University of Helsinki
Department of Computer Science
Series of Publications C, No. C-2002-07

F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts

Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen

Helsinki, February 2002

Report C-2002-07

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts

Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen
Department of Computer Science, University of Helsinki
Report C-2002-07
February 2002
19 pages

Abstract. Spurious TCP retransmission timeouts (RTOs) have been reported to be a problem on network paths involving links that are prone to sudden delays due to different reasons. Especially many wireless network technologies contain such links. Spurious retransmission timeouts often cause unnecessary retransmission of several segments, which is harmful for TCP performance. Recent proposals for avoiding unnecessary retransmissions after a spurious RTO require use of TCP options which must be implemented at both ends of the connection. We introduce a new TCP sender algorithm for recovery after a retransmission timeout and show that unnecessary retransmissions can be avoided without TCP options. The algorithm effectively avoids the unnecessary retransmissions after a spurious retransmission timeout, improving the TCP performance considerably. The algorithm is friendly towards other TCP connections, because it follows the congestion control principles and injects packets to the network at same rate as a conventional TCP sender. We implemented the algorithm and compared its performance to conventional TCP and Eifel TCP when RTOs occurred either due to sudden delays or due to packet losses. The results show that our algorithm either improves performance or gives similar throughput as the other TCP variants evaluated in different test cases.

Key Words: Mobile computing, Transport protocols, TCP/IP, Performance measurements

CR Classification: C.2.2,C.4,D.4.4,D.4.8

Contents

1 Introduction **1**

2 Related Work on Spurious Retransmission Timeouts **3**

3 F-RTO Algorithm **4**

4 Discussion of F-RTO Behavior in Specific Scenarios **7**

4.1 Sudden delays 7

4.2 Lost retransmission 8

4.3 Burst losses 9

4.4 Packet reordering 10

5 Performance Analysis **10**

5.1 Test Arrangements 10

5.2 Results 12

5.3 Fairness towards conventional TCP 16

6 Future Work and Concluding Remarks **17**

1 Introduction

In the recent years the variety of Internet links with different properties has increased dramatically. The high speed networks have reached Gigabit rates, whereas the increasing number of mobile wireless access networks have introduced a prolific number of mobile hosts attached to the Internet through a slow, wireless links. Moreover, the challenging characteristics of wireless links, in particular high packet loss rate or delays due to various reasons such as link-layer retransmissions or hand-offs between the points of attachment to the Internet, have introduced a large set of problems for the Internet transport protocols.

The TCP protocol [Pos81] is the dominant Internet transport protocol and its congestion control algorithms [APS99] are essential for the stability of the Internet. Because these algorithms have a strong effect on TCP performance, finding solutions to improve TCP performance in various challenging conditions has yielded a large number of studies. For example, a taxonomy of the different solutions for improving TCP performance over slow wireless links can be found in [MDK+00]. The traditional problem regarding the use of TCP over wireless links or other challenging channels has concerned TCP congestion control. If a packet is lost, TCP interprets it as an indication of congestion and a TCP sender needs to reduce its transmission rate. Hence, TCP performance deteriorates with increasing packet loss rate. If the packet loss occurred due to corruption, reducing the TCP transmission rate is, however, the wrong action to take.

TCP uses the *fast retransmit* mechanism [APS99] to trigger retransmissions after receiving three successive duplicate acknowledgements (ACKs). If for a certain time period TCP sender does not receive ACKs that acknowledge new data, the TCP retransmission timer expires as a backoff mechanism. When the retransmission timer expires, the TCP sender retransmits the first unacknowledged segment assuming it was lost in the network. Because a retransmission timeout (RTO) can be an indication of severe congestion in the network, the TCP sender resets its congestion window to one segment and starts increasing it according to the slow start algorithm.

Since wireless networks are often subject to high packet loss rate due to corruption or hand-offs, reliable link-layer protocols are widely employed with wireless links. The link-layer receiver often aims to deliver the packets to the upper protocol layers in order, which implies that the following packets are blocked until the head of the queue is successfully transmitted. Thereby, such reliable link layers involve variable delays on packets instead of packet losses.

Wireless links may also suffer from link outages, which cause persistent data loss for a period of time. If the link outage lasts long enough, it triggers the TCP RTO at the sender which then retransmits the unacknowledged TCP segments. However, if the link layer protocol is highly persistent in its retransmissions, it is able to deliver the original packets to the TCP receiver once the link outage is finished. Therefore the TCP RTO may be triggered spuriously having a bad effect on TCP performance, since it often results in unnecessary retransmissions of many segments [LK00]. Other potential reasons for sudden delays that possibly trigger spurious RTOs include a delay due to tedious actions required to complete a hand-off or re-routing of packets to the new serving access point after the hand-off, arrival of competing traffic on a shared link with low bandwidth, and a sudden bandwidth degradation due to reduced resources on a wireless channel [Gur01, KY02]. Unnecessary retransmissions may originate also from other reasons. For example, packet reordering may cause unnecessary fast retransmits.

In this paper we focus on attacking the TCP performance problems resulting from unnecessary retransmissions that originate from spurious RTOs. The possible solutions for alleviating sacrificed performance can roughly be divided in two categories. One alternative is to avoid the RTOs in the first place by changing the algorithm used for the RTO calculation. Different constants and granularities applied to the standard algorithm documented in [PA00] have been studied [AP99]. In addition, totally new algorithms for setting the RTO timer have been suggested (e.g. [LS00]). However, we believe it is very difficult to come up with an algorithm that results in a good performance in various different network environments.

Another way to mitigate the performance penalty due to spurious retransmission timeouts is to change the TCP sender behavior after a timeout. In particular, there are two symptoms that typically follow the spurious retransmission timeout when regular TCP is used:

- The TCP sender often transmits several segments unnecessarily, because after the RTO and first retransmission the cumulative acknowledgements for the original transmissions appear at the TCP sender one at a time, triggering further unnecessary retransmissions. Often a full TCP window is retransmitted quite unnecessarily.
- The TCP sender unnecessarily adjusts the TCP congestion control parameters and reduces its sending rate.

In this paper we present a new algorithm, called *Forward RTO-Recovery (F-RTO)*, for a TCP sender to recover after a retransmission timeout. Although the main motivation of the algorithm is to recover efficiently from a spurious RTO, we require it to achieve similar performance with the conventional RTO recovery in other situations where RTO may occur. Our approach requires modification only at the TCP sender, while adhering to the TCP congestion control principles. Moreover, the F-RTO recovery algorithm does not require use of any TCP options or additional bits in the TCP header, unlike other recent suggestions for avoiding unnecessary retransmissions, for example Eifel TCP [LK00]. The F-RTO algorithm uses a set of simple rules for avoiding unnecessary retransmissions after a spurious RTO. When the first acknowledgements arrive after retransmitting the segment for which the RTO expired, the F-RTO sender does not immediately continue with retransmissions like the regular RTO recovery does, but it first checks if the acknowledgements advance the window to determine whether it needs to retransmit, or whether it can continue sending new data.

We implemented the F-RTO algorithm in Linux OS and compared its performance to conventional TCP and Eifel TCP in different scenarios where RTOs occurred either due to sudden delays or due to packet losses. The results show that when RTOs are spurious, the F-RTO algorithm significantly improves TCP performance compared to regular RTO recovery and performs slightly better than Eifel TCP. When the RTOs are due to packet losses, F-RTO yields similar throughput as regular TCP while Eifel TCP has performance problems.

The rest of the paper is organized as follows. In Section 2 we describe the problem caused by unnecessary retransmission timeouts in greater detail and discuss the related work with suggested solutions to the problem. In Section 3 we describe the F-RTO algorithm for making forward transmissions after RTO. We continue by giving some examples of the F-RTO algorithm behavior in different situations involving RTOs in Section 4. In Section 5 we describe the experiments made with F-RTO in different network environments and the results of the experiments. Finally, we give some thoughts on the future work on F-RTO and conclude our work in Section 6.

2 Related Work on Spurious Retransmission Timeouts

Most of the retransmissions in current TCP implementations are expected to be triggered by duplicate ACKs. A TCP retransmission timeout is considered as a fallback mechanism for the cases when retransmissions cannot be triggered by duplicate ACKs. More specifically, a RTO-triggered retransmission is needed when a retransmission is lost, or when nearly a whole window of data is lost, thus making it impossible for the receiver to generate enough duplicate ACKs for triggering TCP fast retransmit. Under these assumptions, retransmitting the unacknowledged segments immediately after the RTO is likely to be the most efficient way of recovering.

However, if the underlying network introduces a sudden delay on the packets in flight, the retransmission timer may expire unnecessarily. Particularly, this phenomenon is very possible with the various wireless access network technologies. Figure 1 shows a time-sequence diagram of a TCP transfer, when a 3-second delay occurs on the link. The retransmission timer expires because of the delay, unnecessarily triggering the RTO recovery and retransmission of all unacknowledged segments. This happens because after the delay the ACKs for the original segments arrive at the sender one at the time and each of the ACKs trigger the retransmission of segments for which the original ACKs will arrive after awhile. This continues until the whole window of segments is eventually unnecessarily retransmitted. Furthermore, because a full window of retransmitted segments arrive unnecessarily at the receiver, it generates duplicate ACKs for these out-of-order segments. Later on, the duplicate ACKs unnecessarily trigger fast retransmit at the sender.

There is no known way to prevent the retransmission timeout from expiring because of a sudden delay. However, by having additional information in the TCP segments, the unnecessary retransmissions following the spurious RTO can be avoided. The *Eifel algorithm* [LK00] suggests that the TCP sender indicates whether a segment is transmitted for the first time, or whether it is a retransmission. When this information is echoed back in the acknowledgement, the sender can determine whether the original segment arrived at the receiver and declare the retransmission either correct or spurious action. Based on this knowledge, the sender either retransmits the unacknowledged segments in the conventional way, assuming the RTO was triggered by a segment loss, or reverts the recent changes on the congestion control parameters and continue with transmitting new data. The latter alternative is likely to be the correct action to take when the original segment was acknowledged after the RTO, indicating that the RTO was spurious.

The Eifel algorithm suggests using either the TCP timestamps option [BBJ92] or two of the reserved bits in the TCP header for distinguishing the original transmissions from retransmissions. Using the reserved bits in the TCP header requires modification to TCP at both ends. The TCP timestamps option is deployed on some Internet hosts[1], but in order to take advantage of Eifel, timestamps option would need to be deployed at both ends of the TCP connection. Given that the sudden delays are often a problem on wireless links with low bandwidth, including timestamps in each TCP segment increases the TCP header overhead and makes the communication inefficient. Moreover, the TCP timestamps are not supported in the current IP/TCP header compression specifications [Jac90, DNP99].

[1] A study on use of the different TCP options indicates that 15 % of the WWW clients connected to a WWW server on the Internet used TCP timestamps in the early 2000s [All00].

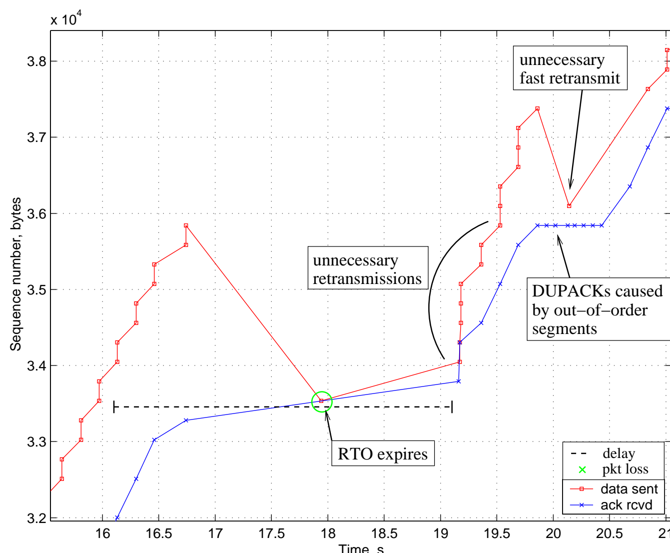


Figure 1: A delay triggers spurious retransmission.

Instead of distinguishing the ACKs of the original transmissions from the ACKs of the retransmissions at the TCP sender, the receiver can indicate whether it received a segment that had arrived earlier. The *Duplicate SACK (D-SACK)* enhancement [FMMP00] suggests to use the first SACK block to indicate duplicate segments arriving at the receiver. This alternative has its benefits over the Eifel algorithm presented above, because the SACK option is being more widely deployed than the TCP timestamps [All00], and the SACK blocks are appended to the TCP headers only when necessary. However, if the unnecessary retransmissions occurred due to spurious RTO caused by a sudden delay, the acknowledgements with the D-SACK information arrive at the sender only after the acknowledgements of the original segments. Therefore, the unnecessary retransmissions following the spurious RTO cannot be avoided by using D-SACK. Instead, the suggested recovery algorithm using D-SACK can only revert the congestion control parameters to the state preceding the spurious retransmission [BA02]. Both ends of the TCP connection need to be aware of the D-SACK extension in order to take advantage of it.

3 F-RTO Algorithm

The F-RTO algorithm affects the TCP sender behavior only after a retransmission timeout, otherwise the behavior is similar to the conventional TCP. The main principle behind the F-RTO algorithm is as follows: if the first ACK arriving after a RTO-triggered retransmission advances the window, transmit two new segments instead of continuing retransmissions. If also the second incoming acknowledgement advances the window, RTO was likely to be spurious, because the second ACK is triggered by an originally transmitted segment the RTO. If either one of the two acknowledgements after RTO is a duplicate ACK, the sender continues retransmissions similarly to the regular RTO recovery algorithm.

When the retransmission timer expires, the F-RTO algorithm takes the following steps at the TCP sender. In the algorithm description below we use `SND.UNA` to indicate the first unacknowledged segment.

1) *When the retransmission timer expires, retransmit the segment that triggered the timeout.* As required by the TCP congestion control specifications, the *slow start threshold (ssthresh)* is adjusted to half of the number of currently outstanding segments. However, the congestion window is not yet set to one segment, but the sender waits for the next two acknowledgements before deciding on what to do with the congestion window.
2) When the first acknowledgement after 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.
   i) *If the acknowledgement advances* `SND.UNA`*, transmit two new segments.* This is the main point in which the F-RTO algorithm differs from the traditional way of recovering from RTO. After transmitting the two new segments, the congestion window size is set to have the same value as `ssthresh`. In effect this reduces the transmission rate of the sender to half of the transmission rate before the RTO. At this point the TCP sender has transmitted a total of three segments after the RTO, similarly to the conventional recovery algorithm. If transmitting new data is not possible due to advertised window limitation, or because there is no more data to send, the sender follows the conventional RTO recovery algorithm and starts retransmitting the unacknowledged data using slow start.
   ii) *If the acknowledgement is duplicate ACK, set the congestion window to one segment.* Two new segments are not transmitted in this case, because the conventional RTO recovery algorithm would not transmit anything at this point either. Instead, the F-RTO sender continues with slow start and performs similarly to the conventional TCP sender in retransmitting the unacknowledged segments. Step 3 of the F-RTO algorithm is not entered in this case. A common reason for executing this branch is the loss of a segment, in which case the segments injected by the sender before the RTO may still trigger duplicate ACKs that arrive at the sender after the RTO.
3) When the second acknowledgement after the RTO arrives, either continue transmitting new data, or start retransmitting with the slow start algorithm, depending on whether new data was acknowledged.
   i) *If the acknowledgement advances* `SND.UNA`*, continue transmitting new data following the congestion avoidance algorithm.* 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. However, since the TCP sender cannot surely know at this point whether the segment that triggered the RTO was actually lost, adjusting the congestion control parameters after the RTO is the correct action. From this point on, the TCP sender continues as in the normal congestion avoidance.

      If this algorithm branch is taken, the TCP sender ignores the `send_high` variable that indicates the highest sequence number transmitted so far [FH99]. The `send_high` variable was proposed as a *"bugfix"* 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. Alternatively, 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.

ii) *If the acknowledgement is a duplicate ACK, set congestion window to three segments, continue with the slow start algorithm retransmitting unacknowledged segments.* 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 that any of the segments was delayed. 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.

If either one of the two acknowledgements arriving after the RTO is a duplicate ACK, the algorithm is safe, because it reverts back to the conventional retransmissions and adjusts the congestion window appropriately. However, the validity of the algorithm when the two first acknowledgements advance `SND.UNA` is worth discussing. As described above, this indicates that at least one segment was delayed. If the next segments in the window were also delayed, for example being blocked by the first delayed segment, the algorithm performs as intended, as we will show in Section 4. If the next segments would not have been delayed, they would have arrived before the delayed segment and triggered duplicate ACKs. We will discuss the F-RTO behavior under packet reordering in more detail in Section 4.

When algorithm branch (3i) is taken, the sender does not reduce the congestion window to one segment, but halves it to the level of `ssthresh`. Because the sender does not enter slow start, it is slightly more conservative than the conventional recovery algorithm. In fact, if the segment that triggered RTO was not lost, the correct behavior would have been to not decrease the congestion window at all. If the D-SACK option is in use, the sender can detect whether the retransmission was unnecessary, and revert the last adjustments on the congestion control parameters in such a case.

Executing algorithm branch (3i) could transmit a burst of segments into the network if the RTO is followed by an acknowledgement that advances `SND.UNA` by several segments at once, while the congestion window is still large enough after halving it. Therefore, we require that such bursts should be avoided by some method. For example, the Linux TCP implementation we are using ensures that the sender never transmits more than three segments when a new acknowledgment arrives by reducing the congestion window appropriately.

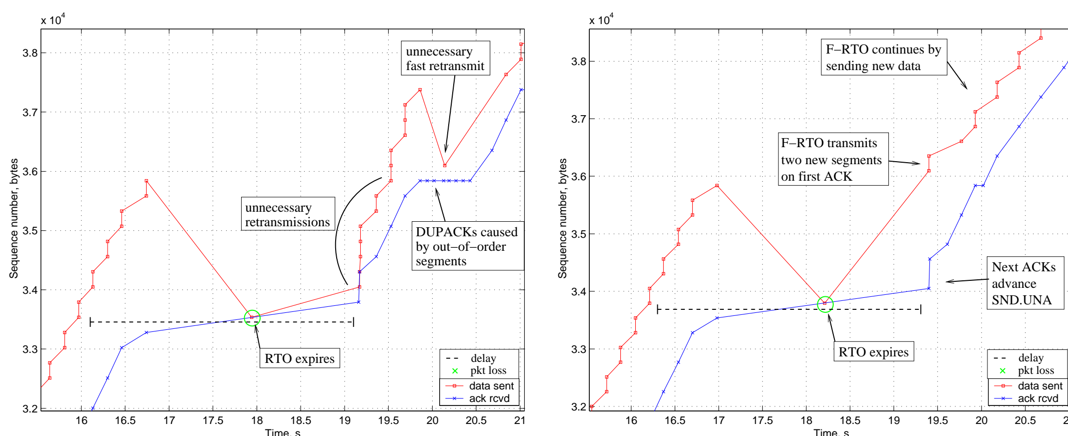
Branch (3i) can also be taken when a segment loss is immediately followed by a delay. In this case the retransmission triggered by the timer was not unnecessarily made. Arrival of new acknowledgements after the RTO indicates that there was only one segment loss in addition to the excessive delay which trigger the RTO. Therefore, we consider that reducing the congestion window to half of its previous size is an adequate action at this point.

4 Discussion of F-RTO Behavior in Specific Scenarios

In this section we discuss the different reasons which may cause the RTO to expire and study the different scenarios after a RTO has expired due to these reasons. We compare the packet traces produced using the regular RTO recovery and using F-RTO, and discuss the differences of the two recovery methods. *Selective Acknowledgements (SACK)* [MMFR96] and *limited transmit* [ABF01] TCP enhancements are used in the examples presented in this section, since SACK can be considered rather widely deployed today, and limited transmit is a sender side modification that can be implemented with F-RTO to further improve the TCP performance. However, the F-RTO algorithm does not require either of these enhancements to be present.

4.1 Sudden delays

Recovering efficiently from spurious retransmission timeouts is the main motivation of the F-RTO algorithm. Figure 2 compares the packet traces of the regular RTO recovery and F-RTO. Figure 2(a) shows that the regular recovery method eventually retransmits the whole window of segments unnecessarily, since the acknowledgements of the originally transmitted segments arrive at the sender after the RTO. When the retransmissions arrive at the receiver, it generates a duplicate ACK for each arriving retransmission, thus causing an unnecessary fast retransmit at the TCP sender.



(a) Conventional RTO recovery.

(b) F-RTO recovery.

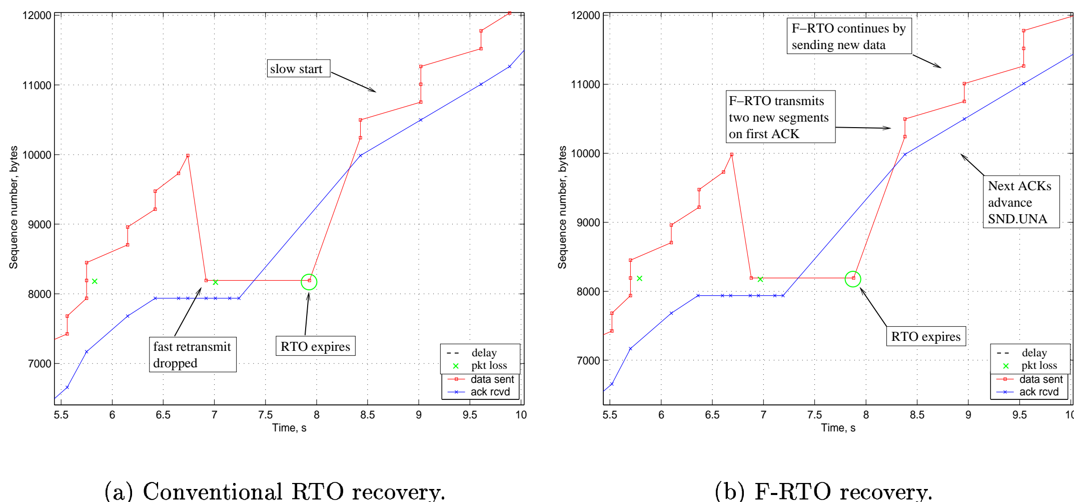
Figure 2: Comparison of the regular RTO and F-RTO after an excessive delay.

Figure 2(b) shows that F-RTO avoids the unnecessary retransmissions following the spurious RTO. The first acknowledgement arriving at the sender after the RTO advances `SND.UNA`, and the sender transmits two previously unsent segments. The second ACK arriving after the RTO acknowledges two originally transmitted delayed segments, hence the sender continues transmitting new data. However, since the congestion window was reduced after the RTO, the sender waits for a few

acknowledgements without sending new segments to balance the number of packets in flight towards the present congestion window size.

4.2 Lost retransmission

A common reason for triggering TCP RTO is the loss of a retransmitted segment. Once a segment has been retransmitted, it can only be retransmitted again after the RTO expires. Figure 3 compares the packet traces of the conventional RTO recovery with the traces of the F-RTO recovery when a retransmitted segment is lost and it is retransmitted again as triggered by RTO. One can notice that the behavior of the traditional RTO recovery and the F-RTO recovery is similar. In the scenario shown, both variants get to transmit two new segments after the RTO, and then proceed with transmitting new data.



(a) Conventional RTO recovery.

(b) F-RTO recovery.

Figure 3: Comparison of the traditional RTO and F-RTO after a lost retransmission.

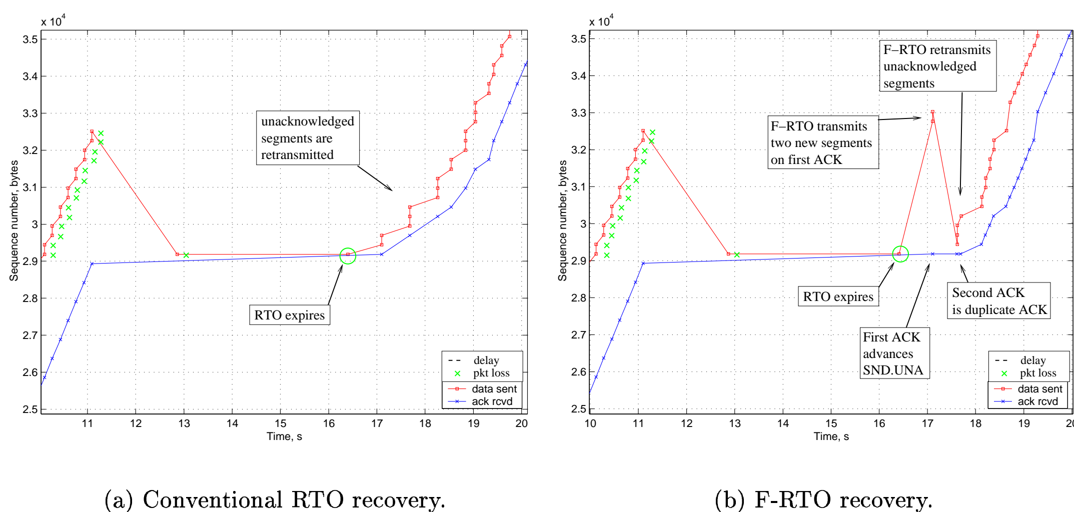
Figure 3(a) shows that when the RTO retransmission arrives at the receiver, it acknowledges the whole window, and the conventional TCP sender can proceed with sending new data in slow start. In the presented case the F-RTO recovery shown in Figure 3(b) differs from the regular recovery only by not entering slow start after the RTO. Because the next ACK arriving at the sender after the RTO acknowledges all outstanding packets, that is, advances `SND.UNA`, the F-RTO sender transmits new segments using congestion avoidance. Instead of setting the congestion window to one segment, F-RTO decreases it to half of its previous size. As one can see, the practical difference between the recovery alternatives is negligible because the number of outstanding packets was rather small when the first packet loss occurred in the presented scenario.

Using congestion avoidance instead of slow start after the F-RTO recovery does not limit the TCP performance in cases where the number of outstanding segments is larger than in the example above. However, because F-RTO sets the congestion window to half of its previous size when the next acknowledgements advance `SND.UNA`, and on the other hand, because we require using burst

avoidance with F-RTO, the conventional RTO recovery algorithm and F-RTO result in similar performance. In our implementation the burst avoidance method decreases the congestion window to allow transmitting at most three segments for the first incoming ACK. If the congestion window size is reduced below the slow start threshold, the sender uses slow start in adjusting the congestion window when the next acknowledgements arrive.

4.3 Burst losses

Because losses of several successive packets can result in a retransmission timeout, it is interesting to compare the F-RTO behavior with the regular RTO recovery in such a case. Figure 4 compares the packet trace of the regular recovery after a RTO caused by a window of lost segments with the packet trace of the F-RTO recovery. One can see from Figure 4(a) that the segment retransmitted after the second RTO is successfully acknowledged, after which the TCP sender retransmits the rest of the lost segments in slow start.



(a) Conventional RTO recovery.

(b) F-RTO recovery.

Figure 4: Comparison of the regular RTO and F-RTO after a burst loss.

Figure 4(b) shows a similar scenario with a F-RTO sender. When the segment retransmitted due to RTO is acknowledged, the F-RTO sender transmits two new segments. Because several other segments were dropped in the last window, the two new segments trigger duplicate ACKs. As given by the F-RTO algorithm, the arrival of the duplicate ACK as the second acknowledgement following the RTO makes the sender retransmit unacknowledged segments in slow start. When the second acknowledgement after the RTO arrives, the sender has a congestion window of three segments, similarly to the conventional RTO recovery after two round-trip times. From this point on the congestion window is increased according to the standard TCP congestion control specifications. More generally, if there are any packets lost in the last window of data in addition to the one that triggers RTO, the F-RTO sender enters slow start and retransmits the unacknowledged segments, because the two new segments transmitted after the RTO would trigger duplicate ACKs at the receiver.

F-RTO has a side-effect of triggering an acknowledgement for every incoming retransmission at the TCP receiver, because the receiver is required to send an immediate ACK when it has out-of-order segments in its buffers [APS99]. However, we believe this detail does not increase the stress on the network significantly, since it only affects the TCP sender's transmission rate during the slow start.

4.4 Packet reordering

Packet reordering is a scenario worth discussing when evaluating the F-RTO behavior, although packet reordering does not usually cause the retransmission timer to expire. A more detailed study on the effect of packet reordering on TCP performance can be found in [BA02], hence we only discuss here how the F-RTO algorithm relates to packet reordering.

A delayed segment that arrives at the TCP receiver out-of-order appears as a hole in the sequence number space of incoming packets, thus having largely similar effects on the TCP behavior than a dropped packet. Packet reordering may cause fast retransmit, but if there are no retransmission timeouts involved, the F-RTO algorithm does not change the TCP behavior from the conventional recovery. A more interesting scenario arises if the RTO timer expires while packets arrive at the receiver out of order. If the out-of-order segments cause duplicate ACKs to arrive at the sender after the RTO, the F-RTO sender retransmits the unacknowledged packets, similarly to the conventional RTO algorithm. If the delayed packets trigger new acknowledgements which arrive at the sender just after the RTO, the F-RTO sender proceeds with sending new data. This is likely to be the correct action, because the acknowledgements were triggered by a segment transmitted before the retransmission timeout.

5 Performance Analysis

In order to validate the discussion in Section 4, we made experiments in networks with characteristics similar to those that could be expected when communicating over a bottleneck wireless link to a fixed server in a nearby network. This is a typical environment where scenarios presented in Section 4 may occur. We compared the F-RTO performance to the performance achieved with the regular RTO recovery, both with SACK TCP and with NewReno TCP. In addition, we conducted experiments with the Eifel algorithm.

5.1 Test Arrangements

The general test setup is illustrated in Figure 5. We emulate the wireless link and the last-hop router by using a real-time wireless network emulator. The end hosts are Linux 2.4 systems, in which we implemented the F-RTO algorithm. The fixed link is an isolated LAN which is connected to the remote host and to the network emulator.

We selected the link parameters to approximate the properties of a typical currently used wireless wide-area networking system. The emulated wireless link has a bandwidth of 28,800 bps and a propagation delay of 200 ms. The last-hop router has a router buffer for holding seven packets. In addition to the router buffer, the emulated wireless link uses a link send buffer and a link receive

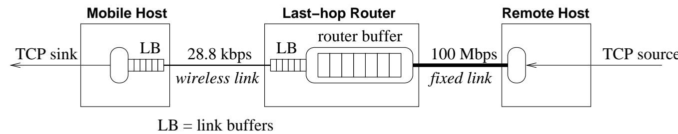


Figure 5: Test setup.

buffer for both uplink and downlink traffic. Any link that provides a retransmission mechanism needs to have a certain amount of buffering capacity. The link send buffer holds packets that have not yet been acknowledged as received, and the link receive buffer collects out-of-order packets for delivering them for the upper layer receiver in the correct order. The link buffers have a size of 1776 bytes, which is large enough to cover the bandwidth-delay product of the link.

We made one set of experiments with a wireless link that does not drop packets, but randomly inflicts sudden delays for some packets. Another set of experiments was made using an unreliable link that drops random packets with given packet drop probabilities. Finally, experiments were conducted by having periods of persistent packet loss on the link. The link scenarios are listed below:

- **Sudden delays**. Since the primary motivation of the F-RTO algorithm is to improve the TCP performance when sudden delays cause spurious retransmission timeouts, we start by a scenario that involves sudden delays on the link. We explained the possible reasons for a sudden delay on the wireless access network in the introduction. Such a delay can occur, for example, due to loss burst with a link layer protocol providing highly persistent reliability. During the delay no packets are delivered to the receiver. In this scenario a packet is delayed with a probability of 0.02. The random delay lengths are exponentially distributed with a mean delay length of 3.5 seconds. Exponential distribution has been reported to characterize the length of the loss periods on a wireless link reasonably well [KZJL01]. Even though the link is reliable in this scenario, packet losses may occur due to congestion at the last-hop router.
- **Packet losses**. In this scenario a packet is randomly dropped by the link with given probability. This scenario models the case of an unreliable link layer over a lossy link. Therefore the packet delays on the link are fairly constant. We tested packet loss probabilities of 2 %, 5 %, and 10 %. The packet losses are uniformly distributed. The main purpose of these scenarios is to test the TCP performance when the retransmission timeouts occur due to lost segments, mainly because retransmissions are lost.
- **Bursty losses**. This scenario is to model the effect of link outages when the link layer is not reliable and drops several successive packets. The link conditions are split into two distinct states. In a good state no packets are dropped at the link. When the link is in bad state, all packets in both directions are lost. The link layer does not retransmit any packets. The two states alternate randomly. The good state length is uniformly distributed between 0.1 seconds and 20 seconds. The bad state duration is exponentially distributed with a mean of 3.5 seconds.

In each of the scenarios presented above we test five TCP variants based on the Linux TCP implementation. For the purposes of the experiments, we disabled the ratehalving algorithm used

by default in the Linux TCP implementation, and made the necessary modifications for supporting the Eifel algorithm. In addition, we modified the SACK loss recovery to follow the conservative algorithm currently under work in the IETF. Firstly, we test a SACK TCP [MMFR96] with the conventional RTO recovery, and with the F-RTO recovery. Secondly, we do experiments with a NewReno TCP [FH99] with both conventional and F-RTO recovery algorithms. Finally, we test a TCP variant using the TCP timestamp option both with the SACK TCP and with the NewReno TCP. This variant implements the Eifel algorithm based on the use of TCP timestamps. Our Eifel sender implementation continues transmitting new data and undoes the changes made on the congestion window and `ssthresh` when it detects a spurious timeout from the timestamps. The limited transmit algorithm is used with all TCP alternatives.

We use unidirectional 100 KB bulk transfers from the fixed end source to the mobile end sink as the workload. The data is transmitted using a single TCP connection using a maximum segment size of 256 bytes. A small segment size is recommended for slow links in order to achieve a better interactive response times [MDK+00], although this is a factor not significant in our tests. For each scenario and TCP variant the experiment is repeated 30 times.

5.2 Results

We present the results of the experiments by using box-plot diagrams. The diagrams compare the throughput of each TCP variant evaluated in the experimentation. The box-plot diagram shows the median throughput for the 30 replications with a horizontal line splitting the filled box. The lower and upper edge of the box represent the 1st and 3rd quartiles of the test results, respectively. The whiskers are drawn at the minimum and the maximum throughput measured with the TCP variant. On rare occasions some test runs were involved with a notably different number of RTOs than the majority of the tests due to randomness of the link events. Because the RTOs typically have a strong effect on the TCP performance, the minimum or maximum throughput values may appear to differ considerably from the results within the upper and lower quartiles in some cases.

In addition to the box-plot diagrams we show with each scenario a table presenting the median values for connection elapsed time from sending the first SYN packet to receiving the last FIN acknowledgement at the sender, the number of packet losses, and the number of retransmitted segments of each TCP variant. If the number of retransmissions is higher than the number of lost packets, at least some of the retransmissions are made unnecessarily. On the other hand, the number of lost packets can be higher than the number of retransmissions, because lost acknowledgements do not necessarily trigger retransmissions.

Sudden delays

Figure 6 shows the box-plot diagrams of the throughput measured with different TCP variants. Additionally, Table 1 shows the median values for the connection statistics described above. The results show that using F-RTO improves performance over the regular RTO recovery both with the SACK TCP and with the NewReno TCP. The number of unnecessary retransmissions with the F-RTO algorithm is considerably smaller than with the conventional RTO recovery algorithm,

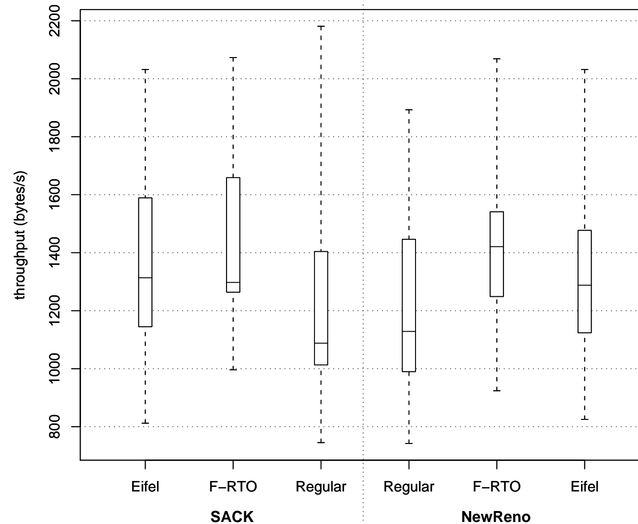


Figure 6: TCP performance with different variants with excessive delays on the link.

Table 1: Results of the tests with sudden delays. The median values of 30 replications.

TCP Variant	**Time (s) / 100 KB**	**Pkts Lost**	**Nr. of Rexmits**
Eifel w/ SACK	77.94	11	16
F-RTO w/ SACK	76.23	4	12
Regular SACK	94.13	9	57
Regular NewReno	90.72	10	60
F-RTO w/ NewReno	75.18	6	13
Eifel w/ NewReno	79.21	11	19

resulting in improved throughput with the F-RTO algorithm. There is no significant difference between the SACK TCP and the NewReno TCP, when RTOs are triggered by excessive delays.

The Eifel TCP avoids most of the unnecessary retransmissions similarly to the F-RTO algorithm. However, the Eifel sender reverts the congestion control parameters back to the values preceding the spurious RTO, and continues sending at the previous rate although the last-hop router could not drain the queue during the delay. Hence, Eifel typically has more packet losses due to congestion than F-RTO, resulting in a slightly lower throughput than F-RTO. This suggests that responding to the spurious RTO by directly reverting the congestion control parameters may be too aggressive action to take.

Packet losses

Figure 7 illustrates the throughput distribution with different TCP variants when the wireless link has a packet loss rate of 5 %. The trend with the packet loss rates of 2 % and 10 % is similar: the

Table 2: Results of the tests with packet errors. The median values of 30 replications.

TCP Variant	**Time (s) / 100 KB**	**Pkts Lost**	**Nr. of Rexmits**
Eifel w/ SACK	80.68	39	26
F-RTO w/ SACK	75.69	36	24
Regular SACK	76.18	36	22
Regular NewReno	82.38	36	26
F-RTO w/ NewReno	81.67	36	26
Eifel w/ NewReno	89.64	38	27

performance of F-RTO is not different from the performance achieved with the conventional RTO recovery, regardless of whether SACK or NewReno TCP is used. In these tests the retransmission timeouts are usually due to lost retransmissions. After the TCP sender has successfully retransmitted the segment which triggered the RTO, it can usually proceed with transmitting new data. Table 2 shows that the number of retransmissions are similar with all TCP variants tested. As expected, the SACK TCP improves the performance over the NewReno TCP, since there are often multiple packet losses in one round-trip time, and SACK recovers more efficiently in such a case.

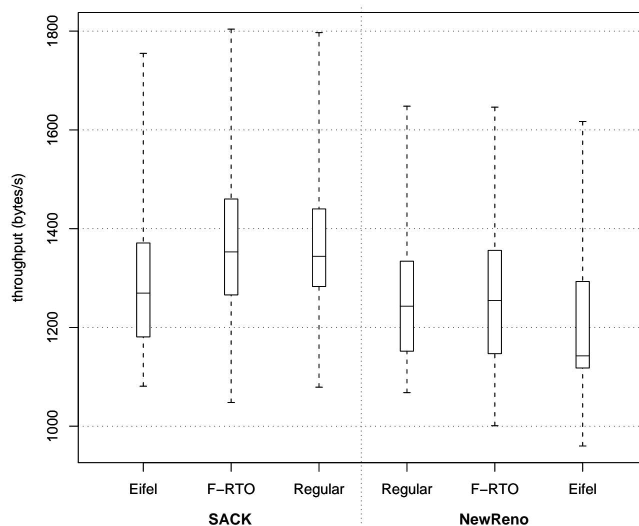


Figure 7: TCP performance with different variants with packet drop probability of 0.05.

Eifel TCP using SACK and TCP timestamps has a lower throughput than SACK TCP without timestamps. However, a closer examination of the TCP packet traces does not show any problems related to the Eifel algorithm. The difference is explained due to use of the TCP timestamps which adds 12 bytes of overhead to each packet transmitted, resulting in approximately a 4 % increase in the number of packets to send with the small segment size we were using. By using a larger segment size the additional packet overhead would have had less effect on the results.

Table 3: Results of the tests with bursty losses on the link. The median values of 30 replications.

TCP Variant	**Time (s) / 100 KB**	**Pkts Lost**	**Nr. of Rexmits**
Eifel w/ SACK	123.89	50	45
F-RTO w/ SACK	64.94	42	40
Regular SACK	71.23	42	39
Regular NewReno	74.48	42	43
F-RTO w/ NewReno	67.80	39	43
Eifel w/ NewReno	80.03	42	42

Bursty losses

Figure 8 shows that the TCP performance with F-RTO does not differ significantly from the performance with the regular RTO recovery when there are link outages. As described in Section 4.3, the F-RTO sender transmits segments at a similar rate as the conventional RTO recovery, although it transmits two new segments before continuing retransmissions. The difference of whether to transmit the two new segments before or after the retransmissions, does not affect the throughput. Use of the SACK TCP does not notably improve the performance with bursty losses, especially if the losses trigger a retransmission timeout. After the RTO the TCP sender retransmits the unacknowledged segments in slow start, regardless of whether SACK TCP or NewReno TCP is used. Table 3 shows the median connection times and the retransmission statistics for the different TCP variants.

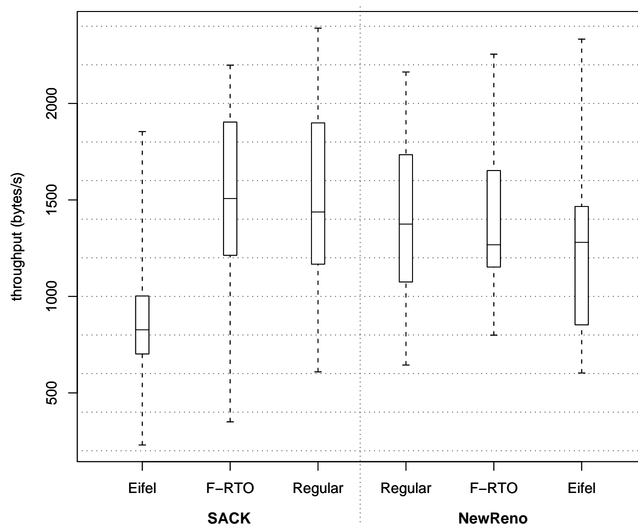


Figure 8: TCP performance with different variants with bursty losses on the link.

The test results show that Eifel TCP gives a clearly worse throughput than the regular TCP when SACK TCP is used. Our experiments revealed a significant problem when using TCP timestamps

for detecting unnecessary retransmissions in Eifel TCP. We will describe the problem below.

When the link is in the bad state as in our link outage scenario, all packets are dropped for a period of time. Therefore, the latest cumulative acknowledgements generated by the receiver are also dropped by the link. This usually leads to a retransmission timeout and an unnecessary retransmission of a segment that had already arrived at the receiver, but for which the acknowledgement was lost. When this unnecessary retransmission arrives at the receiver, it appears as an out-of-order segment and generates a duplicate ACK carrying a timestamp of an earlier data segment[2]. Furthermore, because the earlier acknowledgements were lost during the link outage, the duplicate ACK appears as an acknowledgement for new data to the TCP sender. Therefore, the Eifel decision rules declare that the retransmission was spurious, although a number of data segments were lost in the last window.

The Eifel sender responds to the spurious retransmission indication by sending new data and reverting the congestion control variables. However, in the case described above the sender gets back duplicate ACKs because there were data segments missing. The sender enters fast recovery due to the duplicate ACKs and reduces the congestion window. At this point the sender stops sending data for a while to balance the number of outstanding packets to the congestion window size. Because the sender needs to wait for the halved congestion window's worth of acknowledgements to arrive before it can continue retransmitting, and on the other hand, many of the packets were dropped due to link outage, the pipe runs out of packets while the sender is waiting for incoming acknowledgements. Therefore, the Eifel sender has to wait for another RTO to continue the retransmissions for the rest of the lost segments. This leads to a significant degradation of throughput. Unlike SACK, the NewReno TCP ensures that a retransmission is made for each partial ACK. Therefore the Eifel sender often avoids the second RTO with NewReno.

The events presented above showed up very frequently in our experiments with bursty losses, which explains the poor throughput of Eifel TCP in these tests. The reported behavior is specific to TCP timestamps used as an indication of spurious retransmissions, and we do not believe it to show up, if some other mechanism was used for indicating spurious retransmissions instead of TCP timestamps [3].

5.3 Fairness towards conventional TCP

We expect the connections using the F-RTO algorithm to be friendly towards the TCP connections with regular RTO recovery, because F-RTO is ACK-clocked and it transmits data at an equal rate as the conventional TCP. We back up this reasoning by conducting experiments that use six parallel bulk TCP connections as a workload over the bottleneck wireless link. The workload is separated in two connection sets having three connections each. The three TCP connections in connection set 1 are started at the same time, and the other three TCP connections in connection set 2 are started three seconds after the first connection set. The purpose of this study is to measure how much the

[2]The specification for TCP round-trip time measurements [BBJ92] requires that the echoed timestamp should correspond to the most recent data segment which advanced the window

[3]Some TCP implementations do not strictly follow RFC 1323 by echoing the timestamp of a retransmitted segment arriving out-of-order at the receiver. Such implementation would have avoided the problem described here, but may be vulnerable to IP-spoofing attacks.

connections in connection set 2 interfere with the data transfer in connection set 1. Especially, the effect of the new F-RTO connections on the ongoing TCP transmissions should not differ from the effect of conventional TCP connections.

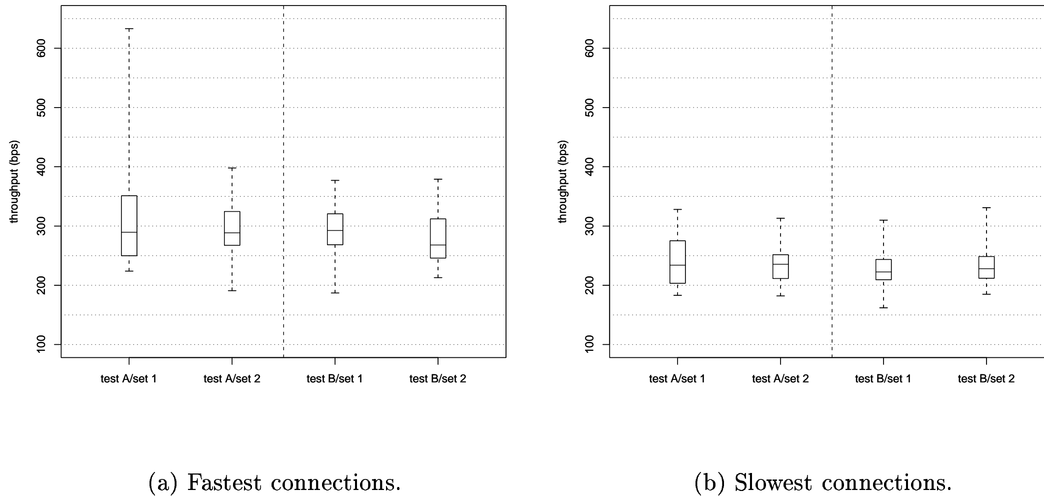
The test setup with multiple TCP connections is similar to the setup presented earlier in Figure 5, with the exception that it consists of six TCP connections separated in two connection sets. Connection set 1 consists of three TCP connections that use the regular RTO recovery. Connection set 2 has another three TCP connections, that use F-RTO in test A, and the regular RTO recovery in test B. All connections transfer 50 KB of bulk data from the remote host to the mobile host. This experiment was made both with and without additional sudden delays on the link. As with the experiments described earlier, the bottleneck link bandwidth is 28,800 bps and the input queue length is 7 packets. Injecting packets from six bulk TCP connections on this kind of network results in severe congestion, which causes a number of packet losses and RTOs triggered at the TCP sender. We repeated the experimentation 20 times.

For each connection set we measured the throughput of the TCP connection that was the last to finish its data transfer, i.e. the slowest connection of its connection set. This metric gives a coarse understanding about the fairness between the TCP connections, because a low throughput of the slowest connection often indicates that the other connections have used a larger share of the common bandwidth. Correspondingly, a high throughput of the slowest connection indicates that the equality between the parallel connections is better. In addition, we present the throughput distribution of the fastest connections in the connection sets.

Figure 9 shows the results of tests with additional delays. Figure 9(a) shows the throughput distribution of the fastest connection for both connection sets in test A using F-RTO connections in connection set 2, and in test B using the conventional RTO recovery in all TCP connections. Figure 9(b) gives the throughput of the slowest connections in the connection sets. The box-plot diagrams show that the connection sets between test A and test B give similar performance. This indicates that the influence of the three new F-RTO connections on the existing TCP connections on the link is not different from the effect of starting three new regular TCP connections. Similarly, the results of the experiments without random additional delays do not show significant difference between the test runs involving F-RTO connections and the test runs having only TCP connections with the conventional RTO recovery. The results support our reasoning of F-RTO being friendly towards the TCP connections with the regular RTO recovery.

6 Future Work and Concluding Remarks

In this work we established that it is possible to avoid most of the unnecessary retransmissions following the spurious TCP retransmission timeouts without any additional information in the TCP packet headers. We presented the F-RTO algorithm which avoids the unnecessary retransmissions following the spurious RTO by determining based on the incoming acknowledgements whether to retransmit or continue sending new data. In addition, because the use of F-RTO algorithm effectively avoids unnecessary retransmits, it obviates the NewReno "bugfix" rule which disables fast retransmit during F-RTO recovery. This allows more efficient recovery from packet losses in some scenarios. An F-RTO sender follows the conventional TCP congestion control principles by sending



(a) Fastest connections.

(b) Slowest connections.

Figure 9: Effect of parallel connections on TCP performance. Test A includes three F-RTO connections, test B uses only regular RTO recovery.

data at an equal rate as the regular TCP and by being clocked by incoming acknowledgements. We showed by experiments that F-RTO improves the TCP performance when there are sudden delays on the link, and it yields competitive performance if the RTOs are caused because of other reasons than delays.

Unlike the Eifel and D-SACK enhancements, the F-RTO algorithm does not provide means for reverting the congestion control information if a RTO occurred spuriously. However, in our future work we intend to run experiments having F-RTO combined with D-SACK, gaining the benefits of the two algorithms: F-RTO avoids the unnecessary retransmissions after a spurious RTO, and D-SACK provides the possibility for reverting the congestion window state after it was adjusted due to the spurious RTO. We also believe that adopting a retransmission algorithm similar to F-RTO would help the Eifel TCP to avoid problems in the scenarios involving ACK losses. In addition to evaluating the different combinations of these algorithms, we intend to study whether it is reasonable to totally revert the congestion control state when detecting a spurious retransmission timeout, or whether a more careful approach for increasing the congestion window after a spurious RTO should be taken. Our experiments suggested that reverting the congestion window may not always be the right choice to take on bottleneck links, since it tends to result in an increased number of congestion losses.

So far we have verified the F-RTO algorithm by implementing it in the Linux OS and running experiments by emulating the expected behavior of the wireless link. Taking this approach makes it possible to study the F-RTO performance in a real network environment when the TCP traffic is generated by the commonly used network applications, which we intend to do as a future work item. Furthermore, we will make a reference implementation of the F-RTO algorithm for the widely used *ns2* simulator in order to conduct simulation studies in various different network setups.

References

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

[All00] M. Allman. A Web Server's View of the Transport Layer. *ACM Computer Communication Review*, 30(5), October 2000.

[AP99] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *Proceedings of ACM SIGCOMM '99*, September 1999.

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

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

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

[DNP99] M. Degermark, B. Nordgren, and S. Pink. IP Header Compression. RFC 2507, February 1999.

[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 Acknowledgment (SACK) Option for TCP. RFC 2883, July 2000.

[Gur01] A. Gurtov. Effect of Delays on TCP Performance. In *Proceedings of IFIP Personal Wireless Communications 2001*, Lappeenranta, Finland, August 2001.

[Jac90] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, February 1990.

[KY02] F. Khafizov and M. Yavuz. Running TCP over IS-2000. In *Proceedings of IEEE ICC 2002*, April 2002. To appear.

[KZJL01] A. Konrad, B.Y. Zhao, A. Joseph, and R. Ludwig. A Markov-Based Channel Model Algorithm for Wireless Networks. In *Proceedings of ACM MSWiM 2001*, pages 28–36, Rome, Italy, July 2001.

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

[LS00] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communication Review*, 30(3), July 2000.

[MDK+00] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, and N. Vaidya. Long Thin Networks. RFC 2757, January 2000.

[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.