arrives 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 case of possible spurious timeout the segments sent before the timeout are still in the network. However, the sender should still be equally aggressive to 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 as spurious and continues by sending new data on 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 again 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. Congestion window can be set to three segments, because two round-trips have elapsed after the retransmission timeout. 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

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

Since F-RTO modifies the TCP sender behavior only after a retransmission timeout and it is intended to avoid unnecessary retransmits only after spurious timeout, we limit the discussion on the effects of packet reordering in F-RTO behavior to the cases where packet reordering 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 also when packet reordering occurs. We illustrate the behavior of SACK-enhanced F-RTO below 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_T0]
    (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: Applying SACK-enhanced F-RTO when RTO occurs during loss recovery

We believe that 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.

The original SACK-based F-RTO requires in algorithm step 3 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 only use 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 segment that was transmitted more than one RTO ago, it can declare the timeout spurious. Defining an efficient algorithm for checking these conditions remains as 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 what segments to retransmit after RTO expires, and whether it is safe to revert the congestion control parameters in this case. This is considered as a topic of future research.

#### Appendix C: Discussion on Window Limited Cases

When the advertised window limits the transmission of two new previously unsent segments, or there are no new data to sent, it was recommended in F-RTO algorithm step (2b) that the TCP sender would continue with conventional RTO recovery algorithm. The disadvantage of doing this is that the sender may continue unnecessary retransmissions due to possible spurious timeout. This section briefly discusses the options that can potentially result in better 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 to have a 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-RTT when ACK losses and a RTT 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 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 timeouts. On the other hand, with this method it is possible to avoid several unnecessary retransmissions due to spurious timeout 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 acknowledgment whether the timeout was spurious. While this method does not send data unnecessarily, it delays the recovery by one round-trip time in cases where the timeout 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.

#### 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](mailto: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

Expires: November 2004

[Page 20]



## IPR Disclosure Agreement

By submitting this Internet-Draft, I certify that any applicable patent or other IPR claims of which I am aware have been disclosed, and any of which I become aware will be disclosed, in accordance with [RFC 3668](#).

## Full Copyright Statement

Copyright (C) The Internet Society (2004). This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

Acknowledgement

Expires: November 2004

[Page 21]

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

Expires: November 2004

[Page 22]