Department of Computer Science
Series of Publications A
Report A-2007-1

TCP Performance in Heterogeneous Wireless Networks

Pasi Sarolahti

Academic Dissertation

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium 6, Metsätalo, Unioninkatu 40, on June 16th, 2007, at 10 o'clock.

University of Helsinki
Finland

ISSN 1238-8645
ISBN 978-952-10-3973-7 (paperback)
ISBN 978-952-10-3974-4 (PDF)
http://ethesis.helsinki.fi/

Computing Reviews (1998) Classification: C.2.6, C.4

Helsinki University Printing House
Helsinki, June 2007 (171 pages)

TCP Performance in Heterogeneous Wireless Networks

Pasi Sarolahti
Nokia Research Center
P.O. Box 407, FI-00045 Nokia Group, Finland
pasi.sarolahti@iki.fi
http://www.iki.fi/pasi.sarolahti/

Abstract

The TCP protocol is used by most Internet applications today, including the recent mobile wireless terminals that use TCP for their World-Wide Web, E-mail and other traffic. The recent wireless network technologies, such as GPRS, are known to cause delay spikes in packet transfer. This causes unnecessary TCP retransmission timeouts.This dissertation proposes a mechanism, *Forward RTO-Recovery (F-RTO)* for detecting the unnecessary TCP retransmission timeouts and thus allow TCP to take appropriate follow-up actions. We analyze a Linux F-RTO implementation in various network scenarios and investigate different alternatives to the basic algorithm. The second part of this dissertation is focused on quickly adapting the TCP's transmission rate when the underlying link characteristics change suddenly. This can happen, for example, due to vertical hand-offs between GPRS and WLAN wireless technologies. We investigate the *Quick-Start* algorithm that, in collaboration with the network routers, aims to quickly probe the available bandwidth on a network path, and allow TCP's congestion control algorithms to use that information. By extensive simulations we study the different router algorithms and parameters for Quick-Start, and discuss the challenges Quick-Start faces in the current Internet. We also study the performance of Quick-Start when applied to vertical hand-offs between different wireless link technologies.

Computing Reviews (1998) Categories and Subject Descriptors:
C.2.6 COMPUTER-COMM. NETWORKS / Internetworking
C.4 PERFORMANCE OF SYSTEMS

General Terms: TCP performance, wireless links

Additional Key Words and Phrases: delay spikes, cross-layer protocol interactions

Acknowledgements

I'm in deep gratitude to a number of people that were essential for this work to happen. Missing any single one of the following people this dissertation would not have been possible.

I'm grateful to my advisor, Prof. Kimmo Raatikainen, who through the past years in a number of ways made it possible to conduct the research for this dissertation. Markku Kojo has provided huge amounts of invaluable feedback for this dissertation, and shown me the right direction starting from the early undergraduate studies. By his inspiring and positive attitude Prof. Timo Alanko has had an important role in guiding my studies at the University of Helsinki, helping to select my focus area as a student and a researcher. I also want to thank Marina Kurtén for advice on improving the English language in this dissertation.

Mika Liljeberg has educated me with a number of interesting and thoughtful discussions at Nokia Research Center about TCP/IP protocols, opening my eyes to see the relevance of problems in practical protocol engineering for small wireless devices. Mika Forssell and Ismo Kangas have made it possible to arrange time to work on this dissertation aside the other projects at NRC.

I'm thankful to Dr. Sally Floyd for the support and valuable advice I got during and after my visit in the International Computer Science Institute at Berkeley. In particular, she introduced me to Quick-Start, that became a relevant part of this dissertation. Without the collaboration with Sally Floyd and Mark Allman this part of the dissertation would not have been possible.

Too few times, and too late, have I expressed my gratitude to my mother and father who took on the difficult task of giving me a good upbringing and support.

Finally, with my deepest gratitude and love I would like to thank Jenni for the patience and matchless support throughout the years it took to finish this work.

Helsinki, May 2007
Pasi Sarolahti

Contents

CHAPTER 1

Introduction

Within the last 10–15 years Internet applications have become part of everyday life in the developed parts of the world. Applications such as the World Wide Web, E-mail, Instant Messaging, or networked games are in widespread use today. Traditionally most people have used desktop computers to access Internet applications. However, portable laptop computers with wireless Internet access have become popular as home computers due to their smaller space requirements. Moreover, recent mobile phones have advanced networking capabilities, and they have become small computer platforms in terms of processing capacity and the variety of third-party applications available for these devices. It can be foreseen that in the near future people will use their mobile terminals for different Internet applications as extensively as they do with their home computers today.

Although the Internet seems a fairly new technology for an average consumer, it has existed and evolved for more than 30 years. The end-to-end design philosophy of the Internet has proved to be robust enough to stand the huge growth of the number of Internet hosts from the 1960s until today [39]. An important design feature of the Internet are the congestion control algorithms that have protected the Internet from collapsing under the vastly increased load [81]. These principles have guided the engineering work of the TCP/IP protocol suite [152] that is used in all modern operating systems of today, as well as in the mobile terminals that provide World Wide Web or E-mail applications.

The *Transmission Control Protocol (TCP)* [133] is used by most of the networking applications, for example the World Wide Web, instant messengers, peer-to-peer file sharing and E-mail. Probably because of its popularity, TCP has also inspired a large research base from the past decades until today.

Using TCP over the wireless network access technologies is especially challenging. The wireless networks have very different characteristics compared to

the networks where the TCP was originally designed, and they evolve rapidly all the time. New wireless technologies are being announced at a constant rate, and the range of network characteristics is likely to grow in the future.

1.1 Scope of the Work

This dissertation investigates the TCP performance in networks that suffer from long or variable delays. While wireless technologies such as the *General Packet Radio Service (GRPS)* [35, 29] and *Wireless LAN* [62] are the primary topic of interest, the research approach and solutions are also applicable to other fixed or wireless networking technologies under TCP. Rather than going into details of any particular layer 2 networking technology, the focus is on the TCP protocol: we investigate the packet-level behavior of TCP, with the underlying path characteristics motivated by the above-mentioned wireless networking technologies.

We focus on the following problems related to the interaction of TCP and challenging delay characteristics:

- *Spurious retransmission timeouts caused by a delay spike*. TCP retransmission timer is adjusted dynamically based on the recently measured round-trip times. However, the TCP sender is unable to prepare for a sudden unexpected delay spike that can occur, for example, due to certain events in GPRS networks [66]. Unexpected delay spike causes TCP's retransmission timer to expire, which in turn causes a number of harmful follow-up effects sacrificing TCP performance.

- *Slow connection start-up on high-delay paths*. TCP's congestion control algorithms limit the utilization of a high-latency connection path that could consume several packets in one round-trip time. For example, the wireless link in GPRS and EGPRS has high latency, and in such environments TCP's slow-start leads to underutilization of the wireless resources.

- *Changes in path characteristics due to mobility*. Because mobility results in change of the communication path, the end-to-end path characteristics, such as bandwidth and round-trip delay can change as well. TCP is known to be slow in adapting to suddenly changing network conditions. Before the TCP sender has adjusted itself to the new path characteristics, several packets may have been lost, or the communication path may remain underutilized for a long period of time. Particularly dramatic changes can occur with vertical hand-offs between different wireless access technologies that

mobile terminals use. For example, a GPRS link has bandwidth and delay characteristics that are several orders of magnitude different than the characteristics of a Wireless LAN link.

Since all of the above-mentioned problems are typically related to the characteristics of the GPRS networks or vertical hand-offs between GPRS and Wireless LAN links, the research parametrization is motivated by these technologies. Two methods are used in research: first, parts of the research are conducted using real Linux implementation attached to an emulated network using *Seawind wireless network emulator* [100], which allows various network parameters to be controlled and configured. Second, the *ns-2 simulator* [122] is used in tests that involve a large number of network components, or require modifications in the network components that are difficult and time-consuming to implement in real network stacks.

1.2 Contributions

This dissertation has the following contributions:

- We give a detailed description of the Linux TCP implementation that has been used in many of the experimentations in this dissertation. This part of the dissertation is based on the experience gained when the author contributed modifications and protocol enhancements to the Linux TCP implementation. Although there are several books about the Linux kernel and its networking stack in general, the author is not aware of any that would give a detailed description of the TCP behavior. A conference paper written by the author and Alexey Kuznetsov, one of the key architects of the Linux TCP/IP stack, gives a detailed description of the special features in Linux TCP implementation [147]. The author has been the main contributor to the paper, receiving some comments from Alexey Kuznetsov. The author also conducted all of the performance examples given in the paper.

- We have developed a new algorithm for improving TCP performance on spurious retransmission timeouts, called *Forward RTO-Recovery (F-RTO)*. The author was the original inventor of the idea of F-RTO, and the design work was done in collaboration with Markku Kojo. The author also implemented F-RTO in the Linux kernel and ns-2 network simulator. Both implementations have been accepted and included in the main code distributions of these systems. This work is discussed in a research publication

co-authored with Markku Kojo and Kimmo Raatikainen [145]. The performance experimentations and analysis presented in the paper and this dissertation were conducted by the author. F-RTO has also been published as an RFC by the IETF [144], and it has been included in the protocol stacks of several operating systems, such as different mobile phone platforms, Linux, HP-UX, and recently also Windows Vista and the latest version of the Windows "Longhorn" server [120].

- We developed a SACK-based enhancement of F-RTO, and analyze different congestion control response variants to F-RTO. This is based on a research publication entirely written by the author [142]. The design and the analysis in this publication were also conducted entirely by the author.

- We analyze the Quick-Start mechanism for setting the initial TCP congestion window to a larger size on high-speed or high-delay paths, and thus significantly improve the TCP start-up performance. This part of the work is based on a joint research paper with Sally Floyd and Mark Allman [143]. Quick-Start was initially specified by Amit Jain and Sally Floyd, and the author has participated to the follow-up work with Floyd and Allman to develop and analyze Quick-Start further. The author has developed the Quick-Start ns-2 implementation, based on the initial code by Srikanth Sundarrajan [156], and conducted most of the simulation analysis presented in the paper. The author also participated in the Quick-Start specification work in the IETF [54].

- Quick-Start can also be used in the middle of a TCP connection, for example after a hand-off has occurred on a mobile host. We analyze the use of Quick-Start in wireless environments, especially in the context of vertical hand-offs between the GPRS and WLAN links, and show how Quick-Start can be used to improve TCP performance in these cases. This is based on a research paper written with Jouni Korhonen, Laila Daniel and Markku Kojo [146]. The author has done a significant amount of the editing work for the paper, implemented most of the applied ns-2 code, and conducted most of the simulations and analysis presented in the paper.

1.3 Related Work

Much research has been conducted on the general topic of TCP over wireless links (e.g., [16, 13, 170, 165]). Most of the research has focused on the problems of treating wireless packet losses incorrectly as congestion notifications, and the

issues related to variable delays have been under less attention. Below we list past research work that is most closely related to the work in this dissertation. More thorough discussion about the technical issues in using TCP over wireless links is carried out in Chapter 2.

Spurious Retransmission Timeouts

The problem of spurious retransmission timeouts with wireless cellular links was identified in the 1990s [101]. Possible reasons for spurious timeouts are persistent link layer ARQ retransmissions, or events that sometimes occur during cellular hand-off [66]. While the problems related to long delays in some wireless cellular networking technologies were identified a long time ago, there have been only a few earlier proposals for mechanisms to improve the TCP performance on spurious retransmission timeouts caused by sudden delay spikes. So far the most thorough discussion on this area has been presented by Ludwig et. al, who proposed and evaluated the *Eifel algorithm* [111, 112, 64], and the different TCP response variants after a detected spurious retransmission timeout [65, 110]. Ludwig has also investigated alternative TCP retransmission timeout estimators that improve performance in variable delay environments [114]. Eifel is designed to detect all types of spurious retransmissions, and in addition to retransmission caused by spurious timeout, it should also alleviate the bad effects caused by packet reordering [17]. However, Eifel requires the use of the TCP timestamp option that might not be supported in all cases, for example with the TCP/IP header compression schemes [82, 42]. F-RTO solves the same problem without using any TCP options, just by proposing a slight change in the TCP retransmission sequence. The fact that F-RTO has been adopted by a number of operating system vendors in less than a year after becoming an IETF RFC hints that this is considered a valuable difference.

Some time after Eifel and F-RTO were published, other alternatives were also proposed to improve TCP performance on spurious retransmission timeouts. *Decorrelated Loss Recovery (DCLOR)* [157] proposes an alternative SACK retransmission sequence that performs better than the standard SACK recovery algorithm after a spurious retransmission timeout. However, the price of DCLOR is that it slightly reduces the performance on timeouts that have been caused by genuine packet losses. *STODER* [159] proposes retransmitting a partial segment after a retransmission timeout and using the resulting acknowledgment to determine if the timeout was spurious. Therefore, on genuine timeouts STODER needs to send one packet in two separate fragments, which can be a small compromise to performance.

Using Explicit Communication to Find the Path Capacity

Another main theme in this dissertation is the use of explicit information in setting the TCP congestion window appropriately by using the Quick-Start algorithm. Quick-Start is an explicit mechanism for a TCP sender to query in-band the available bandwidth from the routers on the network path. If the routers support Quick-Start, the TCP sender can use the result of the query to set the TCP congestion window to a larger size than what TCP would normally use. This way the TCP sender can transmit at a higher rate and utilize the high-latency network path more efficiently. *Explicit Congestion Notification (ECN)* [136] was the first documented mechanism for using explicit information from the network to alter the congestion control state. ECN allows a TCP sender to reduce its transmission rate in response to a congestion that is reported using the ECN bits in the IP header. ECN-capable routers along the connection path can set the ECN bit when they are under congestion. *Explicit Control Protocol (XCP)* [89] is a full-fledged congestion control mechanism where the end-hosts and all the network routers cooperatively determine the correct transmission rate for a flow at a given time. XCP is based on continuous feedback about the current load of the network path being used. In comparison, Quick-Start is a quick mechanism to resolve the current path capacity, after which the normal TCP mechanisms are used for congestion control. *VCP* [163] and Anti-ECN [106] protocols show that TCP performance can be improved by using just one bit in the IP header for the routers to indicate that they are underutilized, and the transmission rate can be increased at a faster rate than normally. These mechanisms are missing the explicit information about the currently available bandwidth on the connection path. None of the related works have analyzed the mechanisms with wireless links.

In Chapter 6 we analyze different algorithms for network routers to process the incoming Quick-Start Requests and decide whether to approve the request. Measurement-based admission control research has investigated various algorithms at network nodes for admitting or rejecting flows, when given some Quality-of-Service requirements (see for example [30]). Quick-Start solves a somewhat similar problem in terms of the router algorithms for approving Quick-Start requests. However, while measurement-based admission control algorithms are designed for implementing soft Quality-of-Service based on some target parameters such as bandwidth or packet loss rate, Quick-Start is a light-weight mechanism specifically intended for resolving the appropriate sending rate for a best-effort flow on an underutilized path.

1.4 Structure of the Dissertation

Chapter 2 describes the recent evolution of different wireless networking technologies. It also gives an overview of the basics of TCP with its recent enhancements for wireless links. The problem of spurious retransmission timeouts is described in detail. The chapter also discusses different types of explicit communication mechanisms between the end-hosts and the network that have been proposed earlier. Chapter 3 describes the Linux TCP implementation with its special features that are different from the TCP standards. Performance implications of certain design choices in the Linux kernel are also shown. Chapter 4 presents and analyzes the F-RTO algorithm for improving TCP performance on spurious retransmission timeouts, and compares its performance with the Eifel algorithm. Chapter 5 presents a SACK-based enhancement for the F-RTO algorithm, and compares different congestion control responses to F-RTO. Chapter 6 presents the Quick-Start algorithm, discusses its benefits and challenges related to deployment and security, and compares different variants for router algorithms to process the Quick-Start requests. Chapter 7 proposes an enhancement to Quick-Start and analyzes use of Quick-Start on wireless hosts in the context of vertical hand-offs between Wireless LAN and EGPRS links. Finally, Chapter 8 gives concluding remarks and gives some ideas for follow-up work.

CHAPTER 2

TCP and Wireless Networks

This chapter provides background to the work established in this dissertation. First, in section 2.1, a brief introduction to the evolution of the wireless communication systems is given from the 1960's to the present day to give the reader a short overview on the versatility of different wireless communication technologies that have been designed in the past. The heterogeneity of network characteristics in these systems is the main reason for the TCP/IP performance issues that are being discussed in this dissertation. Section 2.2 discusses the main TCP retransmission and congestion control algorithms, and presents some common performance enhancements that have been proposed earlier. Sections 2.3 and 2.4 focus on the two main issues discussed in this dissertation: the spurious retransmission timeouts caused by high and unpredictable delay variability on some link technologies, and the slow convergence time of TCP/IP congestion control parameters on connection paths with high and variable delay characteristics.

2.1 Evolution of Wireless Communication Systems

One of the earlier wireless packet radio system referred to in the literature is the *ALOHA* network [2, 96] developed at the University of Hawaii in the early 1970's. ALOHA is a multiple access protocol for sharing a single satellite link that has yielded much follow-up work (e.g., [3]). The ALOHA ground stations can broadcast packets at any time to the satellite channel that is listened to by the other stations. If the packet is delivered correctly and no collision occurred on the shared channel, all ground stations (including the sender) get a correct copy of the packet. If another station was transmitting at the same time, the colliding packets are corrupted, and hence discarded. The characteristics of the channel allow the sender

to monitor whether the transmission was successful. If it was not, the sender waits for a random time and retransmits the packet.

The pure ALOHA is not an optimal protocol, because the stations can transmit at any random time, and collision of even a small portion of a packet makes it useless. Therefore *slotted ALOHA* was proposed [137], which divides the time into discrete time intervals. Each time interval is time equal to transmitting one packet. This way the partial overlapping of packet transmission from two ground stations can be avoided, and the likelihood of collision for a packet is reduced. Probability analysis shows that while the pure ALOHA can reach approximately 18 % channel utilization at its best, the slotted ALOHA can achieve about 37 % channel utilization [160, pp. 249–250]. The idea of splitting the transmission channel into discrete time intervals has been reused several times in the successive network designs, some of which are discussed below.

2.1.1 Wireless Local Area Networks

An important step in the research on wireless networking was the introduction of Wireless LAN (WaveLAN, WLAN) radios in the early 1990's [162]. These were an ideal communication system for university campuses, and with the widespread use of the TCP/IP protocols in the university systems, availability of WLAN systems started up the research trend on the behavior of TCP/IP over wireless links [45, 34, 166]. The Wireless LAN system is based on a network of WLAN base stations, typically connected with each other and the rest of the network by a fixed Ethernet cable. Each WLAN base station uses one channel in the assigned radio spectrum, forming up a cell where all nodes can detect the traffic sent by others. The media access protocol in early WLAN systems is based on *Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)* protocol [62, p. 130]. Prior to sending in a CSMA/CA system, a host transmits a *Request To Send (RTS)* message to the receiver that responds with *Clear To Send (CTS)* message if the channel is not in use and the sender is free to transmit. Because also the other hosts in the cell get these two messages, they know that the wireless channel will be allocated for transmission and know not to transmit for the allocated time period. *Multiple Access with Collision Avoidance (MACA)* [87] is an enhancement to CSMA/CA that does not perform the data carrier detection, but instead the stations include the amount of data to be transmitted in the RTS and CTS messages. This simplifies the basic CSMA/CA protocol and relieves the traditional "hidden terminal" [95] and "exposed terminal" problems [160, p. 264] in the CSMA/CD system. MACA has been further enhanced specifically for Wireless LAN systems [19], for example by improving the channel allocation mechanism for base stations.

The first IEEE 802.11 standard on Wireless LAN was published in 1997, and it was slightly revised a few times in the following years [40, 69, 76]. IEEE 802.11 supports transmission rates of 1 Mbps and 2 Mbps at 2.4 GHz frequency. The legacy 802.11 did not get deployed to a significant extent before the *IEEE 802.11b* specification was released in 1999 [77]. The 802.11b stations can have a transmission rate of 11 Mbps, though by changing the channel encoding they can also transmit at 5.5 Mbps, 2 Mbps, and 1 Mbps when the link conditions are not good enough for transmitting at the higher rates, for example because of the distance between the mobile terminal and the base station. It is worthwhile to note that despite the given theoretical transmission rates, the actual throughput for the upper layer protocols is often lower due to the use of CSMA/CA that uses an additional wireless round-trip time to avoid collisions of the data frames. While the 802.11b deployment started in university and company campuses, it is nowadays in widespread use in various public locations and homes.

Recently the IEEE 802.11 family has been extended with two standards capable of transmitting at 54 Mbps. *802.11a* uses 5 GHz frequency band [78], whereas *802.11g* is placed on the usual 2.4 GHz frequency band using advanced channel encoding mechanisms to gain the higher transmission rate [75]. The drawback of 802.11g is that the 2.4 GHz frequency band is used by numerous WLAN-capable laptops and other devices, and the communication can suffer from interference in crowded locations. The advantage of 802.11g is that the lower frequency helps in providing somewhat larger coverage areas and better penetration of solid objects. There is work ongoing in the IEEE on new standards providing an even higher transmission rate for WLANs [168]. There is also work ongoing to enhance the WLAN service capabilities in other ways, for example with the upcoming 802.21 standard, which is intended to provide information and triggers from the wireless network for the upper layer protocols to be used for better control in modern heterogeneous networks [43].

2.1.2 Wireless Wide Area Networks

Roughly at the same time as the research on TCP/IP protocols over wireless LANs, research on using TCP/IP over cellular phones and other wireless wide area networks (WWANs) began. We will skip the work on using analog circuit-switched technologies such as the retired *Nordic Mobile Telephone (NMT)* system [108], although some research on TCP/IP over these systems was conducted in the early 1990s [4], and focus on the more substantial research that started with the introduction of digital cellular wireless technologies. We next discuss the wireless wide area network technologies that are going to be referred to in the rest of this dissertation. There are some technologies that are not described, since

they are considered less significant to an average (European) mobile terminal user, such as PDC [70], IS-95, or CDMA2000 [97].

GSM (Global System for Mobile communications) [135] is the most widely deployed system for digital wireless wide-area networks, taken into use in the early 1990s. GSM uses up to 124 frequency channels in each wireless cell, each channel split into 26 time division multiplexed (TDM) time frames. Transmission of a TDM frame takes 4.615 milliseconds and it is shared between eight users that each have a dedicated time slot in the TDM frame. One GSM data user can have a data transmission rate of 9600 bps. Since GSM is a circuit-switched system, the transmitting host needs to establish a dial-up connection with a modem to transmit TCP/IP data over the GSM channel. Later, *High-Speed Circuit Switched Data (HSCSD)* (see brief description for example in [104]) was introduced to enhance the GSM radio speeds by supporting a better channel encoding that is capable of transmitting at 14.4 Kbps in a single GSM time slot, and with the possibility of using up to four time slots for a single GSM connection. Furthermore, HSCSD is able to choose between the 9.6 Kbps and 14.4 Kbps encodings depending on the quality of the wireless link. As a result, transmission speeds of up to 57.6 Kbps can be achieved in the GSM data transmission.

Figure 2.1 shows the main components of a GSM and GPRS systems. A *Mobile Station (MS)* communicates over the radio link with *Base Transceiver Station (BTS)* that communicates with the mobile stations in its coverage area. *Base Station Controller (BSC)* controls the base stations, for example, by allocating the radio frequencies and controlling the hand-offs from one BTS to another in cases where both of the BTS nodes are controlled by the same BSC. *Mobile Switching Center (MSC)* connects the GSM subsystem to the rest of the *Public Switched Telephone Network (PSTN)* and handles various tasks related to call control, roaming, and so on. For TCP/IP traffic a *modem pool* is needed to convert the circuit switched traffic into IP packets that are sent to the Internet. There are also some other components in a GSM system that are not shown in the figure, such as *Home Location Register (HLR)*, because they are not considered relevant for the scope of this work, analyzing TCP/IP communication performance.

In the late 1990s the GSM system was extended by the *General Packet Radio Service (GPRS)* [29, 35] that adds packet-switching capabilities to the existing GSM architecture. GPRS uses a common radio access system with the circuit-switched GSM, and can co-exist with the circuit-switched GSM and HSCSD systems. Similarly to HSCSD, a GPRS terminal can use one to four TDM slots for data transfer in one direction. However, being a packet data service, a GPRS terminal allocates the wireless channel only for the time it has packets to transmit. Therefore GPRS can be expected to achieve better channel utilization than

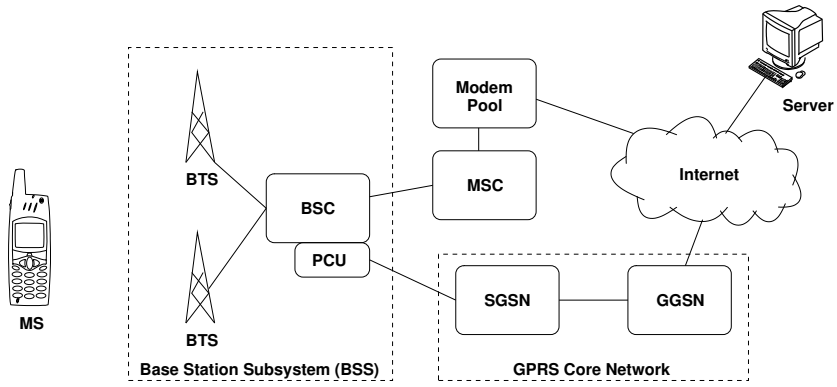


Figure 2.1: The main components of a GSM/GPRS system.

its circuit-switched predecessors. GPRS also has four different channel encoding classes for transmitting at 9.05 Kbps, 12.0 Kbps, 14.4 Kbps, or at 20.0 Kbps in a single time slot. The closer the wireless terminal is to the base station, and the better the radio link quality is, the higher an encoding class can be used in the radio communication. With these settings, a GPRS user can expect data transmission rates of 30 - 80 Kbps.

The more substantial changes to the GSM system brought by GPRS are in the core network. The packet-switched GPRS core network uses different components than the traditional circuit-switched side. The packets from the Internet first arrive at *Gateway GPRS Support Node (GGSN)* at the mobile user's home network that encapsulates the arriving packets using the GTP tunneling protocol. The packets are tunneled to *Serving GPRS Support Node (SGSN)* at the network where the mobile terminal is currently located. SGSN detunnels the packets arriving from GGSN and sends them to the mobile station via the *Packet Control Unit (PCU)* that converts the data from the wireless link into packetized traffic. The main difference between the circuit-switched GSM and GPRS systems is that in GSM the data goes in circuit-switched connection all the way to the modem pool in the fixed network, whereas in GPRS the data traffic goes in IP packets through the GPRS core network.

A few characteristics of the GPRS system specifications have triggered fruitful research issues on TCP/IP performance, some of which have been described in [66]. First, the GPRS mobile station needs to allocate a *Temporal Block Flow (TBF)* state with the GPRS *Base Station Controller (BSC)* using an ALOHA style random access channel. To allocate the channel resources, a mobile station first has to wait for the control message channel to become idle, and then send a Packet

Channel Request message to the BSC. The BSC responds by indicating the allocated channel and number of time slots allocated for the mobile. The allocation of the TBF adds a delay of more than 100 ms to the uplink data transmission over an idle channel [66]. Furthermore, the early GPRS specifications require release of the TBF state immediately after the data buffers are emptied. As a result, with data patterns that occasionally send small pieces of data the round-trip times are generally higher than for bulk data transfer. The second characteristic of GPRS data transmission is occasional delay spikes in data transfer. Usually these are caused by GPRS cell reselection: after a mobile station moves and makes a decision to use a new cell, it needs to perform the channel allocation procedures on the new BSC as described above. After the TBF is established with the new BSC, the mobile station needs to inform the current SGSN of the change, which then tells the old BSC to release the resources allocated for the mobile host. It has been reported that cell reselections suspend the data transmission for 3 to 15 seconds [66]. Cell reselections can cause either delay spikes in the data transfer, or loss of several packets, or both, depending on the direction of the data transfer. The cell reselection performance has been improved in the later versions of GPRS specifications, particularly in Enhanced GPRS discussed below.

In the beginning of the 2000s the GPRS system was enhanced with new features, one of them being a more efficient *Eight Phase Shift keying (8PSK)* channel modulation scheme that triples the transmission rates available in a GPRS system. The enhanced GPRS system is called EGPRS or *EDGE (Enhanced Data rates for GSM Evolution)* [150, 155]. The data propagation delays on the radio link are similar to the traditional GPRS system, but the maximum transmission rate increases to 384 Kbps, if all eight time slots are in use for transmission in one direction. In practice, with four downlink time slots, the maximum transmission rate is 236.8 Kbps.

Along with the EDGE specification, third-generation cellular standards have been specified by two standardization bodies, 3GPP and 3GPP2. The third-generation system specified by 3GPP is called *Universal Mobile Telecommunications System (UMTS)* [86]. It is based on the use of *Wideband Code Division Multiplexing (WCDMA)* [73], which is capable of a maximum of 2 Mbps data transfer rate. The radio link propagation delays are also lower than in the GPRS-based radio links. Recently WCDMA systems have been further enhanced with the *High-Speed Downlink Packet Access (HSDPA)* technology [109]. HSDPA can improve the downlink data transfer rates by a factor of five for some traffic patterns by utilizing technologies such as Adaptive Modulation and Coding, fast scheduling and Hybrid Automatic Repeat Request at the Node B, the base station in the UMTS system.

2.1.3 Interaction of different wireless technologies

The MosquitoNet was among the first projects to investigate mobility between different network access technologies [12]. Utilizing the capability of using multiple network interfaces, MosquitoNet was able to achieve seamless hand-offs. *Wireless overlay network* was introduced to refer to a network with heterogeneous hierarchy of different wireless access technologies with varying coverage ranges and characteristics. The term *vertical hand-off* was used in this context already in 1998 [151].

Many of the current handheld terminals support both WWAN and WLAN technologies. Typically a terminal has a GPRS and possibly WCDMA connectivity, and a 802.11-based wireless LAN radio, that are largely independent with separate radio hardware, and can be used in parallel. Because the coverage ranges of WCDMA and WLAN technologies are smaller than in GSM/GPRS radios, GPRS is still the main technology used for data communication in rural areas, but in urban areas it is possible that either WCDMA or WLAN access is available. As a result, the range of transmission speed and delays observed by the user is large, varying from a few tens of kilobits per second to a few tens of megabits per second. Furthermore, a TCP sender adjusts some of its parameters based on the measured performance in the recent past. The substantial variance in possible wireless link characteristics therefore imposes a great challenge to the TCP performance.

In today's mobile terminals GPRS and WCDMA radios typically share the same layer two control functionality, and usually have the same Internet access provider. Therefore the access provider can have a strong role in determining which technology is being used at a given time. In addition, the IP layer typically sees these technologies as a single logical access interface. However, the wireless LAN access can be offered by a different provider, and it often uses a different IP address in a separate network interface of the TCP/IP stack. Therefore IP mobility technologies such as Mobile IP [129, 85] are needed for hand-offs between these interfaces. Although we discuss the different IP mobility mechanisms more in Chapter 7, we do not go into details of the IP mobility mechanisms, but focus on TCP algorithms and performance.

2.2 Transmission Control Protocol (TCP)

The Internet that has become a considerable part of life for people in developed countries is a result of a development process that started in the late 1960s. While the network has grown rapidly, and the number of hosts attached to the Internet is orders of magnitude larger than it was a couple of decades ago, it has remained

operational with reasonably small modifications to the traditional base protocols. It could be argued that the successful growth would not have been possible without following certain well-established design principles when defining the TCP/IP protocols [39]. This section summarizes the evolution of TCP in the past 30 years and presents some of the performance enhancements proposed to TCP for wireless networking. Some of the performance enhancements follow the original design principles better than the others.

2.2.1 Evolution of TCP

A paper published in 1974 by Vint Cerf and Robert Kahn presented the *Transmission Control Program*[1] *(TCP)* [37]. The paper introduced several concepts that are still important today, although many of the details have changed. TCP was designed to be a protocol by which two hosts in different subnetworks can have reliable data transfer. The hosts were addressed by identifiers that indicate the network in which the host is located, as well as the unique identifier within that network. Networks are connected to each other by *Gateways*. TCP used source and destination port numbers by which several data flows could be multiplexed to the network, and sequence numbers were used for reliable, ordered delivery of data packets. TCP also had a retransmission mechanism based on the use of positive acknowledgments and a *retransmission timer*, and a window-based flow control mechanism to aid reliable transfer. These basic concepts are still in use today, although some parts of the protocol design have evolved over time.

The Internet protocols, including TCP, are specified in *Request For Comments (RFC)* documents. The first RFC was written in 1969 as part of the ARPANET project, and as of August 2006, there are 4600 RFCs. The Transport Control Protocol specification, RFC 793 [133] was written by Jon Postel in 1981, based on the design in the 1974 paper by Vint Cerf and Robert Kahn. RFC 793 is still in effect, and it defines the baseline TCP protocol, although several RFCs have been published since then to update some parts of TCP. At the same time, specifications of a couple of other important core Internet protocols were released, namely the *Internet Protocol (IP)* [132], and the *Internet Control Message Protocol (ICMP)* [131], are still in use today, although in somewhat evolved form. The User Datagram Protocol (UDP) [130], another important protocol in the Internet, saw birth as an RFC already earlier, in 1980.

The *Internet Engineering Task Force (IETF)* was founded in 1986 to organize the specification of the Internet protocols and to coordinate the authoring of

[1] Yes, the original paper uses *Program*, not *Protocol*.

RFCs. Among several other work topics, the IETF has taken care of maintaining the TCP protocol, and proposed many new enhancements to TCP. One of these is the F-RTO algorithm based on the research conducted for this doctoral dissertation [144].

As the number of ARPANET network hosts and the amounts of transmitted data expanded, some of the network paths were not capable to keep up with the amount of traffic, resulting in periods of collapsing transmission performance in 1986 in transmission between two sites that were geographically just a few hundred meters apart, the Lawrence Berkeley National Laboratory and the University of California at Berkeley campus. This inspired the well-known congestion control paper by Van Jacobson [81] that has become the determining factor in the TCP performance research. Van Jacobson proposed that a TCP sender should employ a congestion control algorithm, starting up its transmission at a slow rate, and then gradually increasing its transmission rate as the acknowledgments of the packets arrive. Because congested routers drop packets that they cannot receive due to lack of buffer space, it was proposed that a packet loss is taken as a sign of network congestion, and the sender should reduce its transmission rate in response to congestion. The way by which the arrival of acknowledgments triggers transmission of new packets is called *acknowledgment (ACK) clocking*, which is one of the important principles to guarantee the network stability today. A new variable to the TCP sender's connection control block, the *congestion window (cwnd)*, was introduced to determine TCP's sending rate, i.e., how many TCP packets are allowed to be outstanding in the network by a TCP connection. The congestion window is maintained separately for each TCP connection between a client application and a server, separately for both directions of data communication.

TCP's congestion window is adjusted in two phases: in *slow-start*, the congestion window and TCP's transmission rate are roughly doubled each round-trip time[2]. The congestion window is initialized to allow transmission of 1–4 segments[3], depending on the maximum segment size. Each incoming acknowledgment for a successfully transmitted TCP segment increases the congestion window by the size of one full-sized segment. TCP sender also maintains a *slow-start threshold (ssthresh)* that determines when execution of the slow-start algorithm is finished, and when execution of the *congestion avoidance* algorithm is started. The slow-start threshold is usually initialized to an arbitrarily large value, and it is decreased at the same time with the congestion window when a packet loss

[2]If delayed acknowledgments [26, 11] are in use, the sending rate is increased roughly by 50 % each round-trip time. Also the use of *Appropriate Byte Counting* [6] affects the rate at which the congestion window is increased.

[3]The TCP transmission unit carried in one IP packet is called *segment*.

occurs, to a value corresponding with half the amount of outstanding data at the time the packet loss was detected at the sender. The current IETF specification for TCP's congestion control is RFC 2581 [11]. Congestion avoidance is applied when the congestion window size is larger than the current slow-start threshold. In congestion avoidance TCP's congestion window is increased by the size of one full-sized segment once in a round-trip time. The congestion window is decreased to half of its previous size when a TCP sender detects congestion. This class of congestion control algorithms are generally called *Additive Increase, Multiplicative Decrease (AIMD)* congestion control algorithms in the research literature, and several alternative variants of AIMD congestion control have been proposed to be used with TCP in the past [55, 28, 164, 36]. A widely used analytical model of TCP throughput with the basic congestion control algorithm is given in [123].

Figure 2.2 is a traditional illustration[4] of the behavior of the TCP congestion control algorithms. The figure shows that in the beginning the TCP sender doubles the congestion window size each round-trip time, until the congestion window size is larger than the slow-start threshold (ssthresh). From that point on the congestion window increases by one segment size each round-trip time. When a congestion notification, for example a packet loss, arrives at the sender, it reduces the slow-start threshold and congestion window to half the size of the current window, and continues transmitting in congestion avoidance at the reduced transmission rate.

As TCP uses only positive cumulative acknowledgments, the only loss signal available in the early TCP implementations was the retransmission timeout. Because waiting for the expiry of the conservatively maintained retransmission timer is a rather inefficient way of recovering from a loss of a single packet in most cases, an improved algorithm called *fast retransmit* [81, 26] was proposed at the same time with slow-start and other congestion control algorithms in Van Jacobson's congestion control paper. Fast retransmit makes use of the fact that the TCP receiver immediately sends a duplicate acknowledgment, i.e., an acknowledgment for the same segment as previously, when it receives an out-of-order segment. Because a likely reason for receiving out-of-order segments is a loss of one or more earlier packets, a duplicate acknowledgment can be taken as a loss signal. However, because it is possible that packets are re-ordered in the network, the sender waits for three consecutive duplicate acknowledgments before retransmitting the first unacknowledged segment to be more robust against unnecessary retransmissions. When using the fast retransmit algorithm, the TCP sender is able to maintain a steady flow of packets to the network, preserving the ACK

[4] A similar figure is used widely in course literature on TCP/IP networking, for example in [160, p. 539].

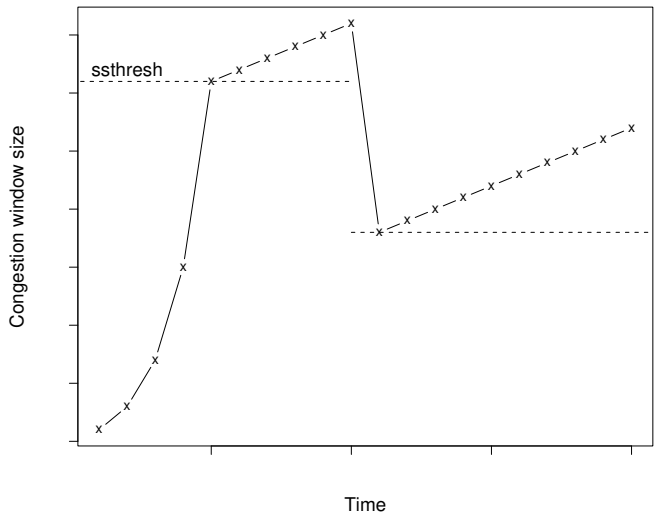


Figure 2.2: Illustration of TCP congestion window behavior.

clocking of outgoing packets, and usually retransmit packets quicker than with the timer-based retransmissions, that often cause a small pause in TCP transmission. In addition, when a retransmission timeout occurs, the TCP sender sets its congestion window size to one segment, while after the fast retransmit the congestion window is set to half of its earlier size (as shown in Figure 2.2). If also the retransmitted packet is lost in the network, the retransmission timeout length is doubled, and a new retransmission is made. The exponential back-off of the retransmission timer continues until the retransmitted packet is acknowledged, or when a user timeout expires after a few minutes, and the connection is aborted.

The TCP sender tries to estimate a typical packet round-trip time (RTT), and use it to determine an appropriate retransmission timeout (RTO) length. When an acknowledgment to a segment arrives at the TCP sender, the IETF specification require that the TCP sender adjusts the RTO estimate as follows [127]:

RTTVAR <- $(1-\beta) * RTTVAR + \beta * |SRTT - R|$
SRTT <- $(1-\alpha) * SRTT + \alpha * R$
RTO <- $max(SRTT + 4 * RTTVAR, 1s.)$

where R is the measured round-trip time for the acknowledged segment, RTTVAR is variation of the recent round-trip times, and SRTT is the smoothed

mean round-trip time based on the recent measurements. α and β are constants with recommended values of $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$.

Later the fast retransmission algorithm was enhanced by *fast recovery*, first implemented in 4.3 BSD Reno in 1990 [83, 11, 51]. Fast recovery follows fast retransmit and lasts until the retransmitted segment is acknowledged. During fast recovery the TCP sender preserves the number of outstanding segments after the congestion window has been reduced to half of its earlier size by sending new segments to the network as acknowledgements come in. This is done after the number of outstanding segments has decreased to match the reduced congestion window size. The TCP congestion control specification temporarily increases the congestion window for each incoming duplicate ACK to allow this forward transmission of a segment, and deflates it back to the original value at the beginning of the fast recovery when the fast recovery is over.

There are two variants of fast recovery: the original one described above, and the *NewReno* algorithm [71, 56][5]. The standard variant exits the fast recovery algorithm when the first acknowledgment that advances the window arrives at the sender. If there is more than one segment dropped in the same window, the standard fast retransmit does not perform efficiently, because the rest of the dropped segments can only be retransmitted after a retransmission timeout that can take a relatively long time to expire. NewReno TCP exits the fast recovery only after all segments in the last window following the segment that triggered fast retransmit have been successfully acknowledged.

TCP acknowledgments indicate the next segment the receiver expects to receive in the byte stream. A basic TCP receiver is not able to indicate lost segments by other means than using the duplicate acknowledgment method, which indicates that at least one of the segments is missing. Therefore, after retransmitting the first unacknowledged segment by fast retransmit, the TCP sender needs to await the acknowledgment for the retransmitted segment in order to know if other packets were lost in the window of data that was outstanding when the sender detected the first packet loss. In practice, a basic TCP implementation can recover from data loss at a rate of at most one segment in a round-trip time because of the minimal information in the acknowledgments.

A significant improvement to the TCP loss recovery performance was achieved with the *Selective Acknowledgment (SACK)* TCP option [117, 51]. With the SACK option, a receiver can acknowledge up to four non-sequential blocks of

[5]Recently a Standards Track version of the NewReno algorithm has been published by the IETF [57]. However, since most of the analysis in this dissertation was conducted before the Standards Track document was published, we mostly refer to the earlier experimental version of the NewReno specification. In practice the differences between the two versions are minimal.

data received beyond the first missing byte. With the extra information, the TCP sender can employ retransmission algorithms that are able to retransmit more than one segment in a round-trip time, thus allowing faster TCP recovery. Different TCP sender implementations have applied slightly different recovery algorithms in response to incoming SACK information. Two of the best known ones are the *Forward Acknowledgment (FACK)* algorithm [116], and the IETF-standardized conservative recovery algorithm [23]. The main difference between these two algorithms is in how quickly the TCP sender decides a packet loss has occurred instead of waiting for a delayed packet to arrive. The FACK algorithm assumes that all segments transmitted before the most recently acknowledged segment in a SACK option have either reached the receiver or been lost. In the conservative algorithm the receiver assumes a segment is lost only after there is a gap of more than three unacknowledged segments between the selective acknowledgments, thus aiming to preserve the robustness of the fast retransmit algorithm against possible packet reordering. The practical difference is that the FACK algorithm recovers slightly faster in lossy situations, but is less robust against unnecessary retransmissions due to packet reordering.

Recently the use of SACK information has been extended to report duplicate segments that arrive at the receiver by a mechanism called *DSACK* [61, 21, 20]. Because segments that arrive in the wrong order at the receiver generate duplicate acknowledgments, it is possible that the sender unnecessarily starts retransmissions, despite requiring three consecutive duplicate ACKs before starting the retransmissions. It has been reported that such packet re-ordering occurs in the Internet [17]. A DSACK receiver generates SACK acknowledgments also for incoming packets that have already been acknowledged by TCP's cumulative acknowledgment. Use of DSACK allows the sender to act appropriately on segments that are either duplicated at the network, or have been unnecessarily retransmitted by the sender, and undo the apparent false congestion control response made due to receiving three consecutive duplicate acknowledgments that have falsely indicated a packet loss.

The *TCP Timestamp option* [25] was suggested to allow more accurate round-trip time measurements for some implementations with coarse-grained round-trip time measurement algorithms, especially on network paths with high bandwidth-delay product. A 32-bit timestamp is attached to each TCP segment transmitted by the sender, which is then echoed back in the acknowledgment for the segment. From the echoed timestamp the TCP sender can measure exact round-trip times for the segments and use the measurement for deriving the retransmission timeout estimator. Particular benefits of the TCP Timestamp option are that it allows measuring round-trip times from TCP retransmissions, which is not possible without it

Table 2.1: TCP congestion control related IETF specifications.

RFC	Description
RFC 793	TCP base specification
RFC 1122	Requirements for hosts
RFC 1323	Performance extensions
RFC 2018	SACK
RFC 2581	Congestion control
RFC 2883	DSACK
RFC 2988	Retransmission timer
RFC 3042	Limited transmit
RFC 3168	Explicit Congestion Notification
RFC 3390	Increasing initial window
RFC 3517	SACK Recovery Algorithm
RFC 3782	NewReno

due to a problem called *retransmission ambiguity* [88]; and that it allows protecting against wrapped sequence numbers on paths with very high-bandwidth delay product.

To give TCP senders additional means of detecting congestion, *Explicit Congestion Notification (ECN)* [136] was suggested for routers to explicitly mark packets when they arrive to a congested point in the network. When a TCP sender receives an echoed ECN notification from the receiver, it reduces its transmission rate in the same way as it does when responding to a packet loss. ECN allows the TCP senders to reduce the transmission rate in response to congestion without having to suffer from packet losses. Explicit cross-layer communication mechanisms are discussed more in Section 2.4.

As discussed above, the TCP algorithms are specified in a number of different RFCs, which can make it difficult to analyze and implement a state-of-the-art TCP behavior. Table 2.1 summarizes the most important Standards Track RFCs that affect the TCP performance. In addition, there are a number of experimental and informational RFCs related to TCP. The IETF has published a TCP roadmap that shortly describes all the current TCP-related RFCs [46].

2.2.2 Enhancements for Wireless Links

The emerging of WLAN networks and IP mobility support [80] inspired various research activities on improving TCP's performance over wireless links. The key problem in the early research was the TCP congestion control taking packet losses

as an indication of congestion, which is a false assumption when a packet loss is caused by data corruption in wireless transfer [33, 169].

Many of the early solutions relied on having an active component at the wireless base station or wireless access router. A few solutions, such as *Indirect TCP (I-TCP)* [13] or the *Mowgli* communication architecture [101] split the TCP connection into two separate connections at the wireless access router, using a regular TCP connection between the fixed server and the wireless access router, and another TCP connection or a protocol specifically tailored for wireless links in the communication between the wireless access router and the mobile host. With the split connection approach, the sender at the fixed Internet can speed up the startup of the connection during slow-start because of the shorter round-trip times of packets that are being acknowledged at the wireless access router. Split connection also helps to avoid TCP congestion control response due to wireless packet losses, because the wireless access router hides the packet losses on the wireless link from the fixed server. A separate protocol aware of the wireless link characteristics can be used to recover efficiently from wireless packet losses.

The *Snoop* approach [16] also uses an intermediate component at the wireless access router, but it does not split the TCP connection. The Snoop module monitors the TCP duplicate acknowledgments and uses the acknowledgements or a local timer to determine if a packet is lost on the wireless link. If the Snoop module determines that a packet is lost on the wireless link, it retransmits the packet locally and hides the packet loss from the fixed end sender by not forwarding the duplicate acknowledgments to it. This way the Snoop module aims to make quick retransmissions of the lost data on the wireless link, and avoid the congestion control actions at the fixed sender due to data loss on the wireless link.

The main disadvantage of the split connection approach and many of the other types of transport-layer proxies is that they violate the end-to-end principle that is one of the key design principles of the Internet protocols [140, 49]. A wireless access router may falsely acknowledge packets even if the wireless host has disconnected from the network for some reason, or has had a software failure that has resulted in loss of the TCP state. In addition, solutions that require access to the transport protocol headers in the middle of the connection path cannot be used with IPsec [93]. Because of these reasons, splitting the connection or using other types of performance enhancing proxies are generally discouraged. The IETF has given recommendations regarding hosts that either split the connection or have some other TCP-aware heuristics near the wireless access point [24].

The IETF has made recommendations on TCP behavior over 2.5G and 3G links [79]. The document discusses the appropriate window sizes for TCP in these environments, and proposes to use certain TCP enhancements, such as the

selective acknowledgments and limited transmit described earlier. Furthermore, the document recommends to use TCP timestamps to improve TCP performance. This dissertation challenges some of the recommendations in the document, for example by showing that some of the benefits of using TCP Timestamps can also be acquired without using them.

Wireless hosts may also suffer from longer periods of disconnection. If such a disconnection takes more than a few minutes, the TCP user timeout expires and the connection is aborted. Because some applications would benefit from the connection state being maintained over longer periods of disconnection, a new TCP option has been proposed to extend the user timeout length on per-connection basis [149]. When connectivity is regained, the TCP sender is triggered to make a quick retransmission, as the next retransmission timeout could take a long time due to a backed-off retransmission timer.

2.3 Problems with TCP's Retransmission Timer

When using TCP in GPRS networks, new kinds of problems emerged. Unlike in the traditional research on TCP over wireless, TCP is not usually affected by packet losses due to data corruption on the wireless link, because the lower protocol layers provide reliable delivery service for upper-layer protocol data. Instead, the additional delays due to the actions related to channel allocation as explained in Section 2.1 can cause the TCP retransmission timeout to expire [119]. Because it is possible that no data has been lost, the TCP retransmission timeout is spurious, followed by unnecessary slow-start retransmissions by the TCP sender. The spurious retransmission timeout also violates the packet conservation rule. The packet conservation rule requires that the number of outstanding segments are maintained at a steady level, apart from the adjustments made to the congestion window. However, after a spurious timeout the TCP sender makes two useless slow-start retransmissions for each packet that leaves the network. In addition, the TCP's congestion window is reset unnecessarily after the retransmission timeout, which further damages the TCP performance.

Figure 2.3 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, spuriously triggering the RTO recovery and unnecessary retransmission of all unacknowledged segments. This happens because after the delay the ACKs for the original segments arrive at the sender one at a time but too late, because the TCP sender has already entered the RTO recovery. Therefore, each of the ACKs trigger the retransmission of segments for which the original ACKs will arrive after a while. 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, which causes further reduction of the congestion window.

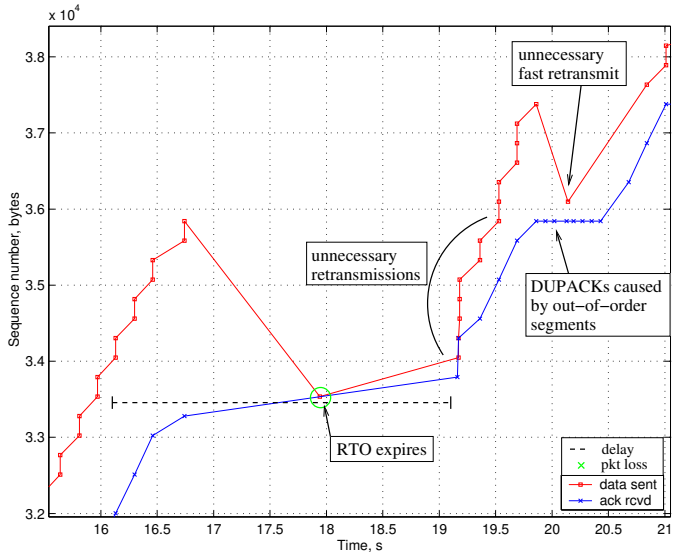


Figure 2.3: A delay triggers spurious retransmission.

The possible solutions for improving TCP performance after spurious RTO can roughly be divided into 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 [127] have been studied [10]. In addition, totally new algorithms for setting the RTO timer have been suggested (e.g. [114]). 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. Chapter 4 presents an algorithm for improving the TCP's performance in the face of spurious retransmission timeouts, called *Forward RTO Recovery (F-RTO)* [145].

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* [111] 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 continues 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 [25] 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[6], but in order to take advantage of Eifel, the 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 TCP/IP header compression specifications [82, 42]. The main difference between Eifel and the F-RTO algorithm is that F-RTO does not require additional TCP options, but it works with basic TCP, just by slightly modifying the sender's TCP retransmission algorithm.

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 (DSACK)* enhancement [61] 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 [5], 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 DSACK 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 DSACK. Instead, the suggested recovery algorithm using DSACK can only revert the congestion control parameters

[6] 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 [5].

to the state preceding the spurious retransmission [20]. Both ends of the TCP connection need to be aware of the DSACK extension in order to take advantage of it.

Recently other algorithms to avoid harmful effects of spurious retransmission timeouts have also been proposed. Like F-RTO, *STODER* [159] does not require any TCP options but it just applies a small modification at the TCP sender. The idea of STODER is to split the retransmitted segment into two smaller pieces after an RTO, and retransmit just the first, smaller piece. By this trick, the sender can separate the acknowledgment of the original segment and the acknowledgment of the retransmission, and is able to take the appropriate actions on spurious retransmission. A slight problem with STODER is that on genuine timeouts the sender needs to transmit one segment more than normally, which involves a bit more packet overhead in the network, and takes one more round-trip time to recover.

The *Correlated Loss Recovery (DCLOR) algorithm* [157] is an alternative TCP retransmission algorithm based on the use of TCP SACK acknowledgments in determining whether the sender is required to retransmit after a retransmission timeout, or whether it can send new data. Immediately after a retransmission timeout the sender transmits new data, and if the incoming SACK blocks indicate that the recently transmitted segment arrived before the original, earlier transmitted segments, it is likely that the retransmission timeout was not caused by a delay spike, but is a genuine RTO. A slight drawback of the DCLOR algorithm is that it follows RTO by immediately transmitting new data, which delays the recovery by one round-trip time on genuine timeouts.

The IETF has organized the work on spurious retransmission timeouts into *detection algorithms* and *response algorithms*. A detection algorithm is used to determine whether an RTO is genuine or spurious, and the response algorithm is activated if an RTO is determined to be spurious. A response algorithm defines how congestion control parameters, the congestion window and slow-start threshold size, are adjusted after a spurious timeout, value of the retransmission timer, and the sequence of segments to transmit. Currently there are RFCs for DSACK [21], Eifel detection algorithm [112] and F-RTO [144], that is modeled as a detection algorithm, and two active documents for alternative response algorithms, namely the Eifel response algorithm [110] and DCLOR [157]. Chapter 5 discusses the response algorithms in more detail.

2.4 Enhancing TCP with Explicit Cross-Layer Communication

Section 2.2 discussed different approaches to improve TCP performance. Some of the mechanisms were based on placing an intelligent proxy node on the connection path that can hinder the negative effects of the wireless link from the fixed end of the TCP connection. Other mechanisms modify the TCP algorithms at the end hosts to perform better on a challenging network environment. We now take a look at another kind of approach to improve the transmission performance at the end hosts: giving tools for the network to tell more about its characteristics to the end hosts.

The normal congestion control mechanisms of TCP usually work well when the network environment is reasonably stable and the end-to-end delay is reasonably short so that the congestion control parameters can be adjusted timely. However, when the delay of an end-to-end path is long, the delay in getting TCP feedback starts to affect the communication performance: first, in TCP slow-start it takes several round-trip-times to increase the congestion window size to be large enough to efficiently utilize the capacity of the end-to-end path. Second, propagation of the loss or congestion notifications takes time, during which the TCP continues increasing its sending rate [41], even though it should reduce it. This has a substantial effect especially in slow-start, when TCP doubles the congestion window size during the round-trip it takes to get the loss notification back to the sender.

Another problematic environment for TCP is that where the end-to-end path can have sudden significant changes in its characteristics. This can happen, for example, due to mobility, especially in vertical hand-offs [115] where two network access technologies are drastically different. A common real-world example is hand-offs between a GRPS access link, which can provide bandwidths of about tens of kilobits per second, and a WLAN link, which can provide bandwidths of about several megabits per second. However, TCP adapts its RTO estimate and congestion control parameters to the changed path characteristics very slowly, because the congestion window can only be reduced by half after a packet loss[7], and in congestion avoidance the congestion window can be increased only by one segment in a round-trip time. When the sudden changes to the path characteristics are measured on the orders of magnitude, it is apparent that the TCP adaption is distressingly slow in these situations.

[7] Or, explicit congestion notification, if that happens to be supported

With a little additional intelligence in the network, or at the different protocol layers of a mobile end host, it is possible to deliver notifications to the TCP sender about the changed path characteristics, to allow it to adapt more timely to the characteristics of a new path. The notification message could contain information about the new path characteristics, or if reliable evaluation of the new characteristics is not possible, the sender could re-initialize its congestion control state and round-trip time measurements on getting the notification.

2.4.1 Classification of explicit cross-layer mechanisms

Based on the past work on explicit notification and communication mechanisms, we propose a taxonomy of the mechanisms between the end hosts and the network can be identified [105]. Signaling or notification mechanisms can be split into in-band and out-of-band mechanisms, based on whether the information is piggybacked along with the transport protocol traffic, or whether the signaling is done by the means of separate control packets, respectively.

The benefit of using in-band signaling is that the signaling can be better assumed to take the same network path as the protocol data. Out-of-band mechanisms could take a different path due to different policy actions: an IPsec policy might not aggregate the signaling protocol to the same security association as the data protocol, or a policy-based routing system could select a different path for the out-of-band signaling than for the protocol data. Sometimes a packet with unrecognized content can cause the whole IP packet to be dropped in the network due to NAT or firewall policy, or because of a defective router. When the message is transferred in-band, the loss notification usually comes naturally with the protocol's own acknowledgment mechanisms. For out-of-band mechanism there might not be any direct mechanisms to inform about the loss. The drawback of an in-band mechanism is that a loss due to additional packet content also hurts the data transfer. Out-of-band messages can also be more susceptible to security problems caused by a third party generating malicious messages.

The following list discusses three types of in-band notification mechanisms that have been proposed in the past, and two types of out-of-band signaling mechanisms.

- **In-band message processed by end hosts.** When a message is attached to the transport protocol header, only the communication end hosts can be assumed to see the message. IPv6 also has extension headers that are only processed by the end hosts. The routers along the network path are not typically capable of processing this kind of message, and if the packet is encrypted with IPsec, it is impossible for other nodes than the end hosts to

read the message. The benefit of using transport header is that it can be expected that the legacy routers and different flavor of network middle boxes are not likely to take unexpected actions on the packet, such as dropping a packet with an unknown option. An example of this kind of notification type is LMDR [158] that uses a TCP option to allow a mobile end host to notify the other end that it has moved.

- **In-band message processed by some routers.** If a message uses some of the reserved bits in the IP header, or is an IP hop-by-hop option, routers along the network path are able to process it and take appropriate actions. There can be two types of messages: those that are only read by a router, and those that can also be altered by the router. The options that are to be altered by the router should not be covered by IPsec authentication [92]. In case of IPv4 this means that such an option should be explicitly marked as a mutable field for IPsec. An IPv6 option includes a bit that tells IPsec whether the option is mutable or non-mutable. IPsec does not cover the reserved bits in the IP header, either. The problem with the use of IP options is that the network is known to drop the majority of packets with unknown IP options [118]. Some explicit notification types are such that they are of benefit even if a single router along the network path supports them. Explicit Congestion Notification [136] is one such mechanism.

- **In-band message processed by all routers.** Some message types need to be processed by all routers in order to have effect. This is a tough requirement for any mechanism to be used in the Internet, and this kind of schemes are likely to remain in limited controlled portions of the network. These messages would also utilize reserved bits in IP header or IP options, with the same challenges as listed above. Additionally, in some cases the sender must be able to verify that all routers have processed the message. One way to do this is by the means of a separate TTL field in the message that is compared to the IPv4 TTL or IPv6 hop count. If the two fields do not give matching information about the number of hops in the path, it can be concluded that there were routers that did not process the notification message. IP tunnels are also a considerable challenge to this kind of mechanisms, as they can hide the inner IP header with the in-band message from the routers. Sometimes the TTL field comparison does not reveal the presence of such tunnels on the path. This work presents a mechanism that falls into this category.

- **Out-of-band message processed by end hosts.** Sending ICMP messages from the receiver to the sender of a packet has been a traditional way of

reporting, for example, some error condition in the data transfer. Usually the transport header, or a part of it, is included in the message to help the receiver of the ICMP message identify the transport connection the ICMP message concerns, and do some primitive security screening.

- **Out-of-band message processed by routers.** Resource ReserVation Protocol (RSVP) [27] uses a specific protocol type for QoS signaling between the sender and the receiver. RSVP requires that every router processes the messages, so it includes a similar kind of TTL-based hop tracking mechanism as mentioned above. In order to have out-of-band messages processed at routers, they need to be set to monitor the given protocol type inside the IP packets, or the IP packets need to use a router alert option [90, 125] to trigger further processing at the router. As with the in-band messages, IP tunnels and layer 2 switching systems such as MPLS [138] may prevent the signaling from working, or cause the signaling to work defectively. An out-of-band message could also be sent from one of the routers along the network path, of which some of the ICMP error messages are a common example. Taking strong actions based on such signaling can be dangerous, though, because there would be many security issues in the validity and authenticity of such messages.

To summarize, when analyzing cross-layer notification mechanisms, a number of issues should be considered based on the experiences from past proposals. To mention two of the more important issues, it should be determined whether some or all nodes along the path are required to process the message, and it should be evaluated whether it is feasible to embed the signaling into the protocol data traffic, or whether a separate signaling flow is more appropriate, either as embedded to some existing signaling such as Mobile IP binding updates [85], or using an entirely new protocol. It is also possible that a combination of different mechanisms is used: for example, a mobile host could use an end-to-end method to tell the corresponding node about change in its status. In response, a corresponding node could trigger a hop-by-hop QoS request in the changed environment.

2.4.2 Adjusting TCP sending rate using Quick-Start

As discussed in Section 2.2, the TCP senders are required to select a low initial sending rate to follow the congestion control principles. As acknowledgments start arriving, the TCP sender then increases its sending rate until it gets an indication of congestion. The appropriate sending rate depends on the bandwidth and propagation delay of the network path between the sender and receiver, as well as the amount of load being placed on the network by others at the given time.

This dissertation investigates the use of an in-band mechanism called Quick-Start to quickly find out the correct initial sending rate, and to quickly adapt the sending rate to the new path characteristics after a hand-off. When used with TCP, Quick-Start is used to quickly set the TCP congestion window size. Quick-Start is expected to be useful on clearly under-utilized network paths that would require several round-trip times from the TCP slow-start before being properly utilized. Even with an initial window of four packets, slow-start takes $log_2 N - 2$ round-trip times to reach a congestion window size of N packets. When using Quick-Start over an under-utilized path, it is possible that a transfer that would otherwise take several round-trip times, could be finished in a single round-trip time, i.e., the potential performance improvements could be huge. Since GPRS is a network technology with particularly high round-trip times, it can be expected that Quick-Start is useful when used over GPRS links. Because the Quick-Start is intended to deliver information about the bottleneck on the connection path, it needs to be processed by every router. The related challenges in such mechanisms are extensively discussed in Chapter 6.

In addition to Quick-Start and other explicit mechanisms to resolve the path capacity mentioned in Section 1.3, there are mechanisms to gain a higher initial sending rate without requiring specific support from routers. For example, SwiftStart [126] would use the first packets sent during slow start to estimate the bottleneck bandwidth, and then use that estimate as the basis for a rapid increase of the congestion window. There are also proposals for sharing information about network conditions between connections, ranging from TCP Fast Start [124] to the Congestion Manager [15], which would allow a new connections to start with a larger congestion window, based on the assessment of the network path conducted by previous connections. Recently, *Re-Feedback* has been proposed as a mechanism to control congestion response using explicit interaction with the network [31]. Re-Feedback can be applied, for example, by extending the use of the Explicit Congestion Notification bits.

2.5 Summary

This chapter outlined the problem area we are focusing on in this work. We discussed the recent evolution of the wireless communication systems. Within a little more than a decade the variety of different wireless communication technologies has increased tremendously, and the range of possible wireless link bandwidths used by a single device can vary from few tens of kilobits per second to few tens of megabits per second. Also the other link characteristics, such as the propagation delay vary significantly between different link technologies. The large variance in

network characteristics makes efficient networking at upper layer protocols difficult, especially with TCP that is based on measurement-based evaluation of the communication path characteristics.

We discussed the basic TCP algorithms and a number of performance enhancements made on TCP during its lifetime to fix the performance problems it was known to have under certain link behavior, and discussed a specific problem we are addressing in our work: spurious retransmission timeouts caused by an unexpected delay spike in link behavior, that are known to occur in GPRS networks. We showed that the delay spikes have a severe effect on TCP performance, they cause unnecessary retransmissions of several TCP segments, and hamper the TCP congestion control behavior. Finally we discussed another problem related to TCP congestion control behavior, the slow startup of a connection on high delay links, and slow convergence times to sudden changes in link characteristics and discuss the different types of explicit communication mechanisms that could be used to enhance the congestion control performance with better knowledge of the communication path characteristics. Later in this thesis we investigate one such mechanism, Quick-Start, in more detail.

CHAPTER 3

Congestion Control in Linux TCP

This chapter describes the Linux TCP implementation used in many of the experiments conducted in this dissertation. Linux is a freely available Unix-like operating system that has gained popularity in the last years. The Linux source code is publicly available[1], which makes Linux an attractive tool for the computer scientists in various research areas. Therefore, a large number of people have contributed to Linux development during its lifetime. In this chapter we describe the design solutions selected in the TCP implementation of the Linux kernel version 2.4. Linux TCP implements many of the RFC specifications in a single congestion control engine, using common code for supporting both SACK TCP and NewReno TCP. The Linux implementation also contains features that differ from the RFCs or other TCP implementations used today, and we believe that the protocol designers working with TCP find this information useful considering their work.

Building up a single consistent protocol implementation that conforms to the different RFCs is not a straightforward task. For example, the TCP congestion control specification [11] gives a detailed description of the basic congestion control algorithm, making it easier for the implementer to apply it. However, if the TCP implementation supports SACK TCP [117], it needs to follow congestion control specifications that use a partially different set of concepts and variables than those given in the standard congestion control RFC [51, 23]. Therefore, strictly following the algorithms used in the specifications makes an implementation unnecessarily complicated, as usually several RFCs are implemented at the same time.

This chapter is organized as follows. In Section 3.1 we discuss some aspects in

[1]The Linux kernel source can be obtained from `http://www.kernel.org/`.

the IETF specifications that were considered unsatisfying by the Linux community and implemented differently. In Section 3.2 we introduce the main concepts of the Linux TCP congestion control engine and describe the main algorithms governing the packet retransmission logic. In Section 3.3 we describe a number of Linux-specific features, for example concerning the retransmission timer calculation. In Section 3.4 we discuss how Linux TCP conforms to the IETF specifications related to TCP congestion control, and in Section 3.5 we illustrate the performance effects of selected Linux-specific design solutions. Section 3.6 summarizes this chapter.

3.1 Why Linux differs from standards?

Some details in the IETF specifications are problematic in practice. Although many of the RFCs suggest a general algorithm that could be applied to an implementation, combining the algorithms from several RFCs may be inconvenient. For example, combining the congestion control requirements for SACK TCP and NewReno TCP can be problematic due to different variables and algorithms used in the specifications.

The TCP congestion control specification artificially increases the congestion window during the fast recovery in order to let out forward transmissions that maintain a steady packet flow to the network and keep the ACK clock operational. Therefore, during fast recovery the congestion window size does not actually reflect the number of segments allowed to be outstanding in the network. When fast recovery is over, the congestion window is deflated back to a proper size. This procedure is needed because the congestion window is traditionally evaluated against the difference of the highest data segment transmitted (`SND.NXT`) and the first unacknowledged segment (`SND.UNA`). By taking a more flexible method for evaluating the number of outstanding segments, the congestion window size can be constantly maintained at an appropriate level that corresponds to the network capacity.

Adjusting the congestion window consistently becomes important when SACK information is used by the TCP sender. By using the selective acknowledgements, the sender can determine the number of outstanding packets in the network with a better accuracy than by just using the cumulative acknowledgements. In order to make a coherent implementation of the congestion control algorithms, it is desirable to have common variables and routines both for SACK TCP and for the TCP variant that is used when the other end does not support SACK.

Finally, the details of the retransmission timeout (RTO) calculation algorithm described in Chapter 2 have been questioned [114]. Because many networks have

round-trip delays of a few tens of milliseconds or less, the RTO algorithm details may not have a significant effect on TCP performance, since the minimum RTO value is limited to one second [127]. However, for high-delay network environments, such as GPRS, the effectiveness of the RTO calculation is important. It has been pointed out that the RTO estimator results in overly large values due to the weight given to variance of the round-trip time in the algorithm [114]. This may cause problems when the round-trip time suddenly drops for some reason. On the other hand, when the congestion window size increases at a steady pace during the slow start, it is possible that the RTO estimator is not increased fast enough due to small variance in the round-trip times. This may result in spurious retransmission timeouts. Alternative RTO estimators, such as the *Eifel Retransmission Timer* [114], have been suggested to overcome the potential problems in the standard RTO algorithm. Although the Eifel Retransmission Timer is efficient in avoiding the problems of the standard RTO algorithm, it introduces a rather complex set of equations compared to the standard RTO calculation. Therefore, evaluating the possible side effects of different network scenarios on Eifel RTT behavior is difficult.

3.2 The Linux Approach

Although Linux conforms to the TCP congestion control principles, it takes a different approach in carrying out the congestion control. Instead of comparing the congestion window to the difference of `SND.NXT` and `SND.UNA`, the Linux TCP sender determines the number of TCP segments currently outstanding in the network. When making decisions on how many segments to transmit, the Linux TCP sender compares the current number of outstanding segments to the congestion window that gives the maximum number of packets that are allowed to be in the network at a time. Unlike the Linux implementation, the TCP specifications and some implementations compare `cwnd` to the number of transmitted octets. This results in different behavior if segments are smaller than allowed by the *Maximum Segment Size (MSS)*: if the implementation uses a byte-based congestion window, it allows several small segments to be injected into the network for each MSS-sized segment in the congestion window. Linux, on the other hand, allows only a given number of packets to be in the network, no matter how small they are. Therefore, the Linux congestion control is more conservative compared to the byte-based approach when the TCP payload consists of small segments.

The Linux TCP sender uses the same set of variables and functions for determining the number of outstanding packets with the NewReno recovery and

with the two flavors of SACK recovery supported. When the SACK information is available, the sender can either follow the *Forward Acknowledgements (FACK)* [116] approach, or a more conservative approach that better conforms to the principles of the SACK recovery algorithm specified in the IETF standards track documentation [23]. As a basis for all recovery methods the Linux TCP sender uses the following equations in defining the number of segments outstanding in the network:

```
left_out <- sacked_out + lost_out
in_flight <- packets_out - left_out + retrans_out
```

In the equation above, `packets_out` is the number of originally transmitted segments above `SND.UNA`, `sacked_out` is the number of segments acknowledged by SACK blocks, `lost_out` is an estimation of the number of segments lost in the network, and `retrans_out` is the number of retransmitted segments. Determining the `lost_out` variable depends on the selected recovery method. For example, when FACK is in use, all unacknowledged segments between the highest SACK block and the cumulative acknowledgement are counted in `lost_out`. The selected approach makes it easy to add new heuristics for evaluating which segments are lost.

If the SACK option is not available, the Linux TCP sender increases `sacked_out` by one for each incoming duplicate acknowledgement. This is in conformance with the TCP congestion control specification, and the resulting behavior is similar to the *NewReno* algorithm with its forward transmissions. The design chosen in Linux does not require artificial inflation of the congestion window, but `cwnd` holds the valid number of segments allowed to be outstanding in the network throughout the fast recovery.

The counters used for tracking the number of outstanding, acknowledged, lost, or retransmitted packets require additional data structures to support them. The Linux sender maintains the state of each outstanding segment in a scoreboard, where it marks the known state of the segment. The segment can be marked as outstanding, acknowledged, retransmitted, or lost. Combinations of these bits are also possible. For example, a segment can be declared lost and retransmitted, in which case the sender is expecting to get an acknowledgement for the retransmission. Using this information the Linux sender knows which segments need to be retransmitted, and how to adjust the counters used for determining `in_flight` when a new acknowledgement arrives. The scoreboard also plays an important role when determining whether a segment has been incorrectly assumed lost, for example due to packet reordering.

The scoreboard markings and the counters used for determining the `in_flight` variable should be in consistent state at all times. Markings for out-

standing, acknowledged and retransmitted segments are straightforward to maintain, but the decision to place a *lost* mark depends on the recovery method used. With the NewReno recovery, the first unacknowledged packet is marked lost when the sender enters the fast recovery. In practice, this corresponds to the fast retransmit of the IETF congestion control specifications [11]. Furthermore, when a partial ACK not acknowledging all the data outstanding at the beginning of the fast recovery arrives, the first unacknowledged segment is marked lost. This results in retransmission of the next unacknowledged segment, as the NewReno specification requires [57].

When SACK is used, more than one segment can be marked lost at a time. With the conservative approach, the TCP sender does not count the holes between the acknowledged blocks in `lost_out`, but when FACK is enabled, the sender marks the holes between the SACK blocks lost as soon as they appear. The `lost_out` counter is adjusted appropriately.

The Linux TCP sender is governed by a state machine that determines the sender actions when acknowledgements arrive. The states are as follows:

- **Open**. This is the normal state in which the TCP sender follows the fast path of execution optimized for the common case, when processing the incoming acknowledgements. When an acknowledgement arrives, the sender increases the congestion window following either slow-start or congestion avoidance algorithms, depending on whether the congestion window is smaller or larger than the slow-start threshold, respectively.

- **Disorder**. When the sender detects duplicate acknowledgements or selective acknowledgements, it moves to the *Disorder* state. In this state the congestion window is not adjusted, but each incoming packet triggers transmission of a new segment. Therefore, the TCP sender follows the packet conservation principle [81], which requires that a new packet is not sent out until an old packet has left the network. In practice the behavior in this state is similar to the *limited transmit* specification by the IETF [7], which was suggested to allow more efficient recovery when the congestion window is small, or when a large number of segments are lost in the last window of transmission. Limited transmit allows fast retransmit in these situations, avoiding retransmission timeout that might be needed if limited transmit was not in use.

- **CWR**. The TCP sender may receive congestion notifications either by Explicit Congestion Notification [136], *ICMP source quench* [131], or from a local device. When receiving a congestion notification, the Linux sender does not reduce the congestion window at once, but by one segment for

every second incoming ACK until the window size is halved. When the sender is in the process of reducing the congestion window size and it does not have outstanding retransmissions, it is in *CWR (Congestion Window Reduced)* state. CWR state can be interrupted by the *Recovery* or *Loss* states described below.

- **Recovery**. After a sufficient number of successive duplicate ACKs arrive at the sender, it retransmits the first unacknowledged segment and enters the *Recovery* state. By default, the threshold for entering the *Recovery* state is three successive duplicate ACKs, a value recommended by the TCP congestion control specification. During the *Recovery* state, the congestion window size is reduced by one segment for every second incoming acknowledgement, similarly to the *CWR* state. The window reduction ends when the congestion window size is equal to ssthresh, i.e. half of the window size when entering the *Recovery* state. The congestion window is not increased during the recovery state, and the sender either retransmits the segments marked lost, or makes forward transmissions on new data according to the packet conservation principle. The sender stays in the *Recovery* state until all of the segments outstanding when the *Recovery* state was entered are successfully acknowledged. After this the sender goes back to the *Open* state. A retransmission timeout can also interrupt the *Recovery* state.

- **Loss**. When an RTO expires, the sender enters the *Loss* state. All outstanding segments are marked lost, and the congestion window is set to one segment. Therefore the sender starts increasing the congestion window using the slow start algorithm. A major difference between the *Loss* and *Recovery* states is that in the *Loss* state the congestion window can be increased according to the congestion control rules after the sender has reset it to one segment, but in the *Recovery* state the congestion window size can only be reduced. The *Loss* state cannot be interrupted by any other state, and the sender exits to the *Open* state only after all data outstanding when the *Loss* state began have successfully been acknowledged. For example, fast retransmit cannot be triggered during the *Loss* state, which is in conformance with the NewReno specification.

Linux TCP avoids explicit calls to transmit a packet in any of the abovementioned states, for example, regarding the fast retransmit. The current congestion control state determines how the congestion window is adjusted, and whether the sender considers the unacknowledged segments lost. After the TCP sender has processed an incoming acknowledgement according to its current state, it transmits a maximum of $(cwnd - in\_flight)$ segments to the network. The sender

first retransmits earlier segments marked lost and not yet retransmitted, or new data segments if there are no lost segments waiting for retransmission.

There are occasions where the number of outstanding segments decreases suddenly by several segments in the TCP bookkeeping. For example, after a loss or reordering of TCP acknowledgments, the next incoming acknowledgment may cover several segments. These situations would cause bursts of data to be transmitted into the network, unless they are taken into account in the TCP sender implementation. The prevalence and impact of *micro-bursts*, i.e. bursts caused by a single event such as ACK losses, are evaluated in [22]. The Linux TCP sender avoids the micro-bursts by limiting the congestion window to allow at most three segments to be transmitted for an incoming ACK. This is similar to the *Use It or Lose It* algorithm described in [8]. Since burst avoidance may cause reduction of the congestion window size below the slow start threshold, it is possible that the sender enters slow start after several segments have been acknowledged by a single ACK.

When a TCP connection is established, many of the TCP variables need to be initialized with some fixed values. In order to improve the communication efficiency at the beginning of the connection, after each connection the Linux TCP sender stores in its destination cache the slow start threshold, the variables used for the RTO estimation, and a variable that tracks the observed magnitude of packet reordering on the connection path. If another connection is established to the same destination, the cached values can be used to get initial values that are more likely to be adequate for the new TCP connection. *TCP Control Block Interdependence* [161] and the *Congestion Manager* [14, 15] are other mechanisms that have been proposed for reusing the past congestion control data in new TCP connections. A possible disadvantage in this scheme is that if the network conditions between the sender and the receiver change for some reason, the values in the destination cache might get outdated.

3.3 Features

We now list the most important Linux TCP features that may differ from a typical TCP implementation. Linux implements a number of TCP enhancements proposed recently by IETF, such as *Explicit Congestion Notification* [136] and *DSACK* [61]. To our knowledge, Linux was among the first systems to implement these features.

3.3.1 Retransmission timer calculation

Some TCP implementations use a coarse-grained retransmission timer, having granularities up to 500 ms. The round-trip time samples are often measured once in a round-trip time. In addition, the present retransmission timer specification requires that the RTO timer should not be less than one second [127]. Considering that most of the present networks provide round-trip times of less than 500 ms, studying the feasibility of the traditional retransmission timer algorithm standardized by the IETF has not excited much interest.

Linux TCP has a retransmission timer granularity of 10 ms and the sender takes a round-trip time sample for each segment[2]. Therefore it is capable of achieving more accurate estimations for the retransmission timer, if the assumptions in the timer algorithm are correct. Like many other implementations, Linux TCP deviates from the IETF specification by allowing a minimum limit of 200 ms for the RTO. Because of the finer timer granularity and the smaller minimum limit for the RTO timer, the correctness of the algorithm for determining the RTO is more important than with a coarse-grain timer. The traditional algorithm for retransmission timeout computation has been found to be problematic in some networking environments [114]. This is especially true if a fine-grained timer is used and the round-trip time samples are taken for each segment.

In Section 3.1 we described two problems regarding the standard RTO algorithm. First, when the round-trip time decreases suddenly, RTT variance increases momentarily and causes the RTO value to be overestimated. Second, the RTT variance can decay to a small value when RTT samples are taken for every segment while the window is large. This increases the risk for spurious RTOs that result in unnecessary retransmissions.

The Linux RTO estimator attacks the first problem by giving less weight for the mean deviance (`MDEV`) when the measured RTT decreases significantly below the smoothed average. A separate `MDEV` variable is used to calculate the final `RTTVAR` of the original algorithm as described below. The reduced weight given for the `MDEV` sample is based on the multipliers used in the standard RTO algorithm. First, the `MDEV` sample is weighed by $\frac{1}{8}$, corresponding to the multiplier used for the recent RTT measurement in the `SRTT` equation given in Section 2.2.1. Second, `MDEV` is further multiplied by $\frac{1}{4}$ corresponding to the weight of 4 given for the `RTTVAR` in the standard RTO algorithm. Therefore, choosing the weight of $\frac{1}{32}$ for the current `MDEV` neutralizes the effect of the sudden change of the measured RTT on the RTO estimator, and assures that RTO holds a steady value when

[2]Due to retransmission ambiguity, RTTs for retransmissions are not measured unless the TCP timestamps option is in use.

the measured RTT drops suddenly. This avoids the unwanted peak in the RTO estimator value, while maintaining a conservative behavior. If the round-trip times stay at the reduced level for the next measurements, the RTO estimator starts to decrease slowly to a lower value. In summary, the equation for calculating the MDEV is the following:

```
if (R < SRTT and |SRTT - R| > MDEV) {
     MDEV <- 31/32 * MDEV + 1/32 * |SRTT - R|
} else {
     MDEV <- 3/4 * MDEV + 1/4 * |SRTT - R|
}
```

where R is the recent round-trip time measurement, and SRTT is the smoothed average round-trip time. Linux does not directly modify the RTTVAR variable, but makes the adjustments first on the MDEV variable which is used in adjusting the RTTVAR that determines the RTO. The SRTT and RTO estimator variables are set according to the standard specification.

A separate MDEV variable is needed, because the Linux TCP sender allows decreasing the RTTVAR variable only once in a round-trip time. However, RTTVAR is increased immediately when MDEV gives a higher estimate, thus RTTVAR is the maximum of the MDEV estimates during the last round-trip time. The purpose of this solution is to avoid the problem of underestimated RTOs due to low round-trip time variance, which was the second of the problems described earlier.

Linux TCP supports the *TCP Timestamp option* [25] that allows accurate round-trip time measurement also for retransmitted segments, which is not possible without using timestamps. Having a proper algorithm for RTO calculation is even more important with the timestamp option. According to our experiments, the algorithm proposed above gives reasonable RTO estimates also with TCP timestamps, and avoids the pitfalls of the standard algorithm.

Figure 3.1 illustrates the above-mentioned differences of the standard RTO calculation and the Linux algorithm. The figure shows an arbitrarily generated sequence of round-trip time measurements (*mrtt*), simulated results of the output of the standard algorithm (RFC 2988) with the given round-trip times, and the corresponding output of the Linux algorithm. It is worth noting that, to illustrate the differences of the two algorithms, in this graph no minimum limit is applied to the retransmission timeout length. Neither is the timer granularity limited in this simulated scenario in any way. The figure shows how the Linux timer estimate decays slower than the standard algorithm. Furthermore, with small variation in round-trip times the standard algorithm causes the RTO estimate to approach very close to the round-trip times, which increases the risk of spurious retransmission

timeouts. The figure also shows how a sudden reduction of the measured round-trip times causes a momentary increase of the RTO in the standard algorithm, but not in the Linux variant, while Linux RTO estimator values increase as quickly as with the standard estimator, when round-trip times increase again. While the sequence of round-trip times may seem arbitrary (which they are, in this case), for example in the context of vertical hand-offs discussed more in Chapter 7, this kind of sudden changes in round-trip times are possible.

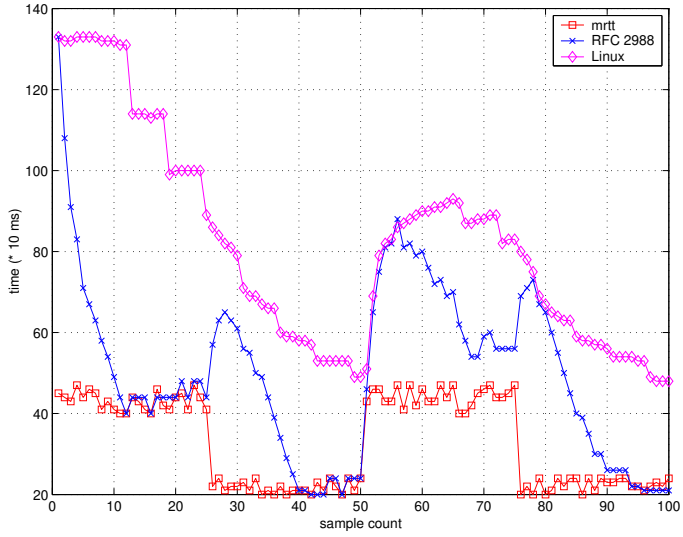


Figure 3.1: Comparison of standard RTO calculation and the Linux algorithm.

The retransmission timer is reset every time an acknowledgement advancing the window arrives at the sender. The retransmission timer is also reset when the sender enters the *Recovery* state and retransmits the first segment. During the rest of the Recovery state the retransmission timer is not reset, but a packet is marked lost, if more than an RTO's worth of time has passed from the first transmission of the same segment. This allows more efficient retransmission of packets during the *Recovery* state even though the information from acknowledgements is not sufficient enough to declare the packet lost. However, this method can only be used for segments not yet retransmitted.

3.3.2 Undoing congestion window adjustments

Because the currently used mechanisms in the Internet do not provide explicit loss information to the TCP sender, it needs to speculate when keeping track of

which packets are lost in the network. For example, reordering is often a problem for the TCP sender because it cannot distinguish whether the missing ACKs are caused by a packet loss or by a delayed packet that will arrive later. The Linux TCP sender can, however, detect unnecessary congestion window adjustments afterwards, and do the necessary corrections in the congestion control parameters. For this purpose, when entering the *Recovery* or *Loss* states, the Linux TCP sender stores the old `ssthresh` value prior to adjusting it.

A delayed segment can trigger an unnecessary retransmission, either by causing a spurious retransmission timeout or by causing packet reordering. The Linux TCP sender has mainly two methods for detecting afterwards that it unnecessarily retransmitted the segment. First, the receiver can inform by a *Duplicate-SACK (DSACK)* that the incoming segment was already received. If all segments retransmitted during the last recovery period are acknowledged by DSACK, the sender knows that the recovery period was unnecessarily triggered. Second, the Linux TCP sender can detect unnecessary retransmissions by using the TCP timestamp option [25] attached to each TCP header. When this option is in use, the TCP receiver echoes the timestamp of the segment that triggered the acknowledgement back to the sender, allowing the TCP sender to conclude whether the ACK was triggered by the original or by the retransmitted segment. The *Eifel* algorithm [111] uses a similar method for detecting spurious retransmissions.

When an unnecessary retransmission is detected by using TCP timestamps, the logic for undoing the congestion window adjustments is simple. If the sender is in the *Loss* state, i.e., it is retransmitting after an RTO which was triggered unnecessarily, the *lost* mark is removed from all segments in the scoreboard, causing the sender to continue with transmitting new data instead of retransmissions. In addition, `cwnd` is set to the maximum of its present value and `ssthresh` * 2, and the `ssthresh` is set to its prior value stored earlier. Since `ssthresh` was set to half of the number of outstanding segments when the packet loss is detected, the effect is to continue in congestion avoidance at a similar rate as when the *Loss* state was entered.

Unnecessary retransmission can also be detected by the TCP timestamps while the sender is in the *Recovery* state. In this case the Recovery state is finished normally, with the exception that the congestion window is increased to the maximum of its present value and `ssthresh` * 2, and `ssthresh` is set to its prior value. In addition, when a partial ACK for the unnecessary retransmission arrives, the sender does not mark the next unacknowledged segment lost, but continues according to present scoreboard markings, possibly transmitting new data.

In order to use DSACK for undoing the congestion control parameters, the TCP sender tracks the number of retransmissions that have to be declared unneces-

sary before reverting the congestion control parameters. When the sender detects a DSACK block, it reduces the number of revertable outstanding retransmissions by one. If the DSACK blocks eventually acknowledge every retransmission in the last window as unnecessarily made, and the retransmission counter falls to zero due to DSACKs, the sender increases the congestion window and reverts the last modification to `ssthresh` similarly to what was described above.

While handling the unnecessary retransmissions, the Linux TCP sender maintains a metric measuring the observed reordering in the network in variable `reordering`. This variable is also stored in the destination cache after the connection is finished. `reordering` is updated when the Linux sender detects unnecessary retransmission during the *Recovery* state by TCP timestamps or DSACK, or when an incoming acknowledgement is for an unacknowledged hole in the sequence number space below selectively acknowledged sequence numbers. In these cases `reordering` is set to the number of segments between the highest segment acknowledged and the currently acknowledged segment, in other words, it corresponds to the maximum distance of reordering in segments detected in the network. Additionally, if FACK was in use when reordering was detected, the sender switches to use the conservative variant of SACK, which is not too aggressive in a network involving reordering.

3.3.3 Delayed acknowledgements

The TCP specifications state that the TCP receiver should delay the acknowledgements for a maximum time of 500 ms in order to reduce the number of acknowledgements generated by the receiver. The specifications do not mandate any specific delay time, but many implementations use a static delay of 200 ms for this purpose. However, a fixed delay time may not be adequate in all networking environments with different properties. Thus, the Linux TCP receiver adjusts the timer for delaying acknowledgements dynamically according to the packet interarrival time, trying to estimate the time it takes to receive the next two segments, while sending acknowledgements for at least every second incoming segment. A similar approach was also suggested in an early RFC by Clark [38]. However, the maximum delay for sending an acknowledgement is limited to 200 ms.

Using delayed ACKs slows down the TCP sender, because it increases the congestion window size based on the rate of incoming acknowledgements. In order to speed up the transmission in the beginning of the slow start, the Linux TCP receiver refrains from delaying the acknowledgements for the first incoming segments at the beginning of the connection. This is called *quick acknowledgements*.

The number of quick acknowledgements sent by the Linux TCP receiver is at most half of the number of segments required to reach the receiver's advertised

window limit. Therefore, using quick acknowledgements does not open the opportunity for the Silly Window Syndrome [38] to occur. In addition, the Linux receiver monitors whether the traffic appears to be bidirectional, in which case it disables the quick acknowledgements mechanism. This is done to avoid transmitting pure acknowledgements unnecessarily when they can be piggybacked with data segments.

3.3.4 Congestion Window Validation

The Linux sender reduces the congestion window size if it has not been fully used for one RTO estimate's worth of time. This scheme is similar to the *Congestion Window Validation* documented in an Experimental RFC 2861 [67]. The motivation for Congestion Window Validation is that if the congestion window is not fully used, the TCP sender may have an invalid estimate of the present network conditions. Therefore, a network-friendly sender should reduce the congestion window as a precaution.

When the Congestion Window Validation is triggered, the TCP sender decreases the congestion window to halfway between the actually used window and the present congestion window. Before doing this, `ssthresh` is set to the maximum of its current value and $\frac{3}{4}$ of the congestion window, as suggested in RFC 2861.

3.3.5 Explicit Congestion Notification

Linux implements *Explicit Congestion Notification (ECN)* [136] to allow the ECN-capable congested routers to report congestion before dropping packets. A congested router can mark a bit in the IP header, which is then echoed to the TCP sender by an ECN-capable TCP receiver. When the TCP sender gets the congestion signal, it enters the *CWR* state, in which it gradually decreases the congestion window to half of its current size at the rate of one segment for two incoming acknowledgements. Besides making it possible for the TCP sender to avoid some of the congestion losses, ECN is expected to improve the network performance when it is more widely deployed to the Internet routers.

3.4 Conformance to the IETF Specifications

Since Linux combines the features specified in different IETF specifications following certain design principles described earlier, some IETF specifications are not fully implemented according to the algorithms given in the RFCs. Table 3.1

Table 3.1: TCP congestion control related IETF specifications implemented in Linux. + = implemented, * = implemented, but details differ from specification.

Specification	Status
RFC 1323 (Perf. Extensions)	+
RFC 2018 (SACK)	+
RFC 2140 (Ctrl block sharing)	+
RFC 2581 (Congestion control)	*
RFC 2582 (NewReno)	*
RFC 2861 (Cwnd validation)	+
RFC 2883 (DSACK)	+
RFC 2988 (RTO)	*
RFC 3042 (Lim. xmit)	+
RFC 3168 (ECN)	*

shows which RFC specifications related to TCP congestion control are implemented in Linux[3]. Some of the features shown in the table can be found in Linux, but they do not fully follow the given specification in all details. These features are marked with an asterisk in the table, and we will explain the differences between Linux and the corresponding RFC in more detail below.

Linux fast recovery does not fully follow the behavior given in RFC 2582. First, the sender dynamically adjusts the threshold for triggering fast retransmit, based on the observed reordering in the network. Therefore, it is possible that the third duplicate ACK does not trigger a fast retransmit in all situations. Second, the Linux sender does not artificially adjust the congestion window during fast recovery, but maintains its size while adjusting the `in_flight` estimator based on incoming acknowledgements. The different approach alone would not cause a significant effect on TCP performance, but when entering the fast recovery, the Linux sender does not reduce the congestion window size at once, as RFC 2582 suggests. Instead, the sender decreases the congestion window size gradually, by one segment per two incoming acknowledgements, until the congestion window meets half of its original value. This approach was originally suggested by Hoe [71], and later it was named *Rate-halving* by an expired Internet Draft by Mathis, et. al. Rate-halving avoids pauses in transmission, but is slightly too ag-

[3]After this analysis was conducted some new RFCs have been published for features implemented in Linux. Also the Linux TCP behavior might have changed in the latest versions. These are not shown in the table. For example RFC 4138 specifying the F-RTO algorithm is one such RFC.

gressive after the congestion notification, until the congestion window has reached a proper size.

As described in Section 3.3, the round-trip time estimator and RTO calculation in Linux differs from the Proposed Standard specification by the IETF. Linux follows the basic patterns given in RFC 2988, but the implementation differs from the specification in adjusting the RTTVAR. A significant difference between RFC 2988 and Linux implementation is that Linux uses the minimum RTO limit of 200 ms instead of 1000 ms given in RFC 2988.

RFC 2018 defines the format and basic usage of the SACK blocks, but does not give detailed specification of the congestion control algorithm that should be used with SACK. FACK is the default congestion control algorithm applied when the SACK option is in use. However, since FACK results in overly aggressive behavior when packets have been reordered in the network, the Linux sender changes from FACK to a more conservative congestion control algorithm when it detects reordering. The SACK recovery algorithm specified by the IETF [23] is similar to the conservative SACK alternative in Linux. Furthermore, Linux follows the DSACK basics given in RFC 2883.

Linux implements RFC 1323, which defines the TCP timestamp and window scaling options, and the limited transmit enhancement defined in RFC 3042, which is taken care of by the *Disorder* state of the Linux TCP state machine. However, if the `reordering` estimator has been increased from the default of three segments, the Linux TCP sender transmits a new segment for each incoming acknowledgement, not only for the two first ACKs. Finally, the Linux destination cache provides functionality similar to the RFC 2140 that proposes Control Block Interdependence between the TCP connections.

3.5 Performance Issues

We now illustrate the behavior of the selected Linux TCP features by a few simple test cases, and discuss the potential performance effect of these features. We illustrate the implications of using quick acknowledgements, rate-halving, and congestion window reversion. We do this by disabling these features, and comparing the time-sequence diagrams of a pure Linux TCP implementation and an implementation with the corresponding feature disabled. We use Linux hosts as connection endpoints communicating over a 256 Kbps link with MTU of 1500 bytes. Between the sender and the 256 Kbps link there is a tail-drop router with buffer space for seven packets, connected to the sender with a high-bandwidth link with small latency. We have chosen a simple experimentation setup to illustrate the functionality of the TCP enhancements, without trying to build a detailed model

of any real network setup. However, our parameter setup is roughly similar to the characteristics of a modern wireless link technology, with a buffer size chosen to match the link's bandwidth-delay product, to keep the bottleneck link utilized also on periods of disruption in data transfer. The test setup is illustrated in Figure 3.2. In addition to the low bandwidth, the link between the router and TCP receiver has a fairly high propagation delay of 200 ms. The slow link and the router are emulated using the Seawind real-time network emulator [100]. With the network emulator we can control the link and the network parameters and collect statistics and log about the network behavior to help the analysis.

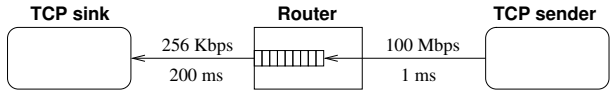


Figure 3.2: Test setup.

We first illustrate the effect of quick acknowledgements on TCP throughput. Figure 3.3(a) on page 53 shows the slow start performance of unmodified Linux implementing quick acknowledgements, and Figure 3.3(b) shows the performance of an implementation with the quick acknowledgements mechanism disabled. The latter implementation applies a static delay of 200 ms for every acknowledgement, but transmits an acknowledgement immediately if more than one full-sized segment's worth of unacknowledged data has arrived. One can see that when the link has a high bandwidth-delay product like in our case, the benefit of quick acknowledgements is noticeable. The unmodified Linux sender has transmitted 50 KB in 2 seconds, but when the quick acknowledgments are disabled, it takes 2.5 seconds for the sender to transmit 50 KB. In our example, the unmodified Linux receiver with quick acknowledgements enabled sent 109 ACK packets, and the implementation without quick acknowledgements sent 95 ACK packets. Because quick acknowledgements cause more ACKs to be generated in the network than when using the conventional delayed ACKs, the sender's congestion window increases slightly faster. Although this improves the TCP performance, it makes the network slightly more prone to congestion.

Rate-halving is expected to result in a similar average transmission rate as the conventional TCP fast recovery, but it paces the transmission of segments smoothly by making the TCP sender reduce its congestion window steadily instead of making a sudden adjustment. Figure 3.4(a) illustrates the performance of an unmodified Linux TCP implementing rate-halving, and Figure 3.4(b) illustrates the performance of an implementation with the conventional fast recovery behavior. These figures also illustrate the receiver's advertised window (the uppermost

line), since it limits the fast recovery in our example.

The scenario is the same in both figures: the router buffer becomes full during TCP slow-start, and several packets are dropped due to congestion before the feedback of the first packet loss arrives at the sender. The packet losses at the bottleneck link due to initial slow start is called slow start overshoot [41]. The figures show that after 12 seconds both TCP variants have transmitted 160 KB. However, the behavior of the unmodified Linux TCP is different from the TCP with rate-halving disabled. With the conventional fast recovery, the TCP sender stops sending new data until the number of outstanding segments has dropped to half of the original amount, but the sender with the rate-halving algorithm lets the number of outstanding segments reduce steadily, with the rate of one segment for two incoming acknowledgements. Both variants suffer from the advertised window limitation, which does not allow the sender to transmit new data, even though the congestion window would.

Finally, we show how the timestamp-based undoing similar to the Eifel algorithm [111] affects TCP performance. We generated a three-second delay, which is long enough to trigger a retransmission timeout at the TCP sender. Figure 3.5(a) shows a TCP implementation with the TCP timestamp option enabled, and Figure 3.5(b) shows the same scenario with timestamps disabled. The acknowledgements arrive at the sender in a burst, because during the delay packets queue up in the emulated link receive buffers and are all released when the delay is over[4].

The use of timestamps improves the TCP performance considerably, because the TCP sender detects that the acknowledgement following the retransmission was for the original transmission of the segment. Therefore the sender can revert the `ssthresh` to its previous value and increase the congestion window. Moreover, the Linux TCP sender avoids unnecessary retransmissions of the segments in the last window. The ACK burst injected by the receiver after the delay causes 19 new segments to be transmitted by the sender within a short time interval. However, the sender follows the slow start correctly as clocked by the incoming acknowledgements, and none of the segments are transmitted unnecessarily. A potential drawback of fully reverting the congestion control parameters is that it may create congestion at the bottleneck router. This effect is further emphasized in our scenario due to the burst of acknowledgments that arrive at the sender after the spurious timeout.

A conventional TCP sender not implementing the Eifel-style congestion window reversion retransmits the last window following the first delayed segment

[4]The delay stands for emulated events on the link layer, for example representing persistent retransmissions of erroneous link layer frames. The link receive buffer holds the successfully received packets until the period of retransmissions is over to be able to deliver them in order to the receiver.

unnecessarily. Not only does this waste the available bandwidth, but the retransmitted segments appearing as out-of-order data at the receiver trigger several duplicate acknowledgments. However, since the TCP sender is still in the *Loss* state, the duplicate ACKs do not cause further retransmissions[5]. One can see that the conventional TCP sender without timestamps has received acknowledgements for 165 KB of data in the 10 seconds after the transmission begun, while the Linux sender implementing TCP timestamps and congestion window reverting has received acknowledgements for 175 KB of data. The Linux TCP sender having TCP timestamps enabled retransmitted 22.6 KB in 16 packets, but the Linux TCP sender without timestamps retransmitted 37.1 KB in 26 packets in the test case transmitting a total of 200 KB. The link scenario was the same in both test runs, having a 3-second delay in the middle of transmission. When the TCP timestamps were not used, the TCP sender retransmitted 11 packets unnecessarily.

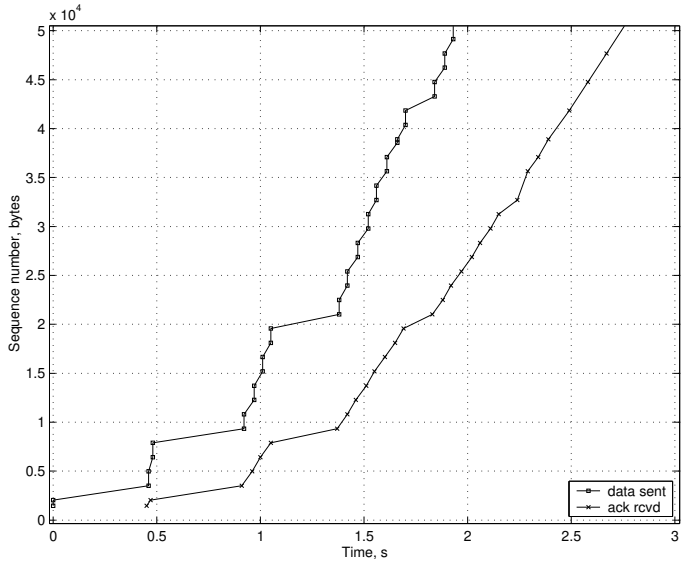
3.6 Summary

This chapter presented the basic ideas of the Linux TCP implementation, and gave a description of the details that differ from a typical TCP implementation. Linux implements many of the recent TCP enhancements suggested by the IETF. Therefore Linux provides a platform for testing the interoperability of the recent enhancements in an actual network. The current design also makes it easy to implement and study alternative congestion control policies.

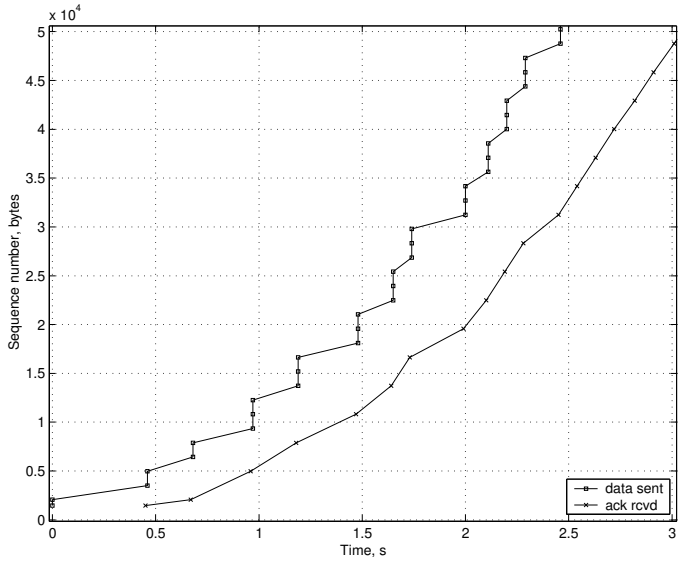
The Linux TCP behavior is strongly governed by the packet conservation principle and the sender's estimate of which packets are still in the network, which are acknowledged, and which are declared lost. Whether to retransmit or transmit new data depends on the markings made in the TCP scoreboard. In most of the cases none of the requirements given by the IETF are violated, although in some situations the detailed behavior may be different from what is given in the IETF specifications. However, the TCP essentials, in particular the congestion control principles and the conservation of packets, are maintained in all cases.

The Linux TCP implementation has been under much discussion and controversy for example in the IETF because of the certain special characteristics described in this chapter. Therefore we hope that this chapter helps in removing some of the uncertainty people have about the Linux implementation. We also hope that the information in this chapter is useful in research that analyzes the TCP performance using the Linux implementation.

[5]The behavior is similar to the NewReno "bugfix" [56]

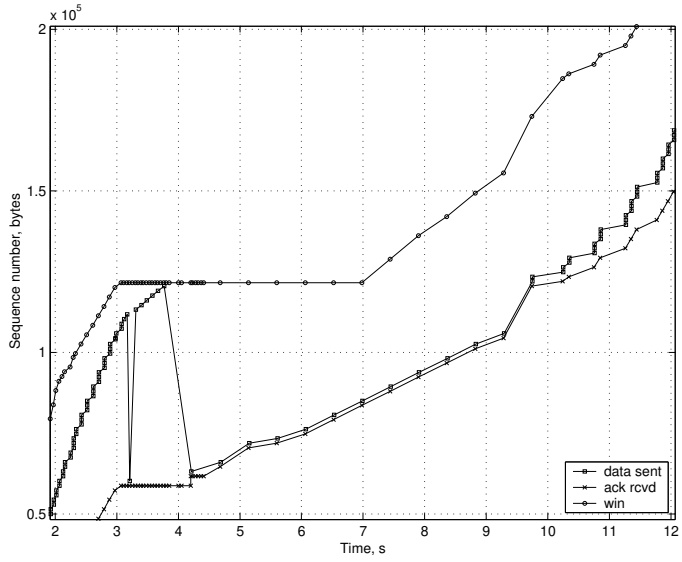


(a) Quick acknowledgements enabled.

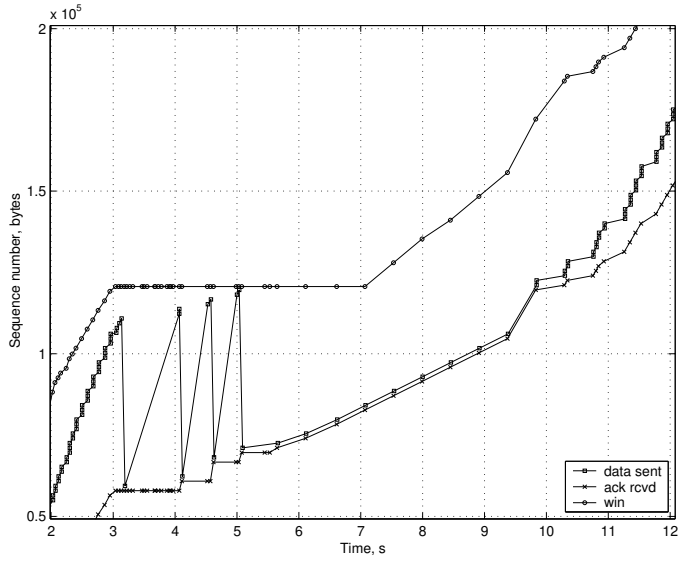


(b) Quick acknowledgements disabled.

Figure 3.3: Effect of quick acknowledgements on slow start performance.

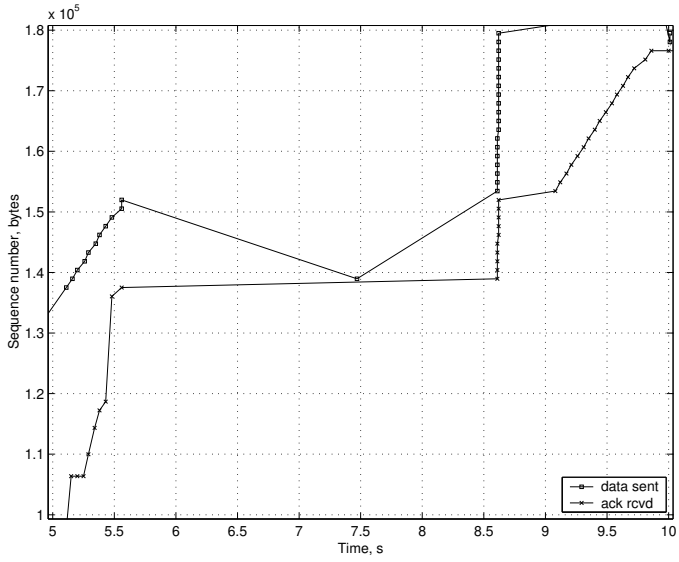


(a) Rate-halving enabled.

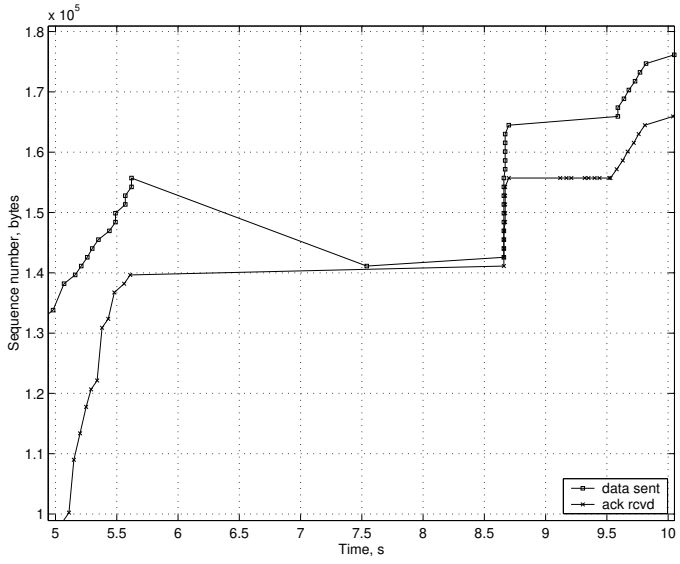


(b) Rate-halving disabled.

Figure 3.4: Effect of Rate-halving on TCP performance.



(a) TCP timestamps enabled.



(b) TCP timestamps disabled.

Figure 3.5: Effect of congestion window undoing on TCP performance.

CHAPTER 4

F-RTO: A Recovery Algorithm for TCP Retransmission Timeouts

In this chapter we focus on attacking the TCP performance problems resulting from unnecessary retransmissions that originate from spurious RTOs. A new RTO recovery algorithm was developed as part of this work, called *Forward RTO-Recovery (F-RTO)*, to improve the TCP performance after a spurious retransmission timeout. The F-RTO algorithm uses a set of simple rules for avoiding unnecessary retransmissions after a spurious RTO. The F-RTO recovery algorithm does not require use of any TCP options or additional bits in the TCP header, unlike the Eifel algorithm [111], for example.

The rest of the chapter is organized as follows. In Section 4.1 we discuss the TCP behavior after spurious retransmission timeouts, and what is the general idea of our approach to improve TCP's behavior on these occasions. In Section 4.2 we give a detailed definition of 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 4.3. In Section 4.4 we describe the experiments made with F-RTO in different network environments and the results of the experiments. Finally, we wrap up the main observations made in this chapter in Section 4.5.

4.1 Spurious Retransmission Timeouts

Because wireless networks are often subject to a high packet loss rate due to corruption or hand-offs, reliable link-layer protocols are widely employed with wireless links [113, 50], and some wireless links may be unusable without some link ARQ mechanism. The link-layer receiver often aims to deliver the packets to the

upper protocol layers in order, which implies that the later arriving packets are blocked until the head of the queue arrives successfully. Due to the strict link-layer ordering, the communication end points observe a pause in packet delivery that can cause a spurious TCP RTO instead of getting out-of-order packets that could result in a false fast retransmit instead. Either way, interaction between TCP retransmission mechanisms and link-layer recovery can cause poor performance.

Wireless links may also suffer from link outages that 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. In this case the TCP RTO may also be triggered spuriously. Other potential reasons for sudden delays that have been reported to 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 [64, 94]. In recent multi-access wireless terminals the hand-offs from low-latency WLAN link to high-latency GPRS link can also cause a spurious timeout.

As described in Chapter 2, TCP uses the *fast retransmits* [11] as the main mechanism to timely trigger retransmissions after receiving three successive duplicate acknowledgements (ACKs). If for a certain time period the TCP sender does not receive ACKs that acknowledge new data, the TCP retransmission timer expires as a backoff retransmission mechanism. 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 in slow-start after the RTO is likely to be the most efficient way of recovering.

In the normal RTO recovery the TCP sender retransmits the first unacknowledged segment, sets the congestion window to one segment and the *slow-start threshold (ssthresh)* to half of the number of currently outstanding segments, when the RTO expires. After this the sender continues in slow-start, increasing the congestion window by one segment on each ACK that advances the window and retransmitting the next unacknowledged segments allowed by the congestion window. However, if no segments were lost but the retransmission timer expires spuriously, the segments retransmitted in the slow-start are sent unnecessarily. The cumulative acknowledgements for the original transmissions appear at the TCP sender one at a time, triggering further unnecessary retransmissions. In particular,

this is very costly for slow links. Because there still are segments outstanding in the network, a false slow start is harmful for the potentially congested network as it injects extra segments into the network at increasing rate. Particularly, this phenomenon is very possible with the various wireless access network technologies that are prone to sudden delay spikes. Additionally, the TCP sender unnecessarily reduces the TCP congestion window to one segment and reduces the slow-start threshold to half of the currently used TCP window. This, in turn, is costly for high-bandwidth links as it takes a long time for the sender to reopen the window.

4.2 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. Although the main motivation of the F-RTO 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 [52, 11]. 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 conventional 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. F-RTO can be considered somewhat similar to the *Limited Transmit* algorithm [7], but applied to the RTO recovery.

The guideline behind F-RTO is that an RTO either indicates a loss, or it is caused by an excessive delay in packet delivery while there still are outstanding segments in flight. If the RTO was due to delay, that is, the RTO was spurious, acknowledgements for non-retransmitted segments sent before the RTO should arrive at the sender after the RTO occurred. If no such segments arrive, the RTO is concluded to be non-spurious and the conventional RTO recovery with go-back-N retransmissions should take place at the TCP sender.

To implement the principle described above, an F-RTO sender acts as follows: if the first ACK arriving after a RTO-triggered retransmission advances the window, transmit two new segments instead of continuing retransmissions. If the second incoming acknowledgement also advances the window, RTO is likely to be spurious, because the second ACK is triggered by an originally transmitted segment that has not been retransmitted after the RTO. If the RTO was genuine and caused by packet loss, the two new segments transmitted after the RTO would

appear as out-of-order segments at the receiver and trigger duplicate acknowledgements. Therefore, if either one of the two acknowledgements after RTO is a duplicate ACK, the sender continues retransmissions similarly to the conventional 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 *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.

   (a) *If the acknowledgement advances `SND.UNA`, transmit up to two new (previously unsent) segments*. This is the main point in which the F-RTO algorithm differs from the conventional 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 two new segments is not possible due to advertised window limitation, or because there is no more data to send, the sender may transmit only one segment. If no new data can be transmitted, the TCP sender follows the conventional RTO recovery algorithm and starts retransmitting the unacknowledged data using slow start.

   (b) *If the acknowledgement is duplicate ACK, set the congestion window to one segment and proceed with the conventional RTO recovery.* 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.

   (a) *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 conservative 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 [56][1]. The `send_high` variable was proposed as a *"bugfix"* for avoiding unnecessary multiple fast retransmits when RTO expires during fast recovery with NewReno TCP. The problem of multiple fast retransmits can occur when the TCP sender unnecessarily retransmits segments that have already been received by the TCP receiver. This can happen, for example, when retransmission timeout occurs during fast recovery. In this case the sender will receive duplicate acknowledgements that are not caused by packet loss, but the out-of-order segments that were unnecessarily transmitted. The NewReno "bugfix" says that when receiving such duplicate acknowledgements below the `send_high` variable that set after each retransmission timeout to indicate the highest sequence number transmitted so far, the sender should not enter fast retransmit or fast recovery. However, when applying the F-RTO, the sender has not retransmitted other segments but the one that triggered RTO at this point, the

[1]The Standards Track revision of NewReno [57] uses variable name `recover` instead of `send_high`, and has included the "bugfix" as part of the standard algorithm. However, we will use `send_high` in this dissertation.

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 to detect whether the new RTO is spurious.

(b) *If the acknowledgement is a duplicate ACK, set the 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 that 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 4.3. 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 4.3.

When algorithm branch (3a) 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 increases the congestion window only once in a round-trip time after RTO, and therefore 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 DSACK 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. The benefits of using DSACK to detect unnecessary retransmissions are analyzed in [20]. In general, it is possible to separate the detection of a spurious RTO from the actions taken as congestion control response, and employ a different response alternative than what was described

above. We will discuss some of the suggested response alternatives that could be applied with F-RTO-based detection in Chapter 5.

An additional condition to the second step of the above-presented algorithm is that *if an ACK acknowledges the whole outstanding window up to the highest transmitted segment at algorithm branch (2a), the TCP sender should not declare the RTO spurious, but follow the conventional TCP behavior.* A common case of this is that the RTO was caused due to lost retransmission, and the rest of the window was successfully delivered to the receiver before the RTO occurred. In this case the ACK following the RTO acknowledges all of the outstanding window, and the F-RTO algorithm as described above could end up in algorithm branch (3a) that is meant to be applied in the case of a spurious RTO. This condition was left out from the above algorithm, because we apply a conservative response after a spurious retransmission timeout, and because the *Use It or Lose It* - type burst avoidance in Linux ensures that the TCP sender is never more aggressive than it is in slow-start. Therefore in this case there is no risk of congestion control violation or performance penalty. However, considering the use of F-RTO as a generic detection mechanism for spurious RTOs the additional condition given above is recommended. The algorithm in the IETF specification of F-RTO requires applying the above condition [144].

Branch (3a) can also be taken in a special case when the RTO retransmission is lost after a spurious retransmission timeout. Because the acknowledgements of the original transmissions arrive at the sender, it can continue transmitting new data without noticing the loss of RTO retransmission. Because any packet loss can be a sign of congestion, fully undoing the congestion control parameters would be a violation of the congestion control principles. The same is true also for the Eifel algorithm, but with DSACK it is possible to notice that the RTO retransmission did not reach the receiver. Therefore, we consider that reducing the congestion window to half of its previous size is an adequate action at this point, because a similar action is taken when the TCP sender enters fast recovery.

4.3 Discussion of F-RTO Behavior in Specific Scenarios

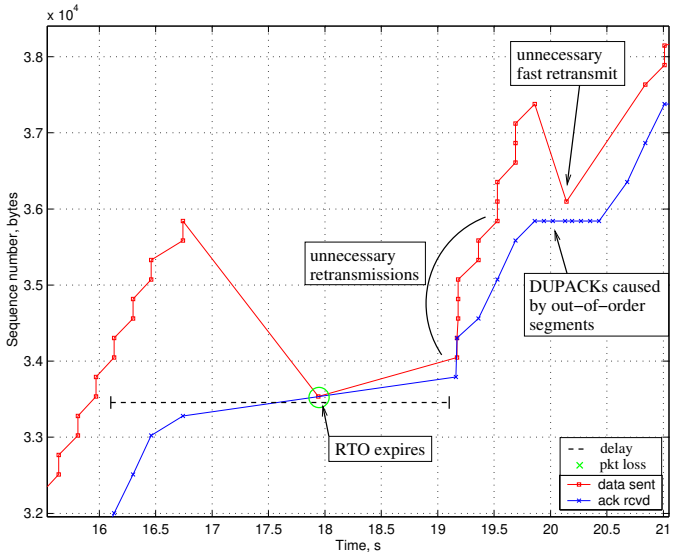
In this section we discuss the different reasons that may cause the RTO to expire and study the different scenarios after a retransmission timeout has expired due to these reasons. We compare the packet traces produced using the conventional RTO recovery and using F-RTO, and discuss the differences of the two recovery methods. *Selective Acknowledgements (SACK)* [117] and *limited transmit* [7] 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.

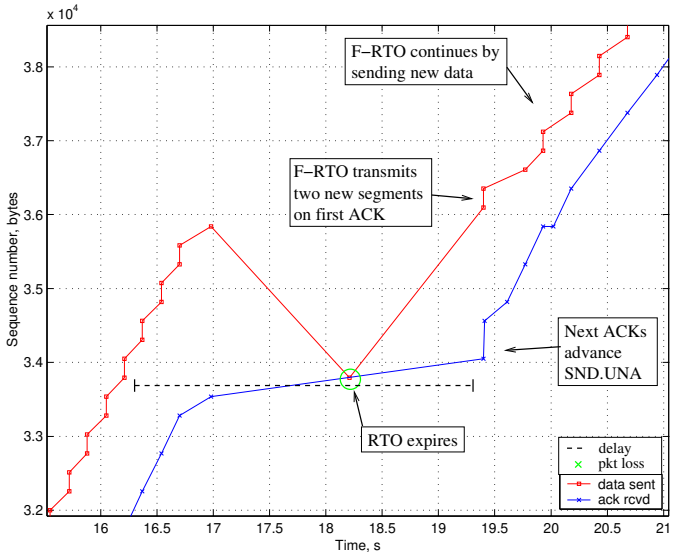
4.3.1 Sudden delays

Recovering efficiently from spurious retransmission timeouts is the main motivation of the F-RTO algorithm. Figure 4.1 compares the packet traces of the conventional RTO recovery and F-RTO. Figure 4.1(a) shows that the conventional 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.

Figure 4.1(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.



(a) Conventional RTO recovery.



(b) F-RTO recovery.

Figure 4.1: Comparison of the conventional RTO and F-RTO after an excessive delay.

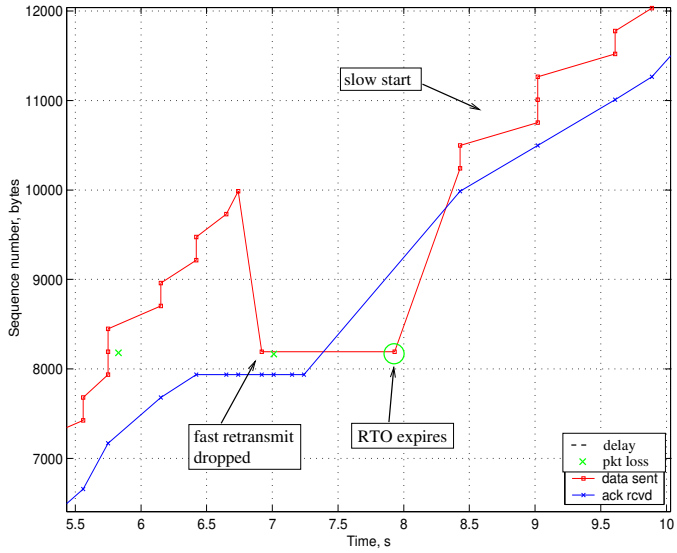
4.3.2 Lost retransmission

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 4.2 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 conventional 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.

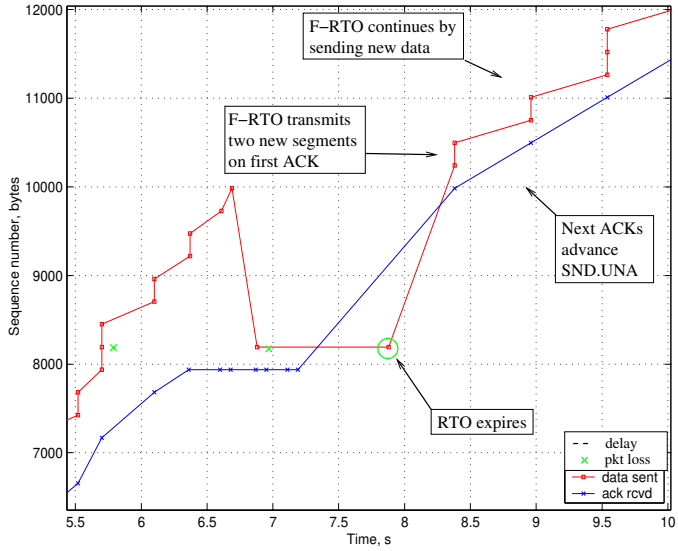
Figure 4.2(a) shows that when the RTO retransmission arrives at the receiver, it acknowledges the whole window, and the conventional TCP sender can proceed with sending new data in slow start. In the presented case the F-RTO recovery shown in Figure 4.2(b) differs from the conventional recovery only by not entering slow start after the RTO[2]. 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, like the conventional RTO recovery does.

[2]This is the scenario targeted at by the additional condition given in the end of Section 4.2.



(a) Conventional RTO recovery.



(b) F-RTO recovery.

Figure 4.2: Comparison of the conventional RTO and F-RTO after a lost retransmission.

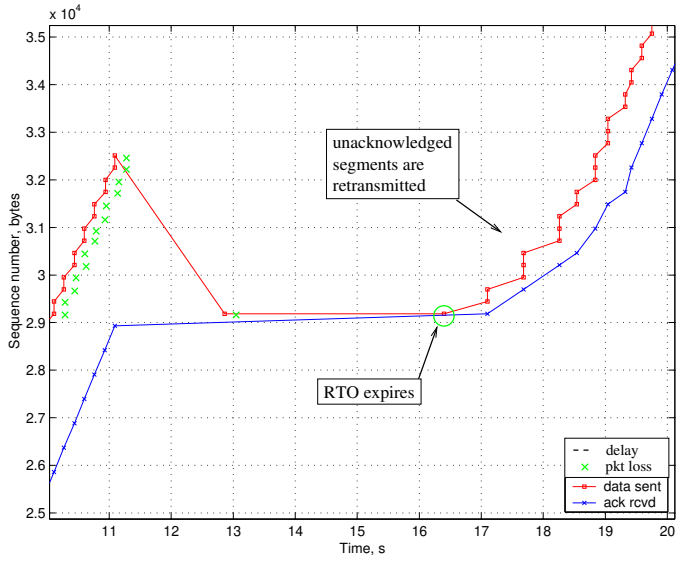
4.3.3 Burst losses

Because losses of several successive packets can result in a retransmission timeout, it is interesting to compare the F-RTO behavior with the conventional RTO recovery in such a case. Figure 4.3 compares the packet trace of the conventional 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 4.3(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[3].

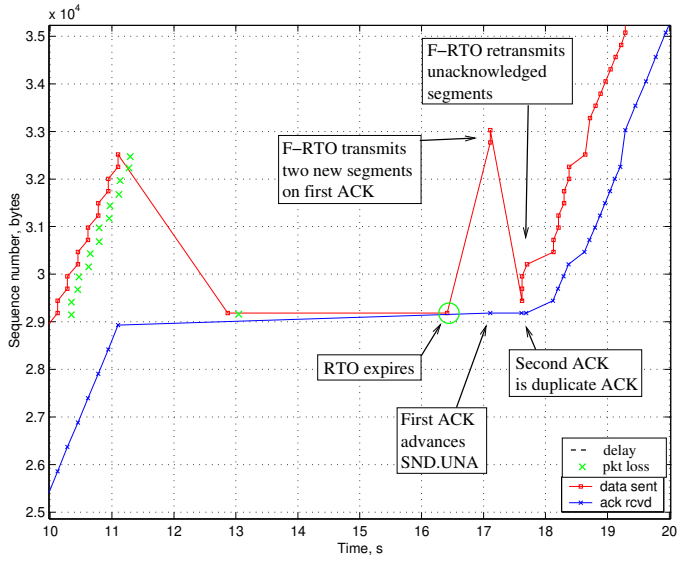
Figure 4.3(b) shows a similar scenario with a F-RTO sender. When the segment retransmitted due to RTO is acknowledged, the F-RTO sender transmits two new segments. Because several other segments were dropped in the last window, the two new segments trigger duplicate ACKs. As given by the F-RTO algorithm, the arrival of the duplicate ACK as the second acknowledgement following the RTO makes the sender retransmit unacknowledged segments in slow start like the conventional RTO recovery would do. 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, the F-RTO sender enters slow start and retransmits the unacknowledged segments similarly to the conventional RTO recovery, because the two new segments transmitted after the RTO would trigger duplicate ACKs at the receiver.

In a scenario where all segments of the original window have been lost, as presented here, 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 [11]. 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.

[3] In this scenario also the first RTO retransmission happens to be lost, but the second retransmission succeeds. However, the behavior of the algorithms would be the same also in the case with a successful first RTO retransmission.



(a) Conventional RTO recovery.



(b) F-RTO recovery.

Figure 4.3: Comparison of the conventional RTO and F-RTO after a burst loss.

4.3.4 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 [20], 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 to a dropped packet, as out-of-order segments trigger duplicate acknowledgements. 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 reverts to conventional RTO recovery and retransmits the unacknowledged segments. If the delayed packets trigger new acknowledgements that 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.

4.4 Performance Analysis

In order to validate the discussion in Section 4.3, 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 4.3 may occur. We compared the F-RTO performance to the performance achieved with the conventional RTO recovery, both with SACK TCP and with NewReno TCP. In addition, we conducted experiments with the Eifel algorithm [111].

4.4.1 Test Arrangements

The general test setup is illustrated in Figure 4.4. We emulate the wireless link and the last-hop router by using a real-time wireless network emulator [100]. The end hosts are Linux systems, in which we implemented the F-RTO algorithm. The fixed link is an isolated LAN that is connected to the remote host and to the network emulator.

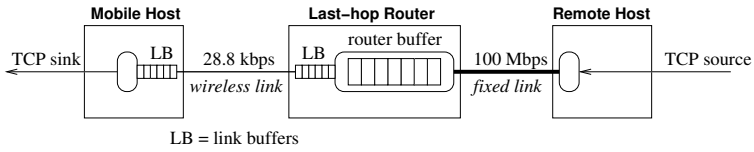


Figure 4.4: Test setup.

We selected the link parameters to approximate the properties of a typical wireless wide-area networking system, such as the *General Packet Radio Service (GPRS)* [35]. 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, which is sufficient for storing the output link's bandwidth-delay product's worth of data, to be able to keep the wireless link utilized on short periods of disruption in data transfer, for example after link-level retransmissions[4]. In addition to the router buffer, the emulated wireless link uses a link send buffer and a link receive 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 frames that have not yet been acknowledged as received, and the link receive buffer collects out-of-order frames for delivering them to 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 use three different experimentation setups that correspond to the scenarios presented in Section 4.3. 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 re-

[4]We applied a MTU size of 296 bytes in these tests due to the slow bottleneck link used in our setup. This is in accordance with the IETF recommendation [41].

liability. During the delay the wireless link receiver does not deliver any packets forward. 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 [102]. 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 that F-RTO does not cause harmful effects on non-spurious RTOs when the retransmission timeouts occur due to lost segments. These timeouts occur mainly when retransmissions are lost, since lost original packets are usually recovered by fast retransmit.

- **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. This is a common loss pattern in some scenarios with wireless hosts, and often results in a retransmission timeout.

In each of the scenarios presented above we test five TCP variants based on the TCP implementation of Linux kernel version 2.4.7 [147]. For the purposes of the experiments, we disabled the ratehalving algorithm used by default in the Linux TCP implementation, and made small modifications to implement the Eifel algorithm as it has been defined by its authors [111][5]. In addition, we modified the SACK loss recovery to behave similarly to the conservative algorithm recently published by the IETF [23]. Firstly, we test a SACK TCP [117] with the conventional RTO recovery, and with the F-RTO recovery. Secondly, we do experiments with a NewReno TCP [56] with both conventional and F-RTO recovery algorithms. Finally, we test a TCP variant using the TCP timestamp option both with

[5]The recent Linux kernels implement a timestamp-based detection algorithm similar to Eifel, but there are a couple of minor differences to the algorithm described in the original article.

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 reverts the changes made on the congestion window and `ssthresh` when it detects a spurious timeout from the timestamps. The limited transmit algorithm [7] 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 better interactive response times [121], although this is a factor not significant in our tests. For each scenario and TCP variant the experiment is repeated 30 times.

4.4.2 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 repetitions 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 4.5 shows the box-plot diagrams of the throughput measured with different TCP variants. Additionally, Table 4.1 shows the median values for the connection statistics described above. The results show that using F-RTO improves performance over the conventional 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, resulting in improved throughput with the F-RTO algorithm. Apart from small random variance, there is no significant difference between the SACK TCP and the NewReno TCP, when RTOs are triggered by excessive delays.

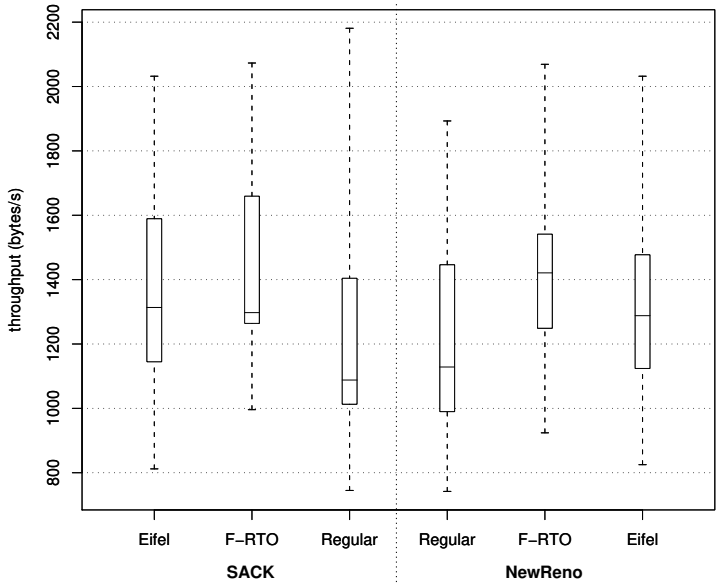


Figure 4.5: TCP performance with different variants with excessive delays on the link.

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 an action to take.

Table 4.1: Results of the tests with sudden delays. The median values of 30 repetitions.

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

Table 4.2: Results of the tests with packet errors. The median values of 30 repetitions.

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

Packet losses

Figure 4.6 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 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 that triggered the RTO, it can usually proceed with transmitting new data. Table 4.2 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.

Eifel TCP using SACK and TCP timestamps has a lower throughput than SACK TCP without timestamps. However, a closer examination of the TCP

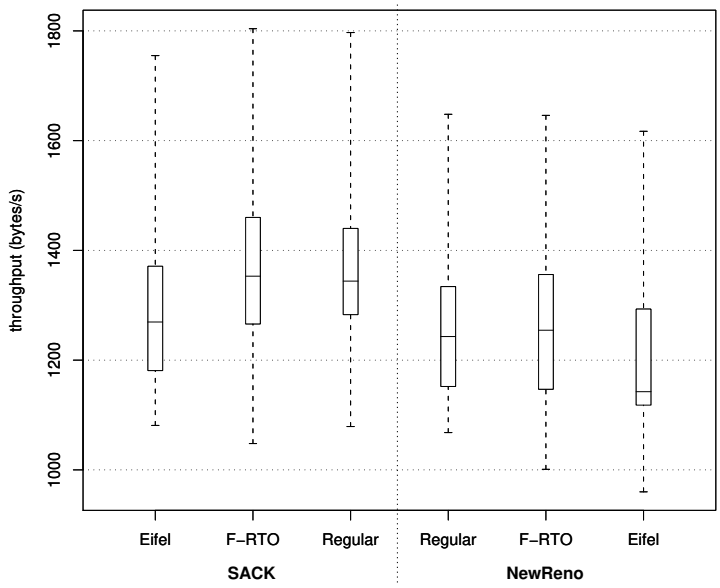


Figure 4.6: TCP performance with different variants with packet drop probability of 0.05.

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.

Bursty losses

Figure 4.7 shows that the TCP performance with F-RTO does not differ significantly from the performance with the conventional RTO recovery when there are link outages. As described in Section 4.3.3, 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 4.3 shows the median connection times and the retransmission statistics for the different TCP variants.

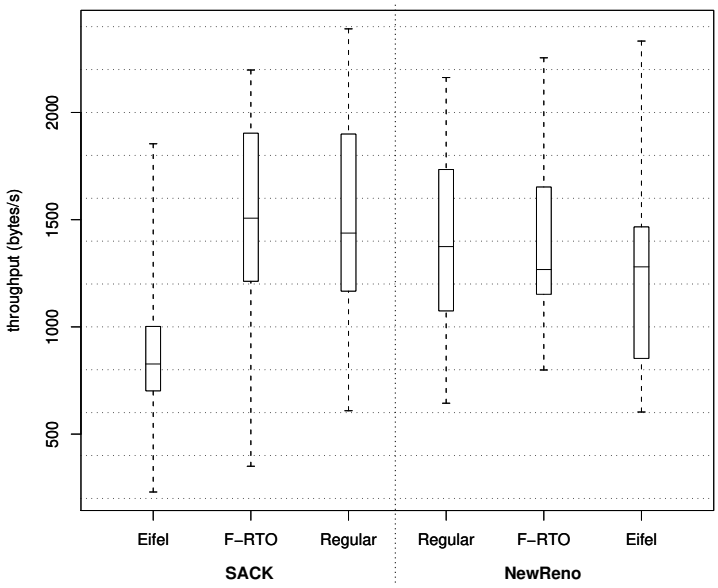


Figure 4.7: TCP performance with different variants with bursty losses on the link.

The test results show that Eifel TCP gives a clearly worse throughput than the conventional 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 ear-

Table 4.3: Results of the tests with bursty losses on the link. The median values of 30 repetitions.

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

lier data segment[6]. 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

[6]The specification for TCP round-trip time measurements [25] requires that the echoed timestamp should correspond to the most recent data segment that advanced the window

retransmissions, and we do not believe it to show up, if some other mechanism was used for indicating spurious retransmissions instead of TCP timestamps[7]. Furthermore, our preliminary tests show that if F-RTO recovery is combined with Eifel, the problem described above does not appear.

4.4.3 Fairness towards conventional TCP

We expect the connections using the F-RTO algorithm to be friendly towards the TCP connections with conventional 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 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 4.4, 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 conventional RTO recovery. Connection set 2 has another three TCP connections that use F-RTO in test A, and the conventional 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 that 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

[7]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 have other negative implications to round-trip time measurement.

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 4.8 on page 82 shows the results of tests with additional delays. Figure 4.8(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 4.8(b) gives the throughput of the slowest connections in the connection sets. The boxplot 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 conventional 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 conventional RTO recovery.

4.5 Summary

In this chapter we have shown 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 that 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 the F-RTO algorithm effectively avoids unnecessary retransmits, it obviates the NewReno "bugfix" rule that disables fast retransmit during an 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 being clocked by incoming acknowledgements and by sending data at an equal rate as the conventional TCP. 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.

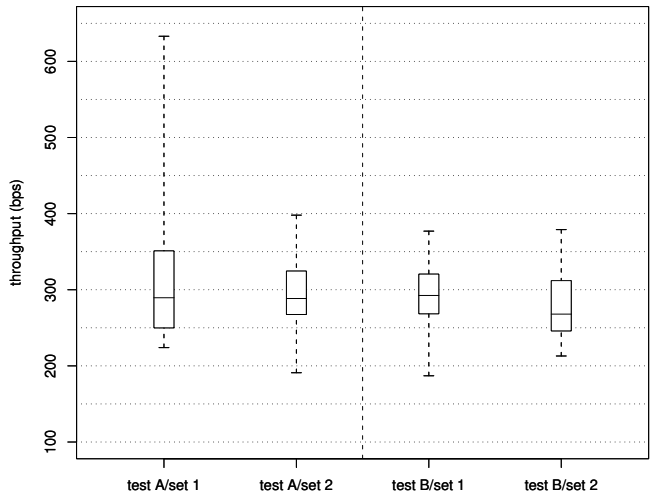
We compared F-RTO with the Eifel algorithm and concluded that their performance is similar in the majority of cases. The Eifel algorithm can perform better than F-RTO, if packet reordering or packet losses are present for the two next segments following the RTO. Eifel makes the detection of spurious RTO already on

the first incoming ACK after RTO, whereas F-RTO is able to detect the spurious RTO after two acknowledgements have arrived. However, in most of the cases F-RTO avoids unnecessary retransmissions as successfully as Eifel does, and after one window has been transmitted, it has delivered the same amount of data as Eifel. On the other hand, while making the detection at the first incoming acknowledgement, the Eifel algorithm can end up at a false positive conclusion, if the outstanding acknowledgements and data segments have been lost in the same window due to a loss burst. The DSACK-based algorithms are not directly comparable with F-RTO, since they are not able to detect spurious retransmissions until one window of data has been transmitted and therefore cannot be used for avoiding unnecessary retransmissions.

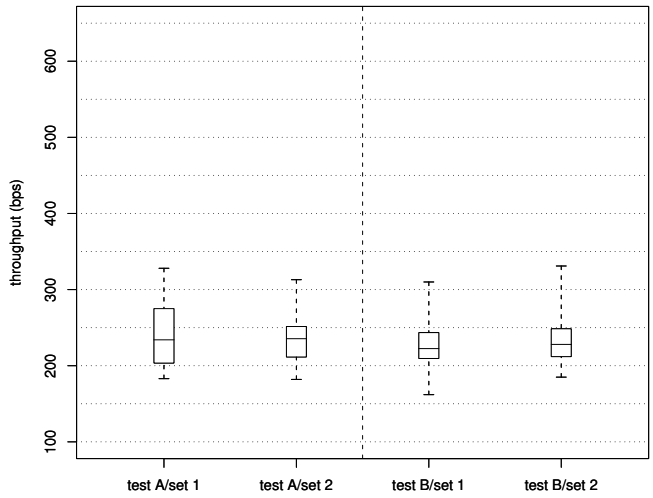
In addition to the experimentation setup chosen for this chapter, we have conducted less systematic tests on F-RTO during its development and testing process in a variety of different network characteristics to verify that it does not harm the TCP behavior under different circumstances. There are also tests conducted by some commercial vendors who have decided to adopt F-RTO. NTT DoCoMo has published their results [167, 72], and we are also aware of some unpublished positive results. For example Microsoft discussed their positive experiences with F-RTO in an IETF meeting[8].

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. We have also contributed our implementation to the Linux kernel development, and F-RTO is included in the Linux kernels starting from version 2.4.21, and in all Linux 2.6 kernels. Therefore, it is possible for the reader using Linux to try out the F-RTO algorithm by getting a recent version of the Linux kernel.

[8]Presentation slides from Microsoft are available at the IETF online proceedings at `http://www3.ietf.org/proceedings/07mar/slides/tsvarea-3/sld9.htm`



(a) Fastest connections.



(b) Slowest connections.

Figure 4.8: Effect of parallel connections on TCP performance. Test A includes three F-RTO connections, test B uses only conventional RTO recovery.

CHAPTER 5

Enhancements on F-RTO

Two problem cases were identified concerning the F-RTO algorithm in Chapter 4, although they did not have a meaningful effect on performance in the presented experiments. First, the presence of packet reordering can cause the F-RTO sender to enter the conventional RTO recovery with go-back-N retransmissions even if the RTO was spurious. Similarly, duplicate ACKs during TCP fast recovery often prevent the F-RTO algorithm from working. Second, if the sender does not have new data to transmit, or the receiver's advertised window does not allow the sender to transmit new data in F-RTO algorithm step 2, the sender may not proceed to detect whether the RTO was spurious. We discuss both of these cases and possible solutions for them in this chapter.

In this chapter we present an enhancement of the F-RTO logic that uses the information in the TCP *Selective acknowledgments (SACK)* option [117], if available. By using SACK, the F-RTO algorithm may detect spurious RTOs that occur during loss recovery, which is not possible with the basic F-RTO algorithm. Because packet losses may occur frequently on a congested network, this is a considerable benefit.

Another topic investigated in this chapter are actions taken on congestion control after a spurious retransmission timeout. In our measurements in Chapter 4 we reduced the congestion window and slow start threshold to half after a spurious RTO was detected. We considered this conservative enough in various possible scenarios. However, reducing the congestion control parameters may not be the best alternative in all situations, especially when the connection path has a high bandwidth-delay product. In the research and standardization forums there have been different suggestions from entering slow start after a spurious RTO [157] to fully reverting the congestion control parameters to the state preceding the spurious RTO [110].

The rest of this chapter is organized as follows. Section 5.1 describes the SACK-enhanced version of the F-RTO algorithm. Section 5.2 discusses the different response alternatives to a spurious RTO that will be studied in this chapter. Section 5.3 describes the measurement environment we are using to test the different response alternatives. Section 5.4 describes the results of the measurements. Section 5.5 discusses some additional considerations on SACK-based F-RTO, and Section 5.6 gives a brief summary of this chapter.

5.1 Detecting Spurious RTO with TCP SACK Option

Although we use the F-RTO algorithm for detecting spurious RTOs, most of the congestion control related considerations in this study should be applicable to other detection methods, such as the Eifel detection algorithm. The idea of the F-RTO algorithm is that, if the sender gets an acknowledgment after an RTO for a segment that was not yet retransmitted due to the RTO, the segment or the corresponding acknowledgment must have been outstanding in the network while the RTO occurs, and the RTO has likely been spurious. If no such indications appear within two round-trip times after the RTO, it is not declared spurious. As described in Chapter 4, the F-RTO algorithm is also robust against packet losses.

If the RTO is not spurious, but caused by data loss, a successful RTO retransmission results in advancement of the cumulative ACK point to the first non-received segment. Because the F-RTO sender continues by transmitting previously unsent data, a duplicate ACK follows, since the segments appear at the receiver as out-of-order segments. This causes the F-RTO sender to retransmit the unacknowledged segments in the conventional way.

As discussed in the beginning of this chapter, the duplicate acknowledgements from packets that have been reordered in the network but not lost can prevent the F-RTO algorithm from working. A potential solution for enhancing F-RTO in the face of reordering comes with the availability of the TCP SACK option. By using the information in the SACK blocks after an RTO, the TCP sender can recognize acknowledgments for segments transmitted before the RTO and thus detect a spurious RTO according to the principles given in Section 4.2 even if there were duplicate ACKs arriving. Furthermore, availability of the SACK information makes it possible to better utilize the F-RTO algorithm during fast recovery periods. In this chapter we study the performance benefits of applying the SACK information in the F-RTO algorithm.

If the TCP endpoints have the SACK option available, many of the F-RTO

problem cases related to duplicate ACKs can be avoided. The SACK-enhanced F-RTO algorithm is implemented at the sender as follows.

1. *When the RTO expires, retransmit the first unacknowledged segment.* Set variable `send_high` to indicate the highest segment transmitted so far. Following the recommendation in SACK specification [117], reset the SACK scoreboard.

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

   (a) if a cumulative ACK acknowledges a sequence number (smaller than send_high, but larger than SND.UNA) transmit up to two new (previously unsent) segments and proceed to step 3. If the TCP sender is not able to transmit any previously unsent data – either due to receiver window limitation, or because it does not have any new data to send – it is possible to apply some of the alternatives for handling window-limited cases discussed in Section 5.5.2. The sender can also simply refrain from entering step 3 of this algorithm, and continue with slow start retransmissions following the conventional RTO recovery algorithm. However, in the latter case the spurious retransmission timeout remains undetected.

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

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

   (a) if the ACK does not acknowledge sequence numbers above `send_high` AND it acknowledges data that was not acknowledged earlier (either with cumulative acknowledgment or using SACK blocks), declare the timeout spurious and continue transmitting new data. The retransmission timeout can be declared spurious, because the segment acknowledged with this ACK was transmitted before the timeout.

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

If the retransmission timeout is declared spurious, the TCP sender continues by sending new previously unsent data, and applies one of the congestion control alternatives described in Section 5.2. If during the above-mentioned steps after the retransmission timeout there are unacknowledged holes between the received SACK blocks, those segments are retransmitted similarly to the conventional SACK recovery algorithm [23]. Similarly to the basic F-RTO algorithm, if the SACK-based algorithm declares the RTO spurious, `send_high` is set to SND.UNA, thus allowing fast recovery on incoming duplicate acknowledgments. This is possible because the problem of multiple fast retransmits cannot occur in this case, as discussed in Section 4.2. The SACK-based recovery algorithm specified by the IETF uses the *RecoveryPoint* variable for this purpose [23]. The Linux implementation of this algorithm applies the *Use It or Lose It*-type burst avoidance similarly to the basic F-RTO algorithm.

The SACK-based algorithm allows declaring an RTO spurious also when a duplicate ACK arrives, if the SACK blocks indicate that some non-retransmitted data segments have arrived at the receiver. Therefore it should enhance TCP performance in cases where there was packet reordering or packet loss in addition to the delay spike that caused the spurious timeout. Especially, the SACK-based algorithm allows detecting a spurious RTO also during the TCP loss recovery phase.

5.2 Responding to Spurious RTO

Once a spurious RTO is detected, the TCP sender should decide on actions following the spurious RTO. Obviously, no packets should be assumed lost, but the sender should continue transmission as if the RTO never occurred. A more challenging consideration is, how the TCP sender should adjust its congestion control parameters and retransmission timer estimate. It has been suggested that the congestion control parameters are reverted to the state preceding the spurious RTO [111].

Even though a related study shows that fully reverting the congestion control state after a spurious RTO is the most efficient alternative in an environment with relatively high bandwidth and delay [65], we are interested in studying what are the performance effects of doing so when the wireless link bandwidth is lower and the wireless access network is subject to competing traffic consisting of multiple parallel TCP transfers. Therefore, we evaluate the performance effects with a more careful alternative of reducing the congestion window after a spurious RTO. Additionally, we study resetting the congestion window to one segment and continuing with slow start after a spurious RTO that has been proposed in the TCP-related discussion forums.

We are studying three alternatives for handling congestion control after detecting a spurious RTO. The congestion window and slow-start threshold are adjusted according to the descriptions below, if the F-RTO algorithm declares the RTO spurious in step (3a) of the algorithm presented above. Similar adjustments can also be made in the basic F-RTO algorithm.

A related paper [65] has also selected similar congestion control alternatives under study in a simulation environment with considerably higher bandwidth and router capacity. We believe none of the alternatives below violates the TCP congestion control principles, and all of them are less aggressive than a conventional TCP sender that does not detect a spurious RTO, and thus unnecessarily retransmits segments in slow start after the spurious RTO.

- **CC 1: Reduce congestion window and slow start threshold to half**. This is similar to what is done when a packet is lost, or when an Explicit Congestion Notification [136] arrives at the sender. When implementing this alternative, a spurious RTO is taken as one kind of congestion notification, and the TCP sender reduces its transmission rate. It could also be thought that a spurious timeout caused by an unexpected delay spike contributes to a minor transient congestion peak at the router before the blocked link, because the incoming acknowledgments of the outstanding segments trigger transmission of new packets at the sender.

- **CC 2: Revert congestion control**. This response alternative does not take the spurious RTO as a congestion notification, but restores the transmission rate to the state preceding the spurious RTO. After detecting a spurious RTO, TCP slow start threshold and congestion window are set to the earlier values stored when the RTO occurred. However, the sender avoids sending bursts of packets due to increasing the congestion window by limiting the congestion window to be no more than three segments over the amount of outstanding data. We are interested in finding out how much this alterna-

tive increases the number of congestion losses in the network, and on the other hand, whether this response alternative improves the TCP throughput regardless of the increased level of congestion.

- **CC 3: Reset congestion window to one segment but set slow start threshold to the value before RTO, if larger**. This is similar to the traditional way of adjusting the congestion window after an RTO, with the exception that the TCP sender does not retransmit segments after detecting a spurious RTO. The TCP sender awaits acknowledgments for all outstanding segments before it continues transmission in slow start. This is expected to be a useful action in cases where the spurious RTO is associated with a change in the network conditions, such as when a wireless hand-off takes place.

5.3 Test Methodology

When inspecting the differences between different congestion control alternatives, we are primarily interested in the following performance metrics.

- **Throughput of the TCP connections** is often the most important performance metric for the end-user. Because we use several concurrent TCP connections in our performance tests, we report the lowest and highest throughput measured from parallel TCP connections. Throughput of the slowest TCP connection is often the most interesting, because it shows the negative effects of congestion and indicates the time taken to transmit all of the data. Additionally, distance between the slowest and the highest throughput gives some understanding about the fairness between the parallel TCP connections.

- **Number of packet losses**. Since in this study we assume the wireless link-layer protocol to be reliable, all packet losses are due to congestion. Therefore the number of packet losses indicates the level of congestion in the network.

- **Number of retransmissions** often depends on the number of packet losses, but because the link is prone to spurious retransmission timeouts, there may be substantial differences between the number of retransmissions and the number of actual packet losses. The number of retransmissions is interesting for the end-user not only because the retransmissions degrade the data throughput, but because the wireless data user is often charged based on the

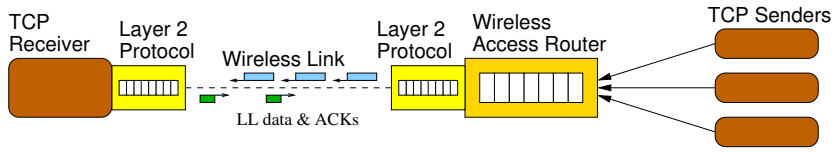


Figure 5.1: Model of the simulation environment.

amount of bytes transmitted, regardless of whether the data is TCP retransmissions or original transmissions. The packet losses/retransmissions ratio tells how much there have been unnecessary retransmissions. The lower the ratio, the more inefficient the TCP sender is in terms of unnecessary retransmissions.

The performance measurements are conducted using Linux end-hosts. The wireless environment is simulated using the *Seawind* real-time emulator [100]. Seawind captures the IP packets transferred between the end hosts and simulates the wireless network characteristics according to given parameters by delaying, queueing and dropping packets in a real-time fashion. We have selected network characteristics that roughly resemble the expected characteristics of the 2.5G and 3G networks, but since the detailed behavior of those networks depend on various configuration parameters, we have our model at a rather generic level.

Figure 5.1 illustrates the network setup we have in our performance tests. There are 1–3 TCP senders in the fixed network that transmit data concurrently. Each TCP sender transmits a 200 KB-sized file to the wireless receiver using packet MTU of 1500 bytes[1]. We model different wireless link bandwidths between 28 Kbps and 384 Kbps, whereas the fixed network between the TCP senders and the wireless access router is a 100 Mbps LAN. The wireless link has one-way propagation delay of 200 ms. In total, packet round-trip times are usually between 500 ms and 2000 ms, depending on the packet queue length at the access router and the available wireless bandwidth.

There are two types of buffering in the access network next to the wireless link. First, there are IP-level router buffers that are configured to hold 7–30 packets, depending on the bandwidth-delay product in the test scenario in question. The router buffer capacity is limited by the packet count, regardless of the size of the packets. Second, there are layer 2 buffers that are sized according to the

[1]We assume a larger packet MTU than that used in Chapter 4, because we use larger link bandwidths in the experiments conducted in this chapter. 1500 bytes is a common value for MTU used in various link technologies, including those based on Ethernet.

bandwidth-delay product of the wireless link. Thus, the total buffer capacity in front of the wireless link is between 18–80 KB, depending on the test scenario.

We model two kinds of delay sources on the wireless link that have been under discussion in the TCP research community. Delay of *type I* is to model a momentary outage on the wireless link with a link level ARQ. During this period the packets on the wireless link are lost. The link layer sender retransmits the packets until they are successfully transmitted. The probability of a delay event is 1 % per packet, and the length of a link outage is exponentially distributed with a mean length of 3500 ms.

Delay of *type II* stands for a wireless handover and the related events in the wireless access network. After a handover and the related short delay the link bandwidth is randomly selected to either 28.8 Kbps, 64 Kbps, or 128 Kbps, reflecting the different conditions in different wireless cells. In these tests the delay spike of random length may occur with a change in the available bandwidth. There are no losses on the wireless link in addition to those that are caused by congestion. The delay length is exponentially distributed with a mean length of 3500 ms. We have a simple model of rather quick random movement of a wireless host. The host remains in one simulated cell for a random time; in a 2-second period there is a 30 % probability that the host moves to another cell and the bandwidth changes.

In our tests we primarily use a modified version of the Linux kernel that implements the standard retransmission timer algorithm [127]. As discussed in Chapter 3, the retransmission timer in the unmodified Linux kernel implementation contains a few enhancements that are expected to improve the performance when the packet round-trip times are highly variable. According to our initial tests this indeed appears to be the case, but a more detailed study of the different retransmission timer estimators remains future work.

5.4 Test Results

We evaluated the regular SACK TCP and the SACK-enhanced F-RTO with three different congestion control variants (CC 1, CC 2, and CC 3) as described in Section 5.2. This section describes the most interesting findings in various tests made with different network parameters. Mean values of 30 replications are shown in the tables below. The tables show the throughput of the slowest and the fastest TCP connection in bytes per second, the number of packet losses at the wireless access router, and the number of retransmitted segments.

Table 5.1 shows results of two parallel TCP connections transmitted over a 128 Kbps link that occasionally has delay spikes of delay type I described in Section 5.3. In this test, the IP router buffer size was 7 packets. As expected, the

regular TCP suffers from a substantially larger number of unnecessary retransmissions and thus results in lower throughput. F-RTO helps to avoid most of the unnecessary retransmissions, although some are still present, because the first retransmission triggered by the RTO is often unnecessary also with F-RTO.

Table 5.1: Two TCP connections over 128 Kbps link, 7 IP buffers.

TCP variant	**Tput low (B/s)**	**Tput high (B/s)**	**Losses**	**Rxmits**	**Losses / Rxmits**
Regular	4067	4891	25.3	50.1	0.50
CC 1	4602	5293	18.6	25.2	0.74
CC 2	4480	5103	26.6	31.7	0.84
CC 3	4427	5052	25.2	32.6	0.77

CC 1 halves the congestion window and the slow start threshold after a spurious RTO, and thus results in least congestion losses. CC 1 also has the best overall throughput, indicating that saving in the number of packet losses is more useful in terms of throughput than reverting the congestion window to its full size. In these tests the pipe capacity available per connection was rather small, so the advantage gained by reverting the congestion window was not meaningful.

With the workload of one TCP connection over the 128 Kbps link CC 2 and CC 3 yield better performance than CC 1, although they cause more packet losses. With only one TCP connection the congestion losses are not as bad a problem for the performance as the underutilization of the link when the congestion window is reduced after a spurious RTO.

Table 5.2 presents the results with similar link setup to that in Table 5.1, but with an IP router buffer size of 30 packets, that is, larger than what it recommended for the wireless link in our model, considering its bandwidth-delay product. With larger router buffer the packet round-trip times are generally higher due to increased queueing delays. This makes the retransmission timer more conservative and causes less spurious RTOs due to delay spikes. In these tests, CC 3 results in the best performance and least packet losses. With a large router buffer, the TCP sender can carry on with slow start after the spurious RTO for several round-trip times before suffering from packet losses due to congestion. On the other hand, the narrow link between the sender and the receiver can be fully utilized by a congestion window of 6 segments that can be achieved in a few round-trip times during slow start.

Table 5.3 shows the results with a link bandwidth of 384 Kbps, router buffer

Table 5.2: Two TCP connections over 128 Kbps link, 30 IP buffers.

TCP variant	Tput low (B/s)	Tput high (B/s)	Losses	Rxmits	Losses / Rxmits
Regular	4600	5358	22.0	47.3	0.47
CC 1	4864	5485	23.5	27.6	0.85
CC 2	4885	5409	28.6	33.1	0.86
CC 3	5256	5847	22.9	28.5	0.80

of 30 packets, and three parallel TCP connections. The link had delays of type I, modeling sudden link outages. Again, the regular SACK TCP is very inefficient due to a large number of unnecessary retransmissions. CC 1 has the least packet losses, but even though CC 2 has almost twice as many packet losses, it can achieve slightly better throughput than CC 1 both for the fastest and the slowest connection. With CC 2 the difference between the fastest and the slowest connection is larger than with CC 1, which suggests that CC 2 can cause unfairness between the different TCP connections. With CC 1 the full link capacity is not efficiently used after a spurious RTO, because the sender reduces the congestion window. Using slow start after a spurious RTO gives the worst results, because in that case the large link capacity is used rather inefficiently. Furthermore, CC 3 has more congestion losses than CC 1 due to slow start overshoot [141, p. 11].

Table 5.3: Three TCP connections over 384 Kbps link, 30 IP buffers.

TCP variant	Tput low (B/s)	Tput high (B/s)	Losses	Rxmits	Losses / Rxmits
Regular	4939	6131	29.5	104.2	0.28
CC 1	5622	7073	22.7	37.0	0.61
CC 2	5767	7512	40.0	55.4	0.72
CC 3	5223	7082	30.7	46.5	0.66

Table 5.4 shows the performance metrics when transmitting two parallel TCP connections over a bottleneck link with variable bandwidth of 28.8 Kbps, 64 Kbps, or 128 Kbps. In this scenario we had type II delays, i.e. when a delay spike occurs, the link bandwidth may change at the same time. In these tests CC 3 that does slow start after a spurious RTO is the most efficient alternative. CC 1 that

decreases the congestion window and slow start threshold after a spurious RTO has the least congestion losses, but it uses the link capacity inefficiently, especially when the bottleneck link bandwidth increases after a delay spike. CC 2 that reverts the congestion control parameters to the earlier values utilizes the link more efficiently, but causes most congestion losses.

Table 5.4: Two TCP connections over link with bandwidth that changes between 28.8 Kbps, 64 Kbps, and 128 Kbps.

TCP variant	**Tput low (B/s)**	**Tput high (B/s)**	**Losses**	**Rxmits**	**Losses / Rxmits**
Regular	2660	3305	22.6	38.1	0.59
CC 1	2587	3535	19.0	22.5	0.84
CC 2	2654	3590	21.5	25.4	0.85
CC 3	2878	3789	20.6	24.5	0.84

We made a set of tests with the basic F-RTO with three parallel connections to compare its performance with the SACK-enhanced F-RTO. Table 5.5 shows that the basic variant results in slightly reduced performance, when compared to results in Table 5.3 made with the same network parameters but using SACK. Because of the undetected spurious RTOs that occur during the loss recovery phase, the basic F-RTO has about 20 % more unnecessary retransmissions than the SACK-enhanced F-RTO. Otherwise the conclusions are similar to those made for Table 5.5.

Table 5.5: Basic F-RTO over 384 Kbps link.

TCP variant	**Tput low (B/s)**	**Tput high (B/s)**	**Losses**	**Rxmits**	**Losses / Rxmits**
Regular	4939	6131	29.5	104.2	0.28
CC 1	5514	7086	31.2	52.1	0.60
CC 2	5899	7420	31.5	48.8	0.65
CC 3	5359	6466	23.9	43.9	0.54

5.5 Additional Considerations

F-RTO can also be used with the *Stream Control Transmission Protocol (SCTP)* [153] that uses a SACK-based retransmission mechanism. This section discusses the related issues on applying F-RTO on SCTP. We also discuss some details that need to be considered when spurious timeout occurs during fast recovery, and possible alternatives a TCP sender can try, if it cannot transmit new segments in F-RTO algorithm step 2 when being limited by the receiver's advertised window, or when the sender does not have new data to send.

5.5.1 SACK-enhanced F-RTO and Fast Recovery

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

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

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

5.5.2 Discussion of Window-Limited Cases

When the advertised window limits the transmission of two new previously unsent segments, or there are no new data to send, the default option in F-RTO algorithm step (2a) is that the TCP sender continues with the conventional RTO recovery algorithm. The disadvantage is that the sender may continue unnecessary retransmissions due to possible spurious timeout. This section briefly dis-

cusses the options that can potentially improve performance when transmitting previously unsent data is not possible.

- The TCP sender could reserve an unused space of a size of one or two segments in the advertised window to ensure the use of algorithms such as F-RTO or Limited Transmit [7] in window-limited situations. On the other hand, while doing this, the TCP sender should ensure that the window of outstanding segments is large enough for proper utilization of the available pipe.

- The TCP sender can use additional information if available, for example TCP timestamps with the Eifel Detection algorithm, for detecting a spurious timeout. However, Eifel detection may yield different results from F-RTO when ACK losses and an RTO occur within the same round-trip time, as discussed in Chapter 4.

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

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

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

5.5.3 Using F-RTO with SCTP

SCTP is a reliable transport protocol that has some advanced features compared to TCP. With a modular packet format, SCTP is easier to extend with new fea-

tures than TCP, which is limited by the 40 bytes of TCP option space. SCTP preserves the upper layer message boundaries instead of transferring a continuous byte stream, and it can support multi-homing of an end host, which improves the robustness of connections. SCTP also supports a partial reliability mode, which is more suitable for data streams with real-time requirements. SCTP's characteristics are ideal for signaling protocols, and it has been initially used for SS7 telephony signaling transport [44, Chapter 14]. There is also much signaling that needs to be taken care of by the protocol layers above transport, for example related to instant messaging and voice-over-IP telephony. Therefore, in these cases SCTP could also be used in mobile terminals that nowadays have multiple radio access interfaces, for example, for SIP signaling [139]. In these cases the problems with using TCP over wireless links are also relevant for SCTP.

SCTP has similar retransmission algorithms and congestion control to TCP. However, some of the terminology and details are slightly different. A single upper-layer message is transmitted in *data chunk*. One SCTP packet can include several data chunks among other SCTP control information that can be included in chunks of other types. SCTP's retransmission timer is called *T3-rtx timer*. The sequence numbers in SCTP are called *Transmission Sequence Number (TSN)*. One TSN per data chunk is assigned, which differs from TCP that assigned one sequence number per byte of transmitted data. SCTP uses a selective acknowledgment mechanism, and the SCTP receiver is also able to report receiving duplicate TSNs with a mechanism similar to DSACK in TCP [61].

The SCTP T3-rtx timer for one destination address is maintained in the same way as the TCP retransmission timer, and after a T3-rtx expires, an SCTP sender retransmits unacknowledged data chunks in slow start like TCP does. Therefore, SCTP is vulnerable to the negative effects of the spurious retransmission timeouts similarly to TCP. Due to similar RTO recovery algorithms, F-RTO algorithm logic can be applied also to SCTP. Since SCTP uses selective acknowledgments, the SACK-based variant of the algorithm is recommended, although the basic version can also be applied to SCTP. However, SCTP contains features that are not present with TCP that need to be discussed when applying the F-RTO algorithm. A recent paper evaluates the effects of spurious retransmissions on SCTP [107].

SCTP associations can be multi-homed. The current retransmission policy states that retransmissions should go to alternative addresses. This means that the retransmission may follow a significantly lower latency path than the original transmissions. 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. If the timeout was caused by a delay spike, and it was spurious in that respect, a suitable response is to continue by sending new data. However, if the original transmission was lost, fully reverting the congestion control parameters is too aggressive. Therefore, taking conservative actions on congestion control is recommended, if the SCTP association is multi-homed and retransmissions go to alternative addresses[2]. The information in duplicate TSN notifications can then be used for reverting congestion control, if desired [21]. Note that the forward transmissions made after RTO in F-RTO algorithm step (2a) should be destined to the primary address, since they are not retransmissions.

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

5.6 Summary

We presented a SACK-based enhancement to the F-RTO algorithm and evaluated its use with three different alternatives for congestion control after a spurious retransmission timeout. Our results show that even though the basic F-RTO performs rather well under delay spikes, the SACK-enhancement improves the performance when spurious RTOs occur during TCP fast recovery.

The general trend between the three congestion control alternatives evaluated was that when using a narrow link with appropriate buffer sizes, reducing the congestion window and the slow start threshold lowers the number of congestion-related packet losses and improves the overall performance. Reverting the congestion control parameters improves the performance due to better link utilization only at the highest link bandwidths tested. Going into slow start after a spurious RTO gives good results when the link bandwidth varies during TCP connection or when there are large buffers available to handle the instantly increasing queue length caused by slow start.

[2]The same scenario is also possible in principle with TCP, when a vertical hand-off is done to a low-latency link. If the original segments are sent to a high-latency path, and the RTO retransmission is sent to a path with significantly lower latency, it is possible that the acknowledgment of the RTO retransmission arrives before the acknowledgments of the original segments.

Our test results show that selecting the most efficient response to spurious RTO is not an easy task when the real network characteristics are unknown to the TCP sender. However, our results support taking conservative actions after a spurious RTO when the wireless link bandwidth is not very high, because on the slower links reverting the congestion control state does not improve the TCP throughput significantly. It is possible, that a hybrid solution between the different congestion control alternatives presented in this chapter would result in acceptable performance in different scenarios. One such alternative might be to slightly reduce the congestion window while keeping the slow start threshold at the level it was when the spurious RTO occurred.

Finally, we discussed the applicability of F-RTO in a couple of special cases: when RTO occurs during SACK-based fast recovery, and when the normal F-RTO algorithm is prevented by the TCP receiver window. We also discussed how F-RTO could be applied with the SCTP protocol. Because SCTP uses similar congestion control and retransmission algorithms than TCP, we believe our results would apply also with it. In addition, while most of the experiments have been run in a setup modeled to be similar to the GPRS or EGPRS wireless link characteristics, we believe that F-RTO is useful also in other network environments that might suffer from spurious timeouts. Because F-RTO is based on the use of the existing TCP mechanisms, and it does not depend on any specific characteristics of the lower protocol layers, we believe that if F-RTO was used in links with higher bandwidths, such as satellite links or fixed network environments, similar trends in results could be found on F-RTO's performance than presented in this chapter. Chapter 4 discussed why we believe F-RTO does not harm the sender or the network in any scenario, but at worst performs similarly as the normal TCP would do.

CHAPTER 6

Evaluating Quick-Start for TCP

As discussed in Sections 2.2 and 2.4, TCP is rather conservative in selecting its initial sending rate, and increasing it using the slow-start or congestion avoidance algorithms. This can be problematic on paths with high latencies, such as GPRS. We now take a look at a mechanism that allows TCP to explicitly query for a larger initial sending rate from the routers along the path, called Quick-Start. Although Quick-Start could be used with a number of transport protocols such as *Stream Control Transmission Protocol (SCTP)* [153] or *Datagram Congestion Control Protocol (DCCP)* [99], we mainly consider its use with TCP.

In this chapter we describe the basic protocol and algorithms of Quick-Start, and evaluate a number of design alternatives on a high-speed network with high degree of multiplexing. In Chapter 7 we evaluate Quick-Start in wireless networks with lower bandwidths and host mobility.

The rest of this chapter is organized as follows. Section 6.1 gives a general overview of Quick-Start and motivates its need for paths with high bandwidth or high latency. Section 6.2 details the Quick-Start mechanism and discusses design issues. Section 6.3 discusses the potential costs and benefits of using Quick-Start. Section 6.4 describes the simulation setup used in our study. Section 6.5 illustrates the potential advantages and disadvantages of Quick-Start and shows its performance in specific situations. Section 6.6 discusses the handling of Quick-Start Requests in the routers and evaluates several algorithms that could be employed by routers. Section 6.7 outlines the possible vulnerabilities of Quick-Start to denial-of-service attacks and potential coping techniques. Finally, Section 6.8 offers conclusions and future work.

6.1 Overview

Quick-Start is described in detail in Section 6.2, but the process is generally that a TCP connection sends a packet that includes a Quick-Start Request in an IP option containing the requested sending rate. Each router along the path either agrees with the request, lowers the requested sending rate, or implicitly signals that the Quick-Start option was not approved or processed. The data receiver reports the information received in the Quick-Start Request back to the sender using a Quick-Start Response in a TCP option, and the data sender determines if all of the routers along the path have agreed to the request and sets the sending rate appropriately.

The assumption behind Quick-Start is that routers will only approve Quick-Start requests when they are under-utilized. Thus, Quick-Start should be generally safe to deploy in general purpose networks, with a negligible risk of causing network congestion. However, because Quick-Start requires support from all routers along the path, this could present a high bar to deployment in the general Internet. Possible deployment of Quick-Start could happen in (i) those Intranets and operator networks with large amounts of under-utilized bandwidth and (ii) cellular wireless networks (such as GPRS/EDGE [150]) with long round-trip delays, as discussed in Chapter 2. Based on the investigation presented in this chapter, Quick-Start is expected to be of benefit in both these cases.

As noted above, Quick-Start is, broadly speaking, useful any time a connection is significantly under-utilizing the network path and has the data required to considerably increase the transmission rate. There are a few concrete cases where the connection is likely to be significantly under-utilizing the network path capacity and could benefit from Quick-Start:

- Typically in the beginning of a connection a TCP sender has little if any knowledge of the network path characteristics. Therefore the sender has to probe the path capacity by using slow-start. By applying Quick-Start the slow-start phase could be significantly shortened.

- After the path characteristics are known to have changed significantly, for example due to wireless hand-off. TCP could have a notification either locally, or by using a mechanism such as *Lightweight Mobility Detection and Response (LMDR)* [158] for a mobile end to notify its peer about the link change. After such notification a Quick-Start Request can be sent to resolve the new path capacity.

- After an idle period. It is recommended that TCP Congestion Window Validation [67] is applied after an idle period in transfer to conservatively re-

duce the congestion window. The thinking behind this is that the path status could have changed during the idle period, and since TCP has not actively probed the path capacity, it needs to re-establish the path capacity by applying slow-start again with reduced congestion window size.

While Quick-Start is a component of congestion control, Quick-Start is not a complete congestion control mechanism, and it is not intended as a replacement for TCP's standard congestion control. Quick-Start is also not a Quality of Service (QoS) or resource reservation mechanism. Quick-Start is in fact most effective in those under-utilized environments where congestion control is not the overriding issue, and where QoS mechanisms are needed the least. In the subsequent sections we show this via simulation.

6.2 Quick-Start Protocol Details

Quick-Start is a collaborative effort between end hosts and routers. This section describes the details of Quick-Start, and discusses the Quick-Start requirements. Quick-Start has also been specified in the IETF [54].

6.2.1 Packet Format

Quick-Start Request is an IP option intended to be processed by each router along the connection path. When the receiver gets the Quick-Start Request option, it responds with a *Quick-Start Response* TCP option. Figure 6.1 shows the format of Quick-Start option for IPv4 packet. IPv6 uses a similar packet format as hop-by-hop option.

Quick-Start Rate Request (Rate), uses four bits in the Quick-Start Request header, and four bits are reserved for future use. To allow for a larger range of possible rate values, Quick-Start Rate Request is exponentially encoded to $K * 2^N$ bits per second, where K is selected to be 40 Kbps, and N is the value in the rate request field. Thus, with N=1 the minimum rate to request is 80 Kbps and the maximum rate to request is 1,310,720 Kbps, with N=15. Although this seems a coarse-grained range, we believe it to be sufficient, since Quick-Start is not intended to be a replacement for the normal congestion control mechanisms, but to make a quick rough check if there is a considerable amount of unused bandwidth on the path.

Recently the Quick-Start specification [54] has extended the packet format with a 32-bit nonce, and a function field that allows the sender to report the final rate that was approved for the TCP connection. The routers closer to the

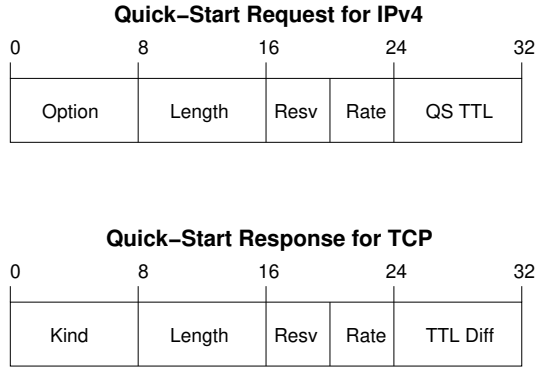


Figure 6.1: Quick-Start packet format.

sender may benefit from this information, since they cannot know if a downstream router has reduced the rate request, and thus make incorrect assumptions on the available bandwidth when making decisions on subsequent Quick-Start requests. These fields were not used in the simulations presented in this chapter, and we will not discuss them any further. However, the problem of unnecessarily high rate requests is discussed shortly.

6.2.2 Quick-Start Processing at the Sender

The Quick-Start *Rate Request* is initialized by the sender to the desired sending rate in bytes per second (Bps). The sender also initializes a *Quick-Start TTL* to a random value and saves the difference between the initial Quick-Start TTL and the initial IP TTL as $TTLDiff$. As discussed in the next subsection, the routers along the network path between the sender and receiver alter the Rate Request, as appropriate. When the Quick-Start Request arrives at the transport receiver, the receiver echoes the rate request back to the sender along with the difference between the Quick-Start TTL and the IP TTL, $TTLDiff'$, in an option in the transport header. Upon reception of an echoed Quick-Start Rate Request the sender verifies that all routers along the path have approved the Quick-Start Request by comparing $TTLDiff$ and $TTLDiff'$. If these two values are not the same then the request was not approved by all routers in the network path and data transmission will continue using TCP's standard algorithms.

When the $TTLDiff$ and $TTLDiff'$ match, the TCP sender then calculates the appropriate congestion window (cwnd) based on the approved sending rate

and measured round-trip time as follows:

$$cwnd = \frac{Rate * RTT}{MSS + H} \tag{6.1}$$

where *Rate* is the approved rate request in bytes per second, *RTT* is the recently measured round-trip time in seconds, *MSS* is the maximum segment size for the TCP connection and *H* is the estimated header overhead for the packets in the connection in bytes. The TCP sender paces out the Quick-Start packets at the approved sending rate over the next RTT[1]. Upon receipt of an acknowledgment for the first Quick-Start packet, the TCP sender returns to ACK-clocked transmission.

Knowing the Rate to Request

One of the problems of Quick-Start is that unnecessary or unnecessarily-large Quick-Start Requests can "waste" potential Quick-Start bandwidth. Because routers must keep track of the aggregate bandwidth represented by recently approved Quick-Start requests (so that the router does not over-subscribe the available capacity), each approved request reduces the chances of approval for subsequent requests. Ideally, a sender should not use Quick-Start for data streams that are not expected to benefit from it, such as those that have only a few packets of data to send. The TCP sender should, in theory, also avoid requesting an unnecessarily high sending rate. However, it can be difficult for the TCP sender to determine how much data will ultimately be transmitted and therefore to form a reasonable rate request. For example, in request-response protocols such as HTTP [18], the server does not know the size of the requested object during the TCP handshake, because it has not received the data request yet. Once the web server does know the requested object, the application would need to determine the size of the object and then inform TCP as to how many bytes will be sent, because the objects are rarely written to TCP socket buffers in a single atomic call. Even if the web server was able to determine the size of the objects, there may still be more data that the web server does not yet know about. Finally, sometimes the application cannot even obtain the size of an object because the object is being read from a pipe or some live source. In Section 6.5.2 we illustrate the problems of not making a reasonably accurate rate request and offer some strategies for coping.

[1]Note that TCPs are required to implement an additional timer for paced transmission when using Quick-Start.

6.2.3 Quick-Start Processing at Routers

A router that receives a packet with a Quick-Start Rate Request has several options. Routers that do not understand the Quick-Start Request option simply leave the option untouched, ultimately causing the Quick-Start Request to be rejected because $TTLDiff'$ will not match $TTLDiff$. Routers that do not approve the request can either leave the Quick-Start Request option untouched, zero the Rate Request, or delete the option from the IP header. Routers that approve the rate in the request decrement the Quick-Start TTL and forward the packet. Finally, a router can approve a rate that is less than the rate in the request by reducing the rate, as well as decrementing the Quick-Start TTL.

Routers should only approve a Quick-Start Request when the output link has been underutilized over some recent time period. In order to approve a Quick-Start rate request, a router generally should know the bandwidth of the outgoing link and the utilization of the link over a recent period of time. At a minimum, the router must also keep track of the aggregate bandwidth recently approved for Quick-Start Requests, to avoid approving too many requests when many Quick-Start Requests arrive within a small window of time. Section 6.6 discusses in more detail the range of algorithms that could be used by routers in approving or denying a Quick-Start request.

Later this chapter speaks of "allocating" capacity, but it is noted that Quick-Start routers do not in fact reserve capacity for a particular flow and then police the usage to ensure that the given flow is able to use the granted capacity. Rather, the router simply tracks the aggregate amount of promised capacity (in the recent past) in an effort not to promise more than the output link can absorb. If, however, a burst of unexpected traffic arrives the Quick-Start "allocations" may prove to be empty promises when the end hosts attempt to use the granted bandwidth and detect congestion.

6.3 Challenges

Practical deployment of Quick-Start would face some real-world challenges. The most significant identified challenges are discussed below.

- **Increased Periods of Congestion.** Quick-Start should be approved only in situations where the network path is under-utilized, thus allowing a connection to quickly use spare capacity. Therefore, the correct use of Quick-Start should not result in increased packet drop rates in the network. In other words, Quick-Start should not cause congestion, but rather should allow

a connection to quickly use the spare capacity in the path. In Section 6.5 we show that proper use of Quick-Start does not increase the aggregate drop rate in a network. However, misconfiguration at Quick-Start routers or some other bug in Quick-Start could introduce inappropriate traffic to congested situations. To mitigate this, such a situation causes a full reset to standard slow start.

- **Misbehaving Nodes and Routers.** Quick-Start may provide new ways for two types of misbehavior. First, misbehaving receivers or routers could try to lead Quick-Start to benefit the connections using Quick-Start. Non-conformant routers or hosts might try to modify the Quick-Start messages to benefit particular connections. For instance, a receiver may increase the rate given in an arriving Quick-Start Request before echoing it back to the sender in an effort to increase the connection's performance. Similarly, a router close to the sender and acting on the sender's behalf could increase the approved sending rate and/or adjust the reported $TTLDiff'$ from the receiver to match the original $TTLDiff$ in an effort to mask the network's lack of Quick-Start support. While it is possible to attempt to misuse Quick-Start, it is not without risk of lower performance since the TCP sender is required to go back to slow-start if the inappropriately high sending rate causes packet losses in the Quick-Start window. In addition, recently additional mechanisms have been added to Quick-Start IETF specification to make misuse more difficult [54]. A second type of misbehavior comes from attackers attempting to prevent legitimate use of Quick-Start. This aspect of Quick-Start is further discussed in Section 6.7.

- **Added complexity at routers and end-nodes.** One of the main costs of Quick-Start is that the required changes to both end-hosts and routers may moderately increase implementation complexity. For end-hosts the additional complexity may be justified by (i) the possible benefits of Quick-Start and (ii) that end hosts often have spare processing capability (although this is not universally true — especially for busy servers). However, the additional complexity at routers can be a difficult issue, since performance and scalability requirements in routers have to be carefully balanced. Packets containing a Quick-Start Request represent an extra burden for routers and could result in extra delay for end-hosts. Of course, all packets would not contain Quick-Start Requests. Additionally, Quick-Start should only be approved in times of under-utilization and therefore the routers may be able to perform an efficient quick check of the utilization and only act on requests when the router is under-utilized (and can likely better absorb the additional

processing requirement).

- **Interactions with Middleboxes.** It is known that there are middleboxes in the current network that drop packets containing known or unknown IP options [118]. This could result in significant delay for connections using Quick-Start requests, as packets using Quick-Start requests would have to be retransmitted without the Quick-Start Request Option (and if the option is transmitted on a SYN segment the initial retransmission timeout of 3 seconds [127] makes this a lengthy process). One consequence is that initial deployments of Quick-Start may be in controlled environments, where it is known that packets with Quick-Start options would be forwarded.

- **IP Tunnels.** Some IP tunneling mechanisms encapsulate the IP packets without decrementing the IP TTL of the IP header. Therefore it is possible that an IP tunnel that is not aware of Quick-Start encapsulates the packet so that the Quick-Start *TTL Diff* does not change. As a result, the Quick-Start request can pass the tunnel without being processed by the routers along the tunnel path, while to the sender it seems that all routers have approved the request. There is no known way to reliably handle Quick-Start Requests on paths with such transparent tunnels. Some common types of tunnels are those used by IPsec [93] and IP in IP encapsulation [128].

- **Deployment.** An additional downside of the Quick-Start approach is that the scheme is not conducive to incremental deployment. Since both end systems and all the routers along some path have to support Quick-Start for the mechanism to work there is quite a high barrier to general use. We expect that initial deployments of Quick-Start would happen within closed networks whereby hosts and routers both have an interest in aiding performance.

6.4 Simulation Setup

In the following sections we use the ns-2 simulator to explore various aspects of Quick-Start. We use a network comprised of three routers, R_1–R_3, arranged in a chain. The two links between the routers have a bandwidth of L_{bw} and a one-way link delay of L_d. Unless otherwise noted, L_{bw}=10 Mbps and L_d=20 msec. The routers employ drop-tail queuing[2] with a maximum queue size of 150 packets.

[2]We believe that drop-tail queueing is used in the majority of the network routers because of its simple and efficient characteristics.

For most simulations, web clients and servers are connected to the ends of the network (to R_1 and R_3) with dedicated 1000 Mbps links with a mean one-way link delay of 12 msec and a maximum delay of 110 msec. The actual link delays are chosen to give a range of round-trip times that matches those from measurements, using the process from [58]. A varying number of web servers, N, are connected to R_1 with a corresponding number of web clients connected to R_3. The measurements presented in the subsequent sections all refer to the traffic from the web servers connected to R_1. We also attach $\frac{N}{2}$ web clients to R_1 and $\frac{N}{2}$ web servers to R_3 to provide background traffic on the return path. When Quick-Start is enabled, all web servers attempt to use Quick-Start. The standard web traffic generator included with ns-2 is used in our simulations, with the following parameter settings: an average of 30 web pages per session, an inter-page parameter of 0.8, an average page size of 10 objects, an average object size of 400 packets and a ParetoII shape parameter of 1.002. We use HTTP/1.0-like transactions, with one web object per TCP connection. These parameters are not picked to match any particular network's traffic distribution, but rather to explore Quick-Start's impact on a wide range of connection sizes. Our web traffic simulations are run for 150 seconds.

In addition, a few simulations make use of a single transfer at a time. These simulations use FTP to transfer a file of given size over the network given above with no reverse traffic present.

Finally, all TCP connections use ns-2's *sack1* TCP variant with an initial *cwnd* of 3 segments (per [9]), an MSS of 1460 bytes, an advertised window of 10,000 segments[3], and the receiver acknowledging each segment. All simulations are repeated 12 times, with averages and standard deviations shown in the graphs.

All simulations presented in the remainder of the chapter use this setup unless otherwise noted.

6.5 Connection Performance

In this section we explore when Quick-Start is and is not of benefit. In addition, we consider how to choose the Quick-Start request size, the implications of Quick-Start on aggregate network traffic and the implications of Quick-Start failures.

[3]This is high enough to make the advertised window a non-issue in our simulations.

6.5.1 Ideal Behavior

In an ideal Quick-Start scenario over an under-utilized network path, the TCP sender would be able to transmit as much of its data in the initial congestion window as the spare network capacity can absorb. Figure 6.2 illustrates an example of the ideal Quick-Start behavior by displaying time-sequence plots of two connections[4]. The first connection is a standard TCP connection that uses slow start to begin transmission (with an initial *cwnd* of 3 segments, per [9]). The second connection on the plot shows a case where an approved Quick-Start Request allows the sender to transmit 25 of its 30 packet transfer in the first round-trip time. When the first acknowledgment for data arrives at the TCP sender, the sender continues in slow-start, sending two packets for each acknowledgment. The connection using Quick-Start completes in just over half the time required by the non-Quick-Start connection.

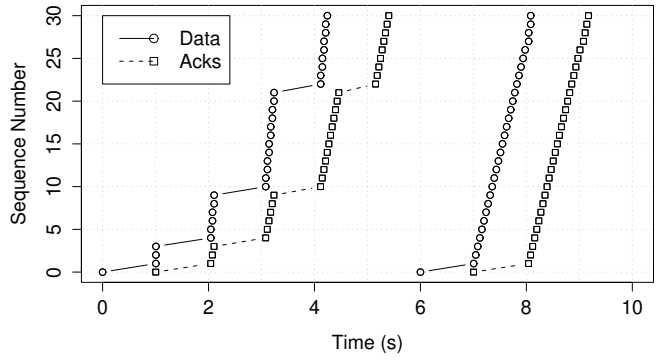


Figure 6.2: Normal TCP Slow-Start (left) vs. Quick-Start (right).

Equation (6.3) gives the number of round-trip times, *NumRtts*, required for transmitting N packets of data in TCP slow-start assuming an ACK for each segment transmitted[5], in addition to the initial SYN exchange, given an initial congestion window of W packets (and, where N and W are both at least 1 segment).

[4] In this scenario the link bandwidth was 384 Kbps and the round-trip delay one second, roughly motivated by a GPRS/EDGE wireless scenario [150].

[5] This assumes that there is no congestion in either direction and the receiver's advertised window does not constrain the congestion window.

$$NumRtts = \left\lceil log_2 \left(\frac{N}{W} + 1 \right) \right\rceil \tag{6.2}$$

From this equation we note the clear attraction to maximizing W as much as is appropriate over a given network path.

Next we use the ns-2 simulator to investigate the ideal impact of Quick-Start. We use a simple scenario with link capacity set at either 384 Kbps or 100 Mbps, various link delays, routers with unlimited buffers, routers willing to allocate 90% of their capacity to Quick-Start requests and TCP making Quick-Start Requests of 20 MB/sec. Figure 6.3 shows the results of the simulations. Although the simulation scenario is not necessarily realistic, it illustrates the potential impact of using Quick-Start. The results confirm the theoretical analysis above, showing that increasing the initial *cwnd* aids performance — especially for medium-sized transfers that are close to the delay-bandwidth product of the network path. In addition, the plots show that Quick-Start is less beneficial for excessively short or long transfers. Short transfers leave little room for improvement since they take little time. The performance of the long transfers in these simulations is dictated by the bottleneck link rate. Therefore, the longer the connection lasts the less impact the startup scheme has on overall performance since the connections perform identically after the startup phase. These results are similar to those presented in a study of an initial implementation of Quick-Start [156].

Figure 6.4 shows a similar graph, but with an analytical estimate of the performance improvement provided by Quick-Start. The number of round-trip times R required to transmit N packets of data is approximated using equation 6.3, where W is the size of the initial congestion window (from either Quick-Start or from the default initial window), and M is the delay-bandwidth product of the path. The number of round-trips R includes one round-trip for the initial TCP SYN/SYN-ACK handshake. For Equation 6.3, we assume that the connection is the only traffic, and that the routers each include a delay-bandwidth product of buffering. As a result, once the congestion window reaches the delay-bandwidth product, the TCP connection continues to keep the pipe full, transferring a delay-bandwidth product of data for each time unit equal to the initial round-trip time.

$$\begin{aligned} M &= bandwidth * RTT/packet\_size \\ R &= log_2 \left(max \left(\tfrac{min(N,M)}{W} + 1, 2 \right) \right) + \left\lceil \tfrac{N}{M} \right\rceil \end{aligned} \tag{6.3}$$

Figure 6.4 assumes a packet size of 1500 bytes, an initial congestion window W of three segments without Quick-Start, and an approved Quick-Start request of 1.3 Gbps, the maximum request size allowed by the specification [54]. Thus, Figure 6.4 illustrates an upper bound on possible improvement with Quick-Start

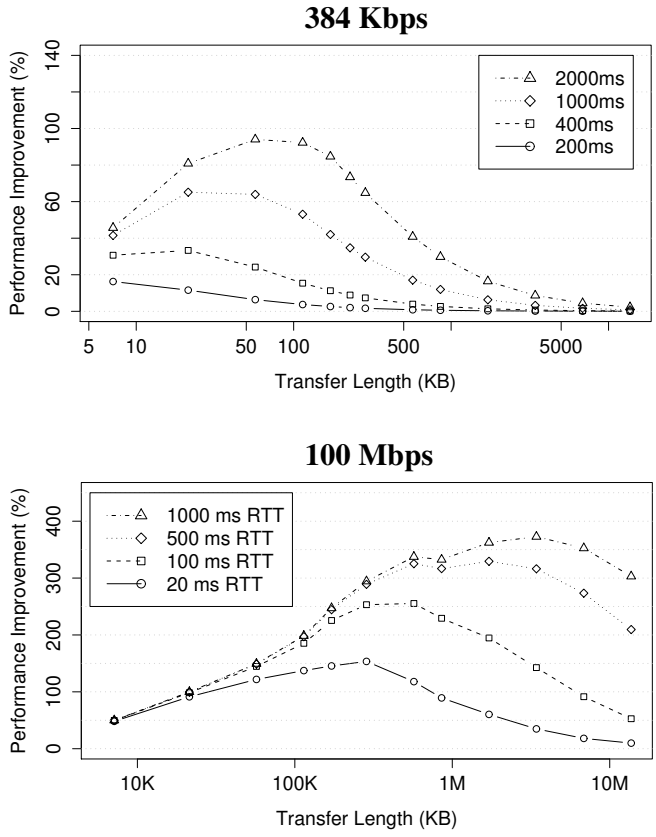


Figure 6.3: Relative improvement with Quick-Start, for a 384 Kbps link and a 100 Mbps link with a range of round-trip times.

– it is not recommended that routers approve Quick-Start requests equal to the entire link bandwidth.

6.5.2 The Size of the Quick-Start Request

We next consider how the sender chooses the Quick-Start request size, and how the size of Quick-Start requests affects the aggregate usefulness of Quick-Start. As discussed in Section 6.2.2, an ideal Quick-Start request would contain the precise sending rate the connection would like to use. However, knowing such a sending rate is non-trivial and depends on a number of factors. A simple Quick-Start implementation for TCP could send a fixed Quick-Start request each time

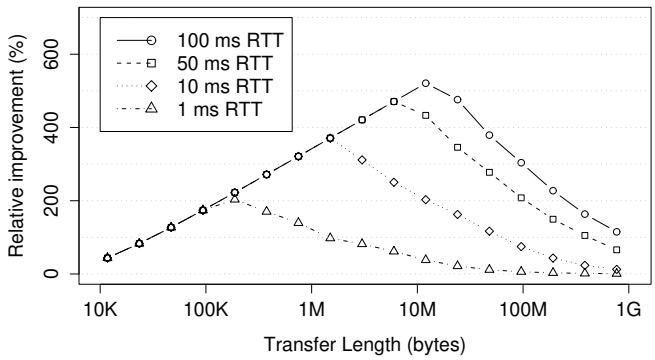


Figure 6.4: Upper bound for relative improvement with Quick-Start, for a single flow over a 10 Gbps link, with a range of round-trip times.

a request is transmitted. This would not be unreasonable for initial Quick-Start requests, since in many cases, the TCP sender has no knowledge about the application or the network path when the TCP SYN segment is sent. For Quick-Start requests sent in the middle of a connection, e.g., after an idle period, the sender may be able to make a more informed Quick-Start Request.

To illustrate the problem with overly large Quick-Start requests we simulate two scenarios involving web traffic that uses one TCP connection for each web object transferred. Figure 6.5 shows the results. Each vertical line on the plots represents a separate TCP connection's length, and each circle indicates the quantity of Quick-Start data transmitted over the given connection. In the first case (top plot), TCP connections use a static Quick-Start request of 2 MB/sec for each connection. In the second scenario (bottom plot) the requests are ideal (even if unrealistic) for the amount of data the given connection will ultimately transmit. In addition, Quick-Start is not used if the connection is able to send all data in 3 segments (per the initial *cwnd* allowed by [9]). This example uses an average web object size of 60 packets.

As shown in the top plot, Quick-Start requests are generally granted for only the first connection in each group. The router is generally unable to approve requests of later connections in each group, because the first connection is granted all of the available Quick-Start bandwidth even though the first connection cannot use such a large allocation. As a result, the extra allocation is "wasted", in that subsequent Quick-Start requests are denied unnecessarily. The bottom plot

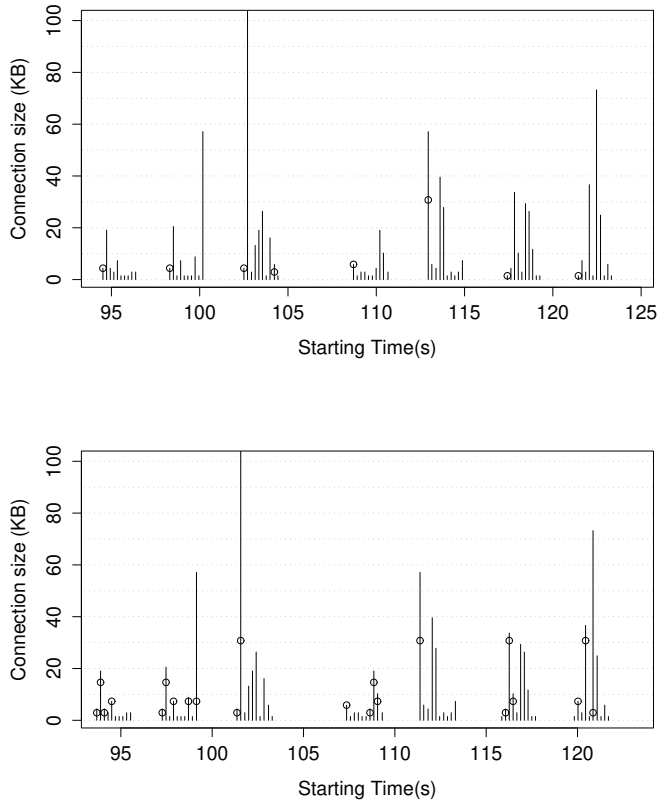


Figure 6.5: TCP connection lengths and starting times. Connections with Quick-Start packets are marked with a circle.

shows that when making ideal Quick-Start requests the Quick-Start requests are approved more often because there are fewer wasted approvals.

While the ideal case above is preferable, TCP connections do not, in general, have enough information to make ideal requests. However, there are several ways systems can cope. First, if an end-host is configured to understand the maximum capacity of its last-mile hop, C bytes/sec, requests could be chosen to be no larger than C. Going even further, a policy decision could be made to disallow any one TCP connection from using more than some fraction of the capacity and that could be used as an upper bound on the Quick-Start request (e.g., on a large web server). In addition, a sender could leverage the size of the local socket send buffer, S bytes, and the receiver's advertised window, W bytes, when

choosing a request size[6]. Given an RTT of R sec[7] TCP can send no faster than $min\ (S, W)\ /\ R$ bytes/sec (assuming W is non-zero and using S if the advertised window is not yet known). Finally, and more speculatively, if an application informed the sender of the size of a particular object (when known), say O bytes, the sender could request precisely the rate required to transmit the object in a single RTT as $(O + (O/MSS) * H)/R$ bytes/sec for a given MSS size and estimated header size of H bytes. While these techniques do not necessarily provide for an ideal Quick-Start request they could well provide a more reasonable request than simple picking a static rate for all cases.

6.5.3 Loss of Quick-Start Packets

We now consider the response of a TCP sender to the loss of a Quick-Start packet, that is, a packet sent in the RTT after a Quick-Start Response triggers an increased sending rate.

Routers should only approve a Quick-Start Request when the output link is significantly underutilized and therefore there should be few congestion losses due to transmitting at the rate determined by Quick-Start. However, it is possible for there to be losses of Quick-Start packets because the allocations are not reservations. If a Quick-Start packet is lost after an approved Quick-Start Request, we call this a *Quick-Start failure*. This situation can arise for a number of reasons, for instance because a burst of traffic arrives at a router immediately after the router approves a Quick-Start Request or because a buggy or broken router simply approves all Quick-Start requests or mis-calculates the rate that should be approved. An explicit congestion notification [136] for a Quick-Start packet is also a Quick-Start failure, and the TCP sender should revert to default TCP congestion control if it gets such a congestion notification.

Generally, after detecting a lost packet from three consecutive duplicate acknowledgements, the TCP sender halves its congestion window and transmission continues using the congestion avoidance algorithm [81, 11], increasing the congestion window by roughly one segment each round-trip time. However, when a Quick-Start failure occurs, the sender cannot make strong assumptions about the current path capacity; in particular, the sender cannot fall back of the fact that a congestion window of half the current size was successfully transmitted in the previous round-trip time, as is the case during slow-start. As a result, halving the congestion window would not necessarily be an appropriate response to

[6]When sending a request in the initial SYN segment of a connection the sender will not know the peer's advertised window.

[7]Or, an approximation if the connection has not yet taken an RTT measurement.

a Quick-Start failure. Instead, as specified in [54], after a Quick-Start failure the TCP sender returns to slow-start, using the default initial window, as it would have done if Quick-Start had not been approved.

Figure 6.6 shows time-sequence plots of several different TCP variants to illustrate TCP's response to a loss of a Quick-Start packet. The top plot in the figure shows a Quick-Start failure followed by fast retransmit and fast recovery (i.e., a simple halving of the congestion window). The second figure shows a Quick-Start failure followed by the proposed response of a slow start from the standard initial congestion window. Finally, the bottom plot shows a connection using standard slow start without Quick-Start. Because after fast recovery the congestion window increases in a linear fashion while Slow-Start increases *cwnd* exponentially, the Slow-Start response may find the appropriate sending rate faster than congestion avoidance, and hence offer better performance (as is illustrated in the figure). In addition, depending on the size of the congestion window used by Quick-Start, a simple halving may not be enough to alleviate congestion within the network and so several multiplicative decreases could be required before TCP finds an appropriate value for *cwnd*. With a Slow-Start response to a Quick-Start failure, the sender loses roughly two round-trip times because of the Quick-Start failure[8], compared to a transfer without Quick-Start (shown in the bottom graph of Figure 6.6). While a Quick-Start failure should be a rare event, Figure 6.6 shows that standard slow start without Quick-Start can be a better choice over a path with a badly behaving or buggy router.

Finally, we note that ECN [136] can be used with Quick-Start. As is always the case with ECN, the sender's congestion control response to an ECN-marked Quick-Start packet is the same as the response to a dropped Quick-Start packet, thus reverting to slow start in the case of Quick-Start packets marked as experiencing congestion.

[8]This assumes SACK-based loss recovery that can detect and repair multiple losses within one RTT [23]. More generally, the connection is lengthened by one Quick-Start RTT and the time required by the loss recovery operation when compared to standard TCP.

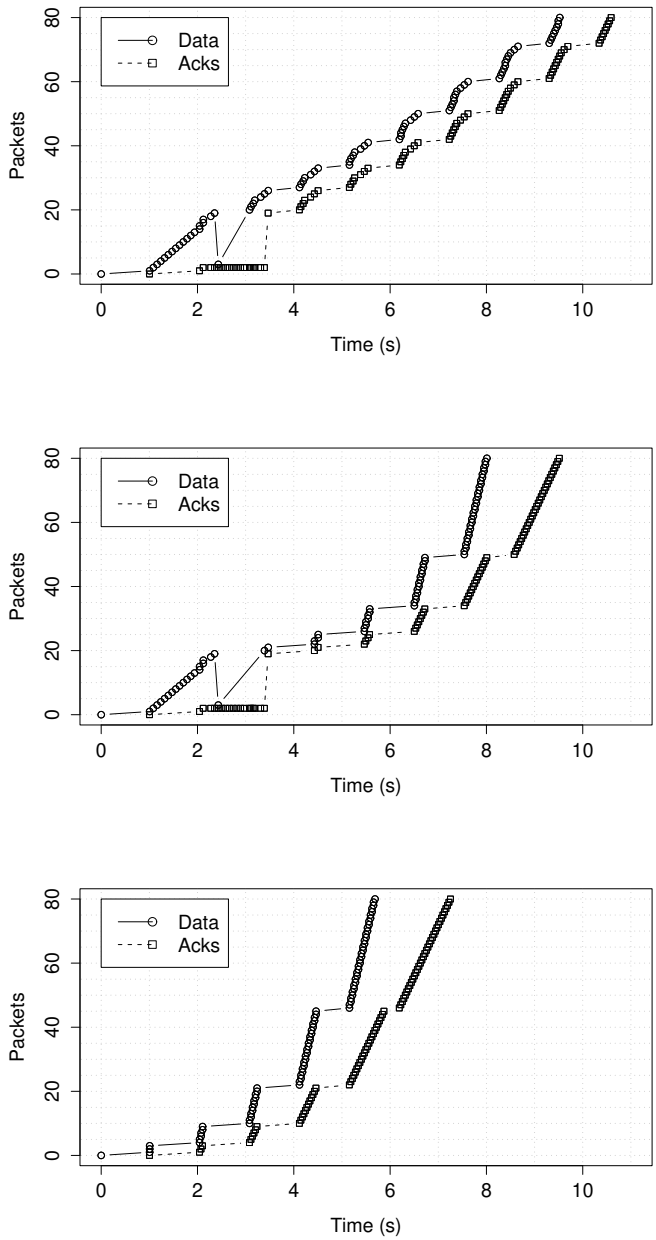


Figure 6.6: The TCP Response to a Quick-Start Failure. Top: Halving the window after a loss. Middle: Slow-Start after a loss. Bottom: Slow-Start without Quick-Start, without losses.

6.5.4 Aggregate Impact of Quick-Start

Because Quick-Start requests are only approved when the output link is significantly underutilized, Quick-Start should have little effect on the long-term aggregate utilization and drop rates on a link. In particular, when link utilization is high, routers should not approve Quick-Start requests; thus, Quick-Start is not a mechanism designed to help a router maintain a high-throughput low-delay state on the output link. In Section 6.6 we study various methods for routers to use to choose whether to approve Quick-Start requests and how much capacity to grant each request. In addition, we illustrate the implications of using Quick-Start when the router is not significantly under-utilized.

For the traffic models used in this chapter, the amount of data requested by a user is independent of whether Quick-Start is used, and independent of the the fate of the Quick-Start requests. While the use of Quick-Start or particular allocations from the routers will have an impact on the time required for particular transfers, the aggregate amount of data requested is not affected. Given this model, although the use of Quick-Start might be of great benefit to the individual user, Quick-Start should have little effect on the long-term aggregate link utilization or packet drop rates.

Figure 6.7 shows the overall utilization and aggregate drop rates with and without Quick-Start, for a simulation scenario with web traffic with an average object size of 400 packets (as described in Section 6.4) on a 10 Mbps shared link as a function of the number of web sessions. As shown in the figure, the utilization and drop rates are largely independent of whether or not Quick-Start is employed. The line labeled "QS Bandwidth" in the top graph of Figure 6.7 shows the relative bandwidth used by Quick-Start packets in the simulations using Quick-Start — indicating that Quick-Start is being put to use at the beginning of transmission. We also conducted simulations with smaller average web object sizes (of 60 packets) and obtained similar results.

Figure 6.8 shows per-connection performance of all traffic involved in a simulation of 3 web servers. Each point on the plot represents the duration of a single connection, with the point type indicating whether Quick-Start is used. The top plot shows the results from a simulation run over 10 Mbps while the bottom plot uses a 100 Mbps bottleneck. For medium to large transfers the plots show Quick-Start improves performance — by a factor of 2–3 in many cases, with larger savings over the higher bandwidth path. These plots show that even though the overall bandwidth usage and drop rates are similar with and without Quick-Start, per-connection performance is increased when using Quick-Start.

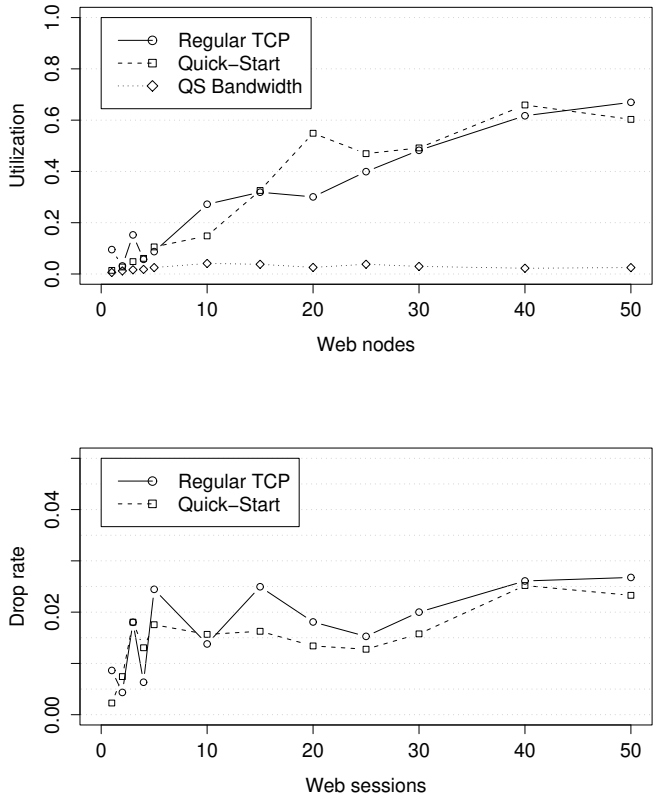


Figure 6.7: Comparison of utilization and drop rates with a 10 Mbps shared link.

6.6 Router Algorithms

This section discusses several possible Quick-Start algorithms for routers to use to choose when to approve Quick-Start requests and how much capacity should be allocated when approving requests. We start with a basic algorithm that requires minimal state, and proceed to an extreme Quick-Start algorithm that keeps per-flow state for approved Quick-Start requests. It is desirable for routers to be able to process Quick-Start requests efficiently. At the same time, the Extreme Quick-Start algorithm explores the ability of the router to selectively approve Quick-Start requests in order to maximize the use of Quick-Start bandwidth by the end-nodes. A final consideration is attackers that wish to leverage Quick-Start in denial-of-service attacks, which we investigate in the next section.

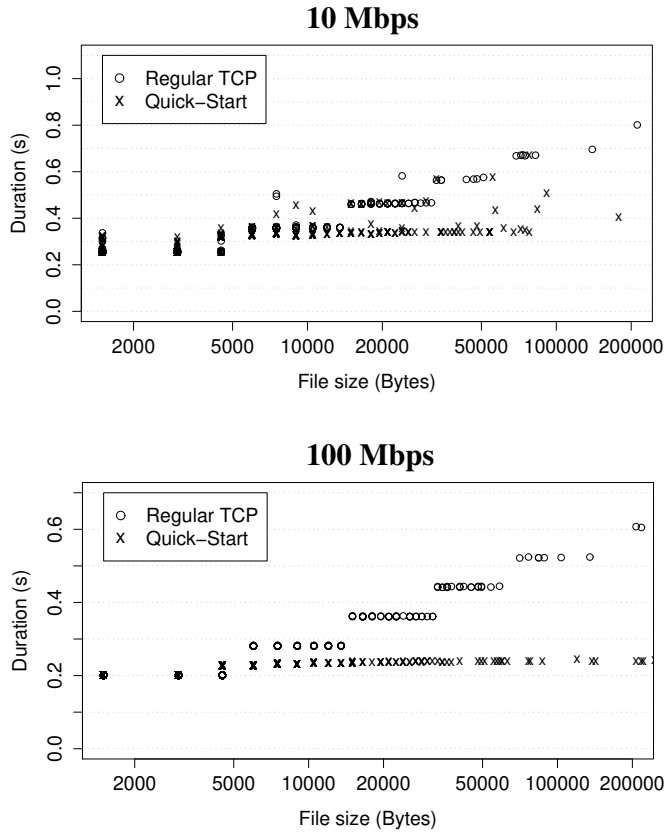


Figure 6.8: Per-connection performance. 10 Mbps and 100 Mbps shared link with 3 web sessions.

6.6.1 Basic router algorithms

Quick-Start requests represent an increased packet processing burden for routers that may also result in an increased end-to-end delay for packets with Quick-Start requests. Therefore, it is important that the algorithm for processing the Quick-Start requests at routers be as efficient as possible, with a small memory footprint.

To know if there is sufficient bandwidth available on the output link to approve a Quick-Start request, the router needs to know the raw bandwidth and have an estimate of the current utilization of the link. The router also has to remember the aggregate bandwidth approved for use by end hosts in the recent past to avoid approving too many requests and over-subscribing the available capacity. In this

section we consider the algorithms used by routers to process Quick-Start requests for point-to-point links; algorithms for multi-access links are left as future work.

The first router design choice concerns the router's method for estimating the recent link utilization. There are a range of measurement and estimation algorithms from which to choose, including alternatives for the length of the measurement period. We discuss two methods for estimating the link utilization, the moving average and measuring the peak utilization. We also note that assessing alternate algorithms is an area for future work.

The **moving average** estimation technique uses a standard exponentially weighted moving average to assess the utilization over the recent past. This scheme was originally used for Quick-Start in [156]. We define $U(t)$ as the utilization at time t, $M(t)$ as the link utilization measurement at time t, δ as the interval between utilization measurements and w as the weight for the moving average. The utilization is defined as:

$$U(t+\delta) \leftarrow w * M(t+\delta) + (1-w) * U(t) \tag{6.4}$$

We note that the weight w should depend on the interval δ, so that the utilization is estimated over the desired interval of time.

The **peak utilization** estimation technique records the link utilization measurements over the most recent N time intervals. Thus, if each time interval is s seconds, then the peak utilization method takes the peak s-second link utilization over the most recent $N * s$ seconds. The peak utilization method reacts quickly to a sudden increase of link utilization, but also remembers a period of high utilization in the recent past. Unless otherwise noted, we use $N = 5$ intervals of 150 msec each.

In addition to the two methods for estimating link utilization, we consider two different algorithms for deciding whether to approve a given Quick-Start request and how much capacity to grant in an approval. Both these algorithms rely on knowing *recent_qs_approvals*, the aggregate bandwidth promised in recently-approved Quick-Start requests — ideally over a time interval at least as long as the typical round-trip times for the traffic on the link. If the time interval for this assessment is too small, then the router forgets recent Quick-Start approvals too quickly, and could approve too many requests, thus over-subscribing the available bandwidth. On the other hand, if the time interval is too large, the router errs on the conservative side and remembers recent Quick-Start approvals for too long. In this case the router counts some of the Quick-Start bandwidth twice, in the remembered request and also in the measured utilization, and as a result may deny subsequent Quick-Start requests unnecessarily. Unless otherwise noted, we use 150 ms as the length of *recent_qs_approvals*.

The **Share** algorithm is introduced in [156] and given in Figure 6.9. The algorithm uses the output link's raw bandwidth and the recent utilization estimate to allocate up to a pre-set fraction ALLOC_RATE of the unused bandwidth for each arriving request. The *rate_request* variable represents the incoming request and *approved* represents the approved rate request that will be forwarded with the packet. The *Share* algorithm does not follow the design criteria we have sketched thus far in this chapter that Quick-Start requests should only be approved when a given link is significantly under-utilized; the *Share* algorithm approves a request for up to a fixed fraction of the available bandwidth, regardless of the levels of utilization. We include an assessment of the *Share* algorithm in this chapter in order to (i) compare the router algorithms we introduce with previous work and (ii) to validate our design criteria that Quick-Start should in fact only be used when all routers along a path are significantly under-utilized.

The **Target** algorithm, given in Figure 6.10, approves Quick-Start requests only when the link utilization, including the potential bandwidth of recently-granted Quick-Start requests, is less than some configured percentage of the link's bandwidth, denoted *qs_thresh*. This gives a router direct control over the notion of "significantly under-utilized". When a Quick-Start request is approved, the approved rate is reduced, if necessary, so that the total projected link utilization does not exceed *qs_thresh*.

Figures 6.11 and 6.12 show simulations with the Share and Target algorithms, respectively. The simulations use a range of values for the ALLOC_RATE parameter in the *Share* algorithm and a range of values for the *qs_thresh* parameter in the *Target* algorithm. Both the Share and the Target algorithms use the peak utilization method for estimating link utilization.

The top graph in Figures 6.11 and 6.12 shows the overall link utilization for each simulation. The middle graph shows the fraction of Quick-Start Requests approved. Finally, the bottom plot shows the fraction of Quick-Start failures. The main difference between the two algorithms is that the Share algorithm approves more Quick-Start requests and experiences a larger number of Quick-Start failures than the *Target* algorithm as the network becomes more congested. We note that the ALLOC_RATE parameter does not control *whether* the Share router approves a Quick-Start request; it only controls the *size* of the approved request. The Share algorithm approves Quick-Start Requests even at high utilization levels. Even though the approved requests are for progressively smaller portions of the bandwidth the rate of failure increases. Finally, we note that the fraction of failure for both algorithms is relatively small. However, given that both algorithms have roughly the same complexity, the *Target* algorithm would be preferred given the results in Figures 6.11 and 6.12.

```
avail_bw = bandwidth * (1 - utilization);
avail_bw = avail_bw - recent_qs_approvals;
approved = avail_bw * ALLOC_RATE;
if (rate_request < approved) {
  approved = rate_request;
}
recent_qs_approvals += approved;
```

Figure 6.9: The Share algorithm for processing Quick-Start requests.

```
util_bw = bandwidth * utilization;
util_bw = util_bw + recent_qs_approvals;
if (util_bw < qs_thresh * bandwidth) {
  // Approve Quick-Start Request
  approved =
      qs_thresh * bandwidth - util_bw;
  if (rate_request < approved) {
    approved = rate_request;
  }
  recent_qs_approvals += approved;
}
```

Figure 6.10: The Target algorithm for processing Quick-Start requests.

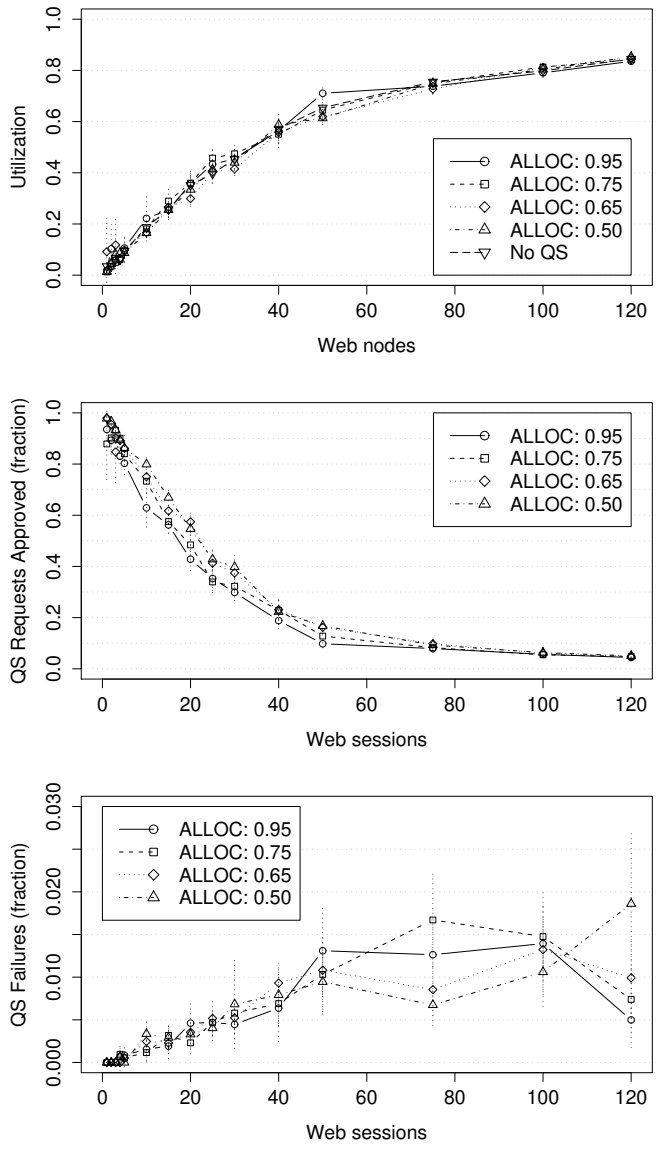


Figure 6.11: Performance of *Share* algorithm.

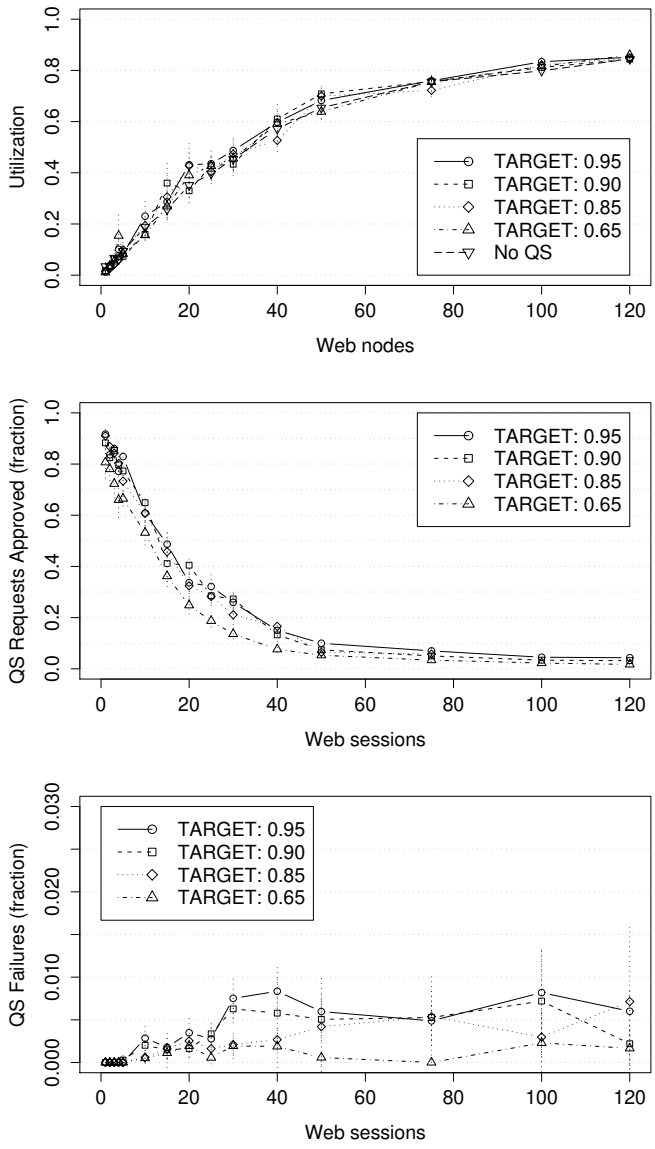


Figure 6.12: Performance of *Target* algorithm.

Figures 6.13 and 6.14 compare the moving average and peak utilization methods for estimating link utilization. The simulations use the *Target* algorithm with a 10 Mbps shared link and a target level of 90 %. The top graphs show the fraction of Quick-Start requests approved, and the bottom graphs show the fraction of approved Quick-Start requests with dropped packets. The moving average simulations were run with a range of values for the weight w, and the peak utilization simulations were run with a range of values for the number of 150-msec intervals over which the peak utilization was chosen. As the figures show, the method for estimating the link utilization does not significantly affect the approval rate of Quick-Start requests, but it does affect the failure rate; simulations using the moving average link utilization have a higher fraction of Quick-Start failures. The legend in each figure shows the overall time interval for the estimation; for the moving average graph, this is estimated as the time needed for $-1/ln(1-w)$ measurements, where a measurement is taken for each departure from the queue [171].

Figure 6.13 shows that the selection of the weight w in the moving average equation does not have a strong effect on the number of Quick-Start failures. The weight controls the time interval over which the link utilization is estimated, but the moving average method still estimates the *average* utilization; it does not take into account the variance of traffic intensity that can be present, particularly on links with low to moderate levels of link utilization. For Quick-Start, where the router does not want to approve Quick-Start requests that could result in even transient congestion, tracking the average link utilization can result in unwanted Quick-Start failures.

For the simulations with the peak utilization method, the Quick-Start failure ratio is generally lower than with the moving average method. When there are more than 50 web servers, using only three recent measurements for peak utilization causes more Quick-Start failures than when a larger number of intervals are used. With twenty intervals there are hardly any Quick-Start failures. However, when ten or more intervals are used, the approval algorithm is also significantly more conservative, with fewer Quick-Start requests being approved.

6.6.2 Extreme Quick-Start in routers

We use the term *Extreme Quick-Start* for a Quick-Start router that maintains per-flow state about Quick-Start requests. With Extreme Quick-Start we can analyze how much Quick-Start performance could be improved if router efficiency was not a limiting factor. For example, an Extreme Quick-Start router could perform the following actions:

- A router could keep track of individual approved Quick-Start requests, and

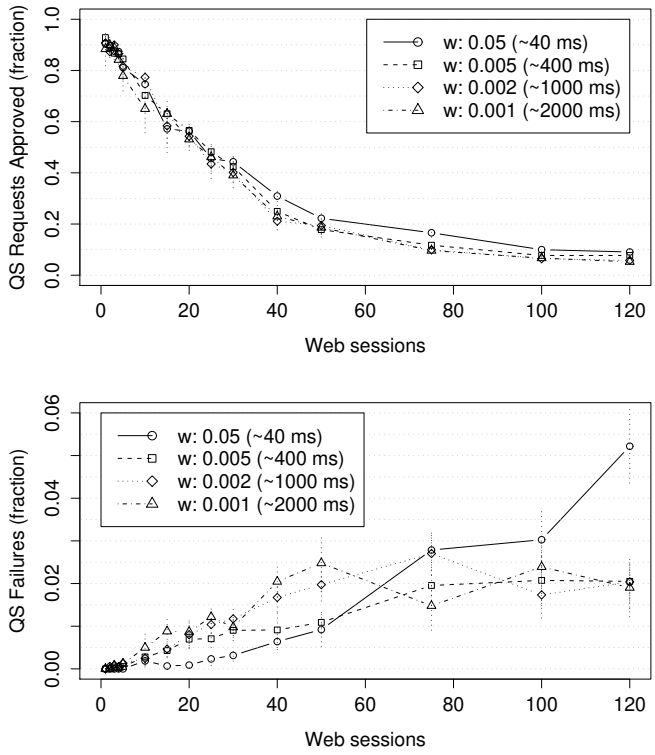


Figure 6.13: Performance of moving average utilization metric.

note when the Quick-Start bandwidth resulting from that request begins to arrive at the router (if in fact it does). This allows the router to more accurately estimate the potential Quick-Start bandwidth from Quick-Start requests that have been approved but not yet used at the end nodes.

- A router could keep track of the fairness of Quick-Start request approvals. If it appears that there are a number of requests that are not approved because earlier requests have allocated all of the available Quick-Start bandwidth, the router could reduce the rate approved for individual requests in order to achieve better fairness between flows.

Our Extreme Quick-Start implementation tracks the Quick-Start Requests made for each traffic flow, and for each arriving packet it calculates how much data for a flow has been transmitted after a Quick-Start Request. For each flow the router

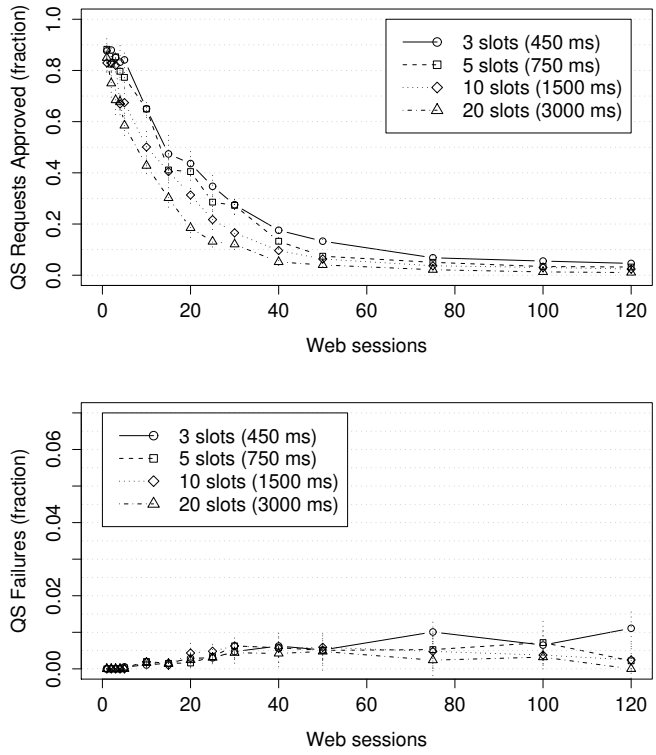


Figure 6.14: Performance of peak utilization metric.

updates a variable `qsr_used` that tracks how much the flow has used from the allocated rate. In order to know how much data is expected to arrive after the incoming rate request the router needs to have an estimate of flow's round-trip time, because the senders are in the Quick-Start state for the duration of one round-trip time during which they are assumed to have used the bandwidth they have requested. For each incoming Quick-Start Request, the router uses the current `qsr_used` values from all open flows and compares them to the currently open Quick-Start requests for each flow to get an exact estimate on how much of the requested bandwidth has already arrived, and is therefore accounted for in the current utilization calculation at the router. This way the router can more accurately estimate the currently available bandwidth instead of using rough aggregate estimate of the outstanding Quick-Start approvals, as done in the basic Quick-Start.

Using the above described information for each flow, our Extreme Quick-Start implementation calculates a simple score for each sender based on the fraction of Quick-Start Requests made by the sender and the actually used bandwidth during the first round-time. This score can be used to identify senders that have a tendency to request more bandwidth that they are going to use, and adjust the approved rate in future requests from those senders accordingly. Section 6.7 describes the use of this algorithm in more detail.

As mentioned above, it is useful for an Extreme Quick-Start router to know the RTTs of flows, in order to set the length of the interval for measuring the arrival rate of packets from a flow after an approved Quick-Start request. There are a number of techniques for routers to estimate flows' RTTs [84]. In the analysis below, we assume that the Extreme Quick-Start router implements a reliable method for evaluating RTTs.

Figure 6.15 compares the basic Quick-Start algorithm and the Extreme Quick-Start algorithm for scenarios with a small range of RTTs. When the RTTs are known (or easily guessed) by the router, and the router can accurately set the length of *recent_qs_approvals* state to roughly match the round-trip time. In these simulations, the basic Quick-Start variant uses the *Target* algorithm with the peak utilization method. The Extreme Quick-Start variant uses a router that keeps track of approved Quick-Start requests separately for each flow, updating its state during the transmission of the Quick-Start window as the packets arrive, and achieving a more accurate estimate of the overall amount of Quick-Start traffic that is still expected to arrive. Figure 6.15 shows a scenario with a range of round-trip times from 80 to 120 msec, and with the length of *recent_qs_approvals* set to 100 msec for basic Quick-Start. From the top plot we see that the utilization is nearly the same regardless of whether basic Quick-Start or Extreme Quick-Start is employed. However, the bottom figure shows that the fraction of bytes transmitted using Quick-Start is greater when Extreme Quick-Start is used by the router to track each allocation in detail. This illustrates Extreme Quick-Start's power in terms of more closely tracking resources such that more requests are approved than when using basic Quick-Start. This kind of scenario is certainly not typical, but there could be some initial Quick-Start deployment scenarios, such as in limited intranets, where there is a limited range of RTTs, and also where the traffic and network characteristics could be accurately estimated.

As a point of contrast we changed the length of *recent_qs_approvals* to 1.5 seconds to investigate Extreme Quick-Start in the context of a basic Quick-Start router that does not have a "typical" RTT and therefore chooses a conservative setting (i.e., this setting results in few Quick-Start failures, but also fewer Quick-Start request approvals). Figure 6.16 shows Quick-Start traffic as a fraction of the

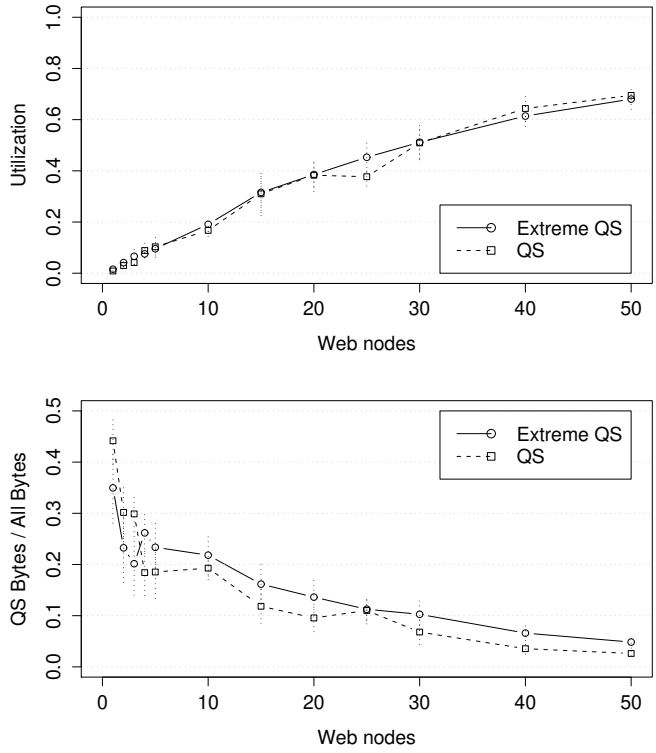


Figure 6.15: Extreme Quick-Start and Basic Quick-Start with highly tuned parameters.

total amount of data transmitted. In this simulation we also found the utilization of basic Quick-Start and Extreme Quick-Start to be nearly identical. The plot shows that the fraction of bytes sent during the Quick-Start phase of the connections is greater when using Extreme Quick-Start. The reason for this is that the Extreme Quick-Start router is able to keep track of the unused allocation separately for each flow as the packets arrive. Therefore, less wasted capacity is allocated by Quick-Start which allows more connections to be approved to use Quick-Start. The difference between basic Quick-Start and Extreme Quick-Start in this figure is larger than the difference shown in Figure 6.15 due to the more conservative setting for the length of *recent_qs_approvals*.

While the basic Quick-Start is relatively light-weight algorithm at routers with only a few bytes of additional aggregate state to maintain, Extreme Quick-Start

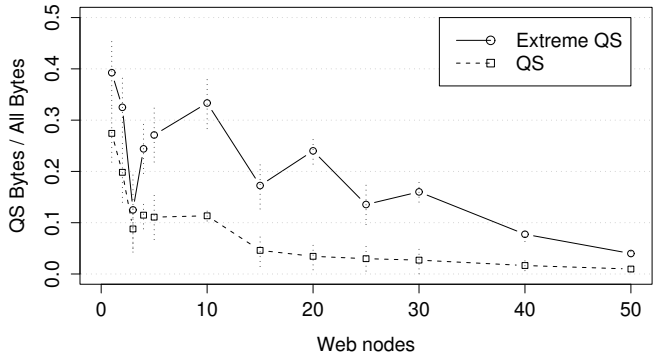


Figure 6.16: Basic Quick-Start and Extreme Quick-Start with conservative parameters.

needs to maintain some state separately for each flow passing the router. This likely makes Extreme Quick-Start an infeasible algorithm to be implemented at all routers, although we have not analyzed the actual processing requirements of Extreme Quick-Start in detail. However, it is possible for different routers to use different algorithms to evaluate the Quick-Start Requests. Therefore it would be possible to implement Extreme Quick-Start at selected points in the network, for example close to the wireless link with possibly less traffic load, to gain some of its advantages while using the basic Quick-Start elsewhere in the network.

6.7 Attacks on Quick-Start

Quick-Start is vulnerable to denial-of-service attacks along two vectors: (i) increasing the routers processing and state load and (ii) causing temporary false allocations of Quick-Start capacity that will never be used but may prevent legitimate flows from having their Quick-Start requests approved. Since Quick-Start requests represent a processing burden on the routers involved, a storm of requests may cause a router's load to increase to the point of impacting legitimate traffic. Given the processing burden imposed by Quick-Start this could well be worse than a simple packet-flooding attack. A simple limit on the rate Quick-Start requests could be considered (with a policy of ignoring requests sent in excess of this rate) to mitigate the attack on the router itself. In the case of Extreme Quick-Start another problematic aspect of a storm of packets is the memory requirement

to track false "connections".

The second type of attack is more difficult to defend against. In this attack arbitrarily large Quick-Start requests are sent by the attacker through the network without any further data transmission. With a relatively low-rate stream of packets, this can cause a router to allocate capacity to the attacker's connections and thus temporarily reduce the amount of capacity that can be allocated to legitimate Quick-Start users. Note that the attack does not actually consume the requested bandwidth and therefore the performance of connections competing with attacks is no worse than connections that simply do not make use of Quick-Start. These attacks are particularly difficult to defend against for two reasons. First, the attack packets do not have to belong to an existing connection to do damage. And, second, since the attack just involves a Quick-Start request traversing the network path in one direction only to trigger bogus allocations, a response is not required. Therefore, spoofed source addresses are a possible aggravating factor for both hiding the location the attack is originating from and causing a simple blacklisting defense to fail.

An additional problematic aspect of Quick-Start is that legitimate requests could well cause the same impact as attack packets. Consider a Quick-Start request that is approved by the first router for some given rate, R, which the router then marks as "allocated" for some period of time. Now assume the same request hits a downstream router that either does not understand Quick-Start requests, reduces the rate to less than R or decides it cannot approve any Quick-Start request. In this case, the first router has allocated some amount of capacity that will not be used because of the conditions elsewhere in the network. From the perspective of the first router this is similar to the attack described above. In other words, capacity allocated for Quick-Start goes unused and therefore reduces the router's ability to approve further Quick-Start requests[9].

Since Quick-Start is a loosely-connected distributed approach, routers have few options to deal with allocations that are never used (or, not fully used). One approach is to use the notions of Extreme Quick-Start to track a host's use of Quick-Start and to disallow Quick-Start for hosts that have previously used less than their previous allocations. This approach is barely useful if an attacker can spoof source addresses because each attack packet could simply use a random source address. Further, it opens the door for another attack type — namely, that an attacker can prevent a particular host from ever using Quick-Start by making a

[9] At first glance, allowing the router to watch the Quick-Start responses offers more information. However, due to asymmetric routing we cannot assume that a router will see the Quick-Start responses. In addition, an arbitrary router has no idea how to tell if the $TTLDiff'$ in the response is valid and therefore whether the sender will ultimately make use of the response.

bogus request on the victim's behalf, thereby getting the victim blacklisted. In addition, using a blacklist approach seems heavy-handed in the context of legitimate traffic that does not fully use their Quick-Start allocation (as sketched above).

Another approach is for Extreme Quick-Start routers to track the fraction of Quick-Start allocations hosts use and then make this a factor in the approval of subsequent requests. For instance, if some host requests a rate of X bytes/sec but uses only $X/2$ bytes/sec because of a downstream limitation, a router may decide to halve future rate requests from that host. An Extreme Quick-Start router has the required information to identify hosts that frequently make Quick-Start requests for more bandwidth than is actually consumed. Therefore, the Extreme Quick-Start router can reduce subsequent rate requests approved for these hosts.

Furthermore, an extension to the Quick-Start protocol itself has been proposed to mitigate the effect of false Quick-Start Requests by adding a third pass to the protocol to follow the Quick-Start response [54]. In this approach, after getting the Quick-Start response, the TCP sender sends a third message to report the approved rate to all routers along the path.

We implemented the following algorithm in the Extreme Quick-Start router. The router stores both the Quick-Start allocation, $A(F)$, and the amount of bandwidth used, $B(F)$, during the Quick-Start phase for each flow, F. After the monitoring period has elapsed, the router calculates the fraction of the allocation actually consumed as $C = B(F)/A(F)$, limiting the maximum C to 1. The router maintains a score $S(H)$ for each sending host H as follows:

$$S(H) \leftarrow w * max(C, S(H)) + (1 - w) * min(C, S(H)) \tag{6.5}$$

In our simulations we set the gain w to 0.2 and used a measurement interval of 1.5 seconds. Instead of a pure moving average, we selected a function that reacts quickly to hosts that often make larger requests than they end up using. When a new request arrives, the router decreases the incoming rate request by the factor $S(H)$ for the given host H.

Figure 6.17 compares the performance of basic Quick-Start and the variant of Extreme Quick-Start sketched above. The web servers make static Quick-Start requests of 2 Mbps for all TCP connections, regardless of the object size. As the figure shows, when adjusting the allocation approved based on previous usage, Extreme Quick Start is able to allow a greater fraction of traffic to utilize Quick-Start compared to the case when the router does not track allocation usage.

Tracking per-host and per-connection state to mitigate this problem may be a high barrier. However, we note that (i) developing schemes based on aggregate traffic that do not require fine-grained tracking may be possible and (ii) even if fine-grained tracking is required a router that is able to approve Quick-Start should

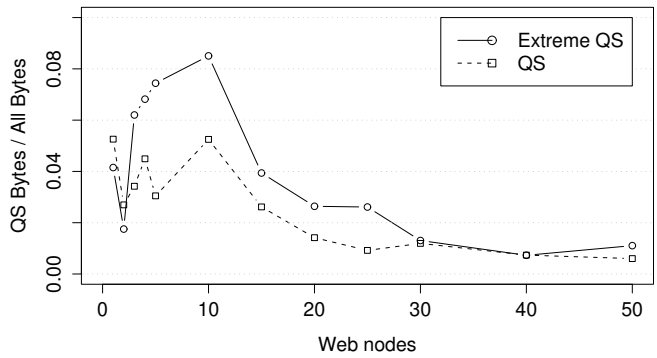


Figure 6.17: Impact of large Quick-Start requests for all TCP connections when accounting for abuse.

be under-utilized and therefore may have some cycles to spare (and could simply turn off *all* Quick-Start activity when busy). We defer an in-depth study of such schemes to future work.

6.8 Summary and Open Issues

In this chapter we have described the Quick-Start protocol and discussed some of the design alternatives in the protocol. We also presented some alternatives for algorithms to be implemented at end hosts and routers for protocol processing, and evaluated the performance and relative difference of the Quick-Start algorithms. We have discussed the potential costs and benefits of Quick-Start on performance in an uncongested environment, the appropriate response to the loss or ECN-marking of a Quick-Start packet, and the range of algorithms for routers for processing Quick-Start requests. However, there are many issues we could not thoroughly study in this work, and we list some of the more significant below as pointers for future research topics.

- How effective would Quick-Start be in practice in realistic scenarios of five or ten years from now? Would Quick-Start be of great benefit to users who could send an entire large transfer in a single round-trip time over an under-utilized path? Or would most of the potential Quick-Start bandwidth

be "wasted" by legitimate requests denied by downstream routers, by requests from aggressive senders sending a request each round-trip time, and by malicious requests whose sole purpose is to deny Quick-Start bandwidth for other users? Recently the Quick-Start algorithm was enhanced with a Quick-Start Rate Report, that aims to improve the effectiveness of Quick-Start by adding a third phase to the protocol to inform the routers what was the actual rate that was approved [54]. However, we have not studied the possibilities or requirements the Rate Report could introduce to Quick-Start processing at routers, for example with the Extreme Quick-Start algorithm.

- Would routers have sufficient incentives to implement Quick-Start, considering the potential benefits, but also the additional processing costs and possible security concerns Quick-Start may introduce? Our current belief is that Quick-Start could be first deployed in networks where the routers and the end-hosts have clear mutual interest in speeding up connection startup. The initial customer demand for the router and end host vendors could become, for example, from a wireless operator that could deploy Quick-Start in its own network to enhance the download times of the content and services provided by the operator itself. Similarly, an organization could take Quick-Start into use in its intranet to be able to use the local content and services efficiently. A possible further research item would be to investigate what are the typical capacities and utilizations in different parts of the network: would a wireless UMTS operator be able to use Quick-Start, or even Extreme Quick-Start with the current equipment at its access routers, or would Quick-Start require additional hardware capacity? Would the situation be more difficult in the UMTS core network? Would it be feasible, within some years of time, to deploy Quick-Start in the backbone network with highly optimized router implementations?

- What would be the minimal sufficient implementation at the routers and would there be sufficient benefit in deploying more complex algorithms in routers?

- Would it be possible to implement faster congestion control startup with a smaller amount of information in packets, such as proposed in the AntiECN or VCP mechanisms? There are 16 bits in the Quick-Start option that comes as an overhead from the generic option processing, and 16 bits of data specific to Quick-Start. The 8-bit TTL field is needed for checking that all network hops have processed the Quick-Start request and at the same time, used as one kind of nonce to give some protection against a misbehaving receiver that tries to forge the Quick-Start responses. Considering the large

space of different kinds of links that may reside on the same connection path, it is also useful to have more than one bit to indicate the approved sending rate. We believe that the current option format is close to the minimum possible to make Quick-Start useful.

- Would we still need an explicit congestion control protocol, such as XCP, if Quick-Start gets adopted by a significant portion of network hosts? Quick-Start has most use for medium-sized connections, and for connections with long lifetime an advanced mechanism would provide finer control than Quick-Start does. However, Quick-Start could be invoked in the middle of a connection, and in combination of explicit congestion notification the transport protocol could be able to speed up and slow down the transmission rate dynamically without having to suffer from packet losses. If the path characteristics changed frequently during a transport connection, we expect a fine-grained explicit congestion control method to adapt to the present conditions more efficiently than the standard TCP congestion control enhanced with Quick-Start and ECN.

- How severe are the additional security issues due to Quick-Start? What are the policing mechanisms that could be deployed in end-nodes and in routers to address these security issues?

- A router should not approve Quick-Start requests if it cannot reliably determine the link utilization all the way to the next hop. What would this mean, in practice, when there is an Ethernet switch, an ATM cloud, or some other non-IP queue between the router and the next-hop IP router? A related problem arises with the IP tunnels such as those used in Virtual Private Network (VPN) solutions. Currently it is unknown if we can ensure proper treatment of Quick-Start in such cases: Quick-Start Request should be processed in every router, but an IP tunnel hides the Quick-Start Request inside an outer IP header, possible encrypting the inner packet header, potentially causing misbehavior of Quick-Start. The tunnel ingress and egress nodes should be enhanced to process Quick-Start Requests appropriately, but that would be a further deployment challenge.

While there are a number of open issues and challenges in the practical deployment of Quick-Start, we believe that the analysis in this chapter helps in further evaluation of the possible usefulness of Quick-Start. One environment where Quick-Start can be expected to be useful, are high-speed wireless links: in many cases the wireless channel is shared by a small number of users, and is therefore underutilized for most of the time. On these links it takes relatively long time

for TCP's congestion window to reach a size that allows effective channel utilization. Therefore Quick-Start could significantly improve the communication performance in such setup.

It is probably unrealistic to expect that Quick-Start would be deployed in the world-wide Internet, but it seems possible to see Quick-Start deployments in limited and better controlled environments, as discussed above. Deploying protocol changes or enhancements in the Internet is a difficult topic, because looking at the past experience, a reasonable belief is that the protocols run in the Internet core routers are not going to be changed more than once in a generation, if even then. It remains for future research to show whether Quick-Start is the right and sufficient change to make in that case, or whether it would be better to use the rare opportunity of change somehow differently, maybe by trying to roll in more fundamental changes, such as XCP, to the congestion control framework.

CHAPTER 7

Using Quick-Start to Improve TCP Performance with Vertical Hand-offs

Public mobile network access is gaining increased diversity in terms of the types of access technologies and scattered deployments within one access technology. This access network diversity combined with an increasing number of multi-radio mobile nodes (MNs) that are equipped with multiple interfaces representing both short range radio (such as Wireless LAN - WLAN) and Wireless WAN (WWAN) access technologies creates an environment, where mobility between access technologies becomes justifiable. Mobility between access networks may involve both horizontal and vertical hand-offs, that is, hand-offs within the same access technology and between different access technologies, respectively.

In this chapter we investigate the use of the Quick-Start algorithm in wireless network environments. Although initially Quick-Start was proposed to start up the TCP connections rapidly, we apply it to quickly probe the capacity of the new network path after a wireless vertical hand-off. We employ an explicit notification that is delivered to TCP to inform it about the hand-off event and to trigger the Quick-Start. Quick-Start is expected to be useful in this case, because the delay and bandwidth characteristics of the different wireless link technologies are often substantially different, and the traditional TCP is known to converge slowly to the new network conditions after a vertical hand-off [48].

Quick-Start allows the TCP sender to find an appropriate congestion window size quickly without having to rely fully on the regular TCP congestion control algorithms that react slowly in high-latency environments. We also propose an enhancement to the Quick-Start algorithm that sets the TCP's slow-start threshold (*ssthresh*) in addition to the congestion window. We study by simulations how TCP performance is affected on vertical hand-offs between WLAN and EGPRS

and how TCP performance can be improved with Quick-Start. As an alternative to Quick-Start, we apply TCP slow-start on the explicit trigger.

The rest of this chapter is structured as follows. Section 7.1 gives some background about vertical hand-offs and past related work. Section 7.2 shortly discusses the IP mobility mechanisms. Section 7.3 describes some of the design issues with explicit notification mechanisms used to help TCP congestion control, and Section 7.4 presents results of our simulation studies conducted with Quick-Start. Finally, Section 7.5 wraps up our investigation.

7.1 TCP Performance on Mobile Hand-offs

Depending on the network environment, a mobile node may be reachable through multiple network interfaces simultaneously or through a single interface at a time, changing the active interface every once in a while. It is also common that one of the network interfaces maintains stable connectivity to the same point of attachment in the Internet (for example, WWAN systems like GPRS/EDGE [29, 150]), while the other network interfaces may change their point of attachment (and the IP address) quite frequently (for example, short range systems like WLAN).

Generally there are two types of hand-off: *make-before-break*, where new IP connectivity is established before the old one is broken, allowing simultaneous communication over the old and new link during the hand-off, and *break-before-make*, where the IP connectivity over the old link is lost before the new one becomes operable, often resulting in packet losses due to the period of disconnection. In a multi-access environment the make-before-break approach is an inherent choice, provided that the applied mobility solution supports using multiple link interfaces simultaneously and the link-level connectivity can be maintained during the hand-off.

Different access networks often represent disparity in link characteristics. For example, link bandwidth, latency, bit-error rate and the degree of bandwidth asymmetry may differ considerably. Therefore, sudden changes in the access link characteristics due to vertical or even horizontal hand-offs may interfere with the transport layer protocols and with the applications that base their protocol behavior on the measured end-to-end path conditions.

Because the TCP congestion window starts from a small initial size and the detection of the correct network capacity is based on packet loss events, adjusting the congestion window is sometimes slow and inefficient in the lack of explicit congestion signals. The ineffectiveness of congestion control is a particular problem in high-latency environments with relatively slow links, such as GPRS/EDGE

(EGPRS). Furthermore, when an MN moves during a TCP connection and executes a vertical hand-off, a typical TCP implementation is unaware of the hand-off event and the potential change in the end-to-end path properties. This causes further challenges to TCP that converges relatively slowly, sometimes after several packet losses, to the correct network capacity.

While there are a number of papers that discuss the interaction of TCP and hand-offs in general (for example [13, 16, 32, 33]), there are less papers that specifically focus on TCP and vertical hand-offs between different access technologies. A thorough discussion on the effect of hand-off on TCP performance is given in [68]. This paper mainly discusses the problems with packet reordering due to the decrease in the propagation delay and congestion-related packet losses due to the decrease in the bandwidth-delay product (BDP). It proposes two schemes, namely, *congestion window reduction* and *nodupack* schemes to improve TCP performance during make-before-break hand-offs. When the hand-off occurs from a high BDP to low BDP network, the remote TCP sender gets an explicit congestion window reduction trigger from the MN and reduces the congestion window. The *nodupack* scheme limits the transmission of duplicate ACKs during the hand-off to avoid unnecessary retransmissions caused by packet reordering. The hand-off is detected either by the TCP receiver noticing that packets arrive through a new network interface or by an explicit trigger from the MN, but the paper does not mention how the trigger is communicated to the remote end.

A comparative study on the effect of vertical hand-off on transport protocols such as TCP and TCP-friendly rate control (TFRC) is presented in [63]. This paper proposes over-buffering to reduce the problem due to the change in BDP to enable smooth changeover between links with different BDP. A drawback of this scheme is that it is difficult to know in advance how much over-buffering is needed.

Huang and Cai [74] propose three schemes to mitigate the effect of increased RTT in make-before-break hand-offs from a fast link to a slow link. These schemes are aimed to soften the dramatic increase in RTT between the old link and new link. First, the fast response scheme requires sending the ACKs over the old link for a short period after the hand-off. In the second scheme, called slow response, a few ACKs are sent over the new link just prior to the hand-off. These two schemes may have practical problems if the old link is not available after the hand-off or if the new link cannot be used prior to the hand-off. The third scheme, called ACK delaying, softens the RTT change by delaying the few first acknowledgments over the new fast link.

The Lightweight Mobility Detection and Response algorithm [158] has been

proposed for making TCP aware of the path change during a vertical hand-off. It is assumed that the MN notices the subnet changes and relays this information to the TCP sender through a TCP option. It recommends that after the hand-off the TCP connection should be treated as a new connection and the TCP sender should reset the congestion control state and the RTO timer and set *ssthresh* to a large value.

7.2 IP Mobility

IP Mobility support is becoming an integral part of the wireless IP data communication [1]. In a multi-access networking environment the MN often needs to reconfigure its IP addresses after changing the point of attachment to the network, or when performing an IP-level hand-off between IP subnetworks. There are several IP Mobility solutions and protocols. Some of them address the IP Mobility problem at the network layer (e.g., Mobile IPv4 [129], Mobile IPv6 [85]), some at the transport layer (e.g., SCTP with dynamic address reconfiguration [154]), and some solutions virtualize remote networks or separate the location and identity transparently from the rest of the system (e.g., MOBIKE [47] and Host Identity Protocol [134]).

Among the IP Mobility solutions and protocols listed earlier there are different approaches to handle mobility. It is possible (i) to have a topologically stable anchor node, like a Mobile IP Home Agent, that the MN registers to. The anchor node represents the mobile terminal while the terminal is outside its home network. Packets are tunneled between the MN and the anchor node when needed. This kind of solution requires deployment of anchor nodes and generally causes inefficient routing of packets. The communicating protocols can (ii) handle mobility directly between the communicating end nodes, and whenever the other end moves the required IP-level information is signaled with the peer. DCCP with mobility extension is an example of such a solution at transport protocol level [98]. The positive sides are the lack of mandatory infrastructure and no need for tunneling, but on the other hand, the lack of a stable anchor node or a rendezvous point complicates locating the moving MN. Finally, (iii) localized mobility management (LMM) [91] handles mobility locally and as much as possible on the access network side without MN's active participation, using tunnels between the access network routers and the anchor nodes. This approach is appealing because it allows terminal mobility also for IP-Mobility-unaware terminals. However, the downside is the new required support and intelligence on the access network.

One of the problems with most existing IP Mobility protocols is that they mainly concentrate on fixing the IP routing and reachability. There are protocols

for reducing the number of packet losses during a hand-off, such as Mobile IP utilizing simultaneous bindings and bi-casting feature, and protocols for enabling low latency hand-offs [103]. However, these solutions still neglect the transport and application layer needs during hand-offs.

7.3 Applying Quick-Start for Wireless Links

When a vertical hand-off occurs, the path characteristics, such as bandwidth and propagation delay, may change dramatically. This causes problems to TCP that typically needs several round-trip times to adapt its transmission rate if the change is significant enough. Also in the connection startup it takes several round-trip times from a TCP's slow-start algorithm to reach an appropriate transmission rate on a high-latency wireless link. The TCP adaptation speed can be improved by an explicit indication from the network that the path characteristics have changed, to indicate that the earlier congestion control state may have become invalid.

The explicit notification mechanisms can be categorized into in-band and out-of-band signaling. Out-of-band signaling could be carried, for example, in ICMP or RSVP packets, whereas in-band notifications are piggy-packed, for example, as IP options, along with the data traffic. We consider in-band mechanisms to have better characteristics for our needs.

Out-of-band mechanisms would have various kinds of difficulties in bearing the explicit information in a hop-by-hop manner. To mention a few, out-of-band signaling would contribute to the overhead in the network, especially since the packets would also need to carry parts of the transport header to make it possible for the end-hosts to identify the correct transport protocol session. In addition, ICMP or RSVP packets might be blocked by some middle-boxes in the network and they would be hidden by IP tunnels, especially with IPsec. While the latter can also be a problem with IP options, we believe that implementing proper ways of handling the notifications in these cases would be somewhat easier with in-band signaling. The in-band versus out-of-band issues are discussed more thoroughly for example in [54].

We assume that the TCP sender gets information about the hand-off event by some way. A mobile host usually is aware of the mobility through its own mobility mechanisms. Instead of hiding the mobility information from the upper layers, the mobile host would need to support internal APIs to allow the mobility management protocol notify the TCP implementation about the mobility events. Often the majority of data is sent by a fixed server in the network. In this case the mobility event would need to be signaled across the network. There have been

some proposals how this could be done [48, 148, 105]. One possibility is to deliver information of changed last-hop link characteristics in-band as part of the normal IP-mobility-related signaling. Other option would be to use a TCP option to indicate that one end of a connection has moved.

After TCP has received a mobility indication, another form of explicit communication is needed to resolve the new path characteristics faster than TCP normally would do. Some of the earlier research based on the older wireless technologies has assumed that the wireless link is the bottleneck on the communication path (for example [4]). This assumption does not necessarily hold today, as the wireless networking technologies have become significantly faster. Because TCP needs to conform to the congestion control principles, and it must not endanger causing severe congestion on the communication path, the information about the last-hop wireless link is not always enough, but the state of the whole path needs to be known in order to determine the appropriate sending rate.

We described the Quick-Start protocol in Chapter 6 and evaluated a number of algorithms that could be applied with Quick-Start at routers and the TCP sender. We now turn to evaluate Quick-Start in wireless networks where hand-offs can cause sudden path changes in the middle of connection. Quick-Start can also be applied in the middle of a connection upon some special events, and we are investigating one such event, namely vertical hand-off between two paths with radically different properties. Quick-Start can potentially improve the communication performance in these environments that are known to be challenging for TCP, as described above.

Because it has been observed that slow-start overshoot is a serious problem with high-latency links [41], we propose an enhancement to Quick-Start that sets the *ssthresh* in addition to the congestion window. We apply a simple logic where the ssthresh is set based on the Quick-Start Response using the same equation as for setting the congestion window, so that after an approved Quick-Start request the congestion window and ssthresh are equally sized. While this is a simple approach, we believe it is an appropriate heuristic over wireless links that do not typically have large amounts of background load. Therefore limiting the TCP congestion window's growth rate based on approved Quick-Start request is expected to prevent congestion losses without significantly limiting the performance. However, if Quick-Start Request temporarily returns a lower rate than what the full link capacity is, it is possible that our approach leads to suboptimal use of the wireless link capacity. Therefore, in future we intend to study more advanced mechanisms for setting the slow-start threshold, for example by applying the *Limited Slow-Start* [53].

The main challenge with Quick-Start is that all routers on the whole network

path need to support it. As discussed above, in order to be able quickly increase the transmission rate, this requirement follows from the principle that congestion can happen on any router or any link from the connection path. However, it would be safe to limit the slow-start threshold without knowing the capacity of the whole connection path. For example, there has been some past work proposing to avoid slow-start overshoot by limiting the TCP's advertised window at the wireless receiver [141]. We discuss more about the general applicability and possible incentives to deploy Quick-Start mobile networks in Section 7.5.

7.4 Simulation Results

This section shows simulation results acquired with ns-2 network simulator. We first discuss the connection startup performance on wireless links, and then move on to investigating different types of vertical hand-offs.

7.4.1 Simulation Arrangements

We are assuming a network topology where the MN is capable of using both Wireless LAN and EGPRS wireless access technologies. The WLAN and EGPRS links both have dedicated base stations that are connected to a common wireless access router with a 100 Mbps link. The router has a 100 Mbps connection to a server in the fixed network. The one-way propagation delay over each link in the fixed network is 2 ms. The WLAN link has a bandwidth of 5 Mbps with one-way propagation delay of 10 ms. The IP packet send queue at the WLAN link has room for 30 packets. The EGPRS link is capable of transmitting 200 Kbps with 300 ms propagation delay. The EGPRS packet queue is capable of holding 32 packets. We believe these parameters approximate fairly well the actual characteristics of EGPRS and WLAN link technologies in a detail that is sufficient for the analysis in this chapter.

Although our simulation model can be considered to be a simplification of a corresponding real-world setup, we think it has the relevant components for evaluating the effect of the vertical hand-offs between two access technologies on TCP performance. We believe that additional complexity on the network side does not have significant effect on the simulation results that are dominated by the wireless link characteristics.

We analyze the behavior of a single TCP connection over the wireless link. Although this might seem a simple setup, it is rather common that the mobile terminals with a limited processing capacity and user interface have only one or few applications and TCP connections active at a time. In addition, the case with

a single TCP connection is most interesting for Quick-Start, as it is intended for under-utilized network paths. With several parallel TCP connections the utilization of the wireless link would often be too high for the Quick-Start requests being approved at the wireless access router, and therefore Quick-Start is not expected to be as useful in such scenarios with wireless links. However, as mobile devices become more efficient and richer in features, the expected number of active TCP connections in a mobile host is expected to increase. Therefore an important topic of future research is to investigate the TCP behavior with Quick-Start and several simultaneous flows on the wireless network. An interesting special use case discussed recently is to use the mobile device as a wireless router. In this case not only the degree of multiplexing on the wireless link is higher, but the mobility of the wireless router is hidden from the hosts behind the mobile router. With mobile router some different form of explicit signaling would be needed from the router to inform the TCP end hosts about mobility. However, we will defer this discussion to future work.

SACK TCP is used in the simulations. When Quick-Start is not active, the TCP initial window selection follows RFC 3390, using an initial window of three 1460-byte segments. In these tests the TCP advertised window is 128 packets, assuming the use of TCP's window scale option [25].

We model the explicit trigger sent by the MN to the server at the fixed network during the hand-off procedure when the server starts to use the new path. When the trigger arrives the Quick-Start sender makes a rate request for 2 MB/sec that covers the whole path capacity in all cases. The routers may approve the request with the requested or a smaller rate, or reject the request. The routers use the *Target* algorithm at routers with 95% target utilization, and the *peak utilization* measurement method for three recent time intervals of 250 ms. The routers also remember the recent Quick-Start requests from the past 250 ms. Chapter 6 gives a detailed description of the Quick-Start algorithms and the above-mentioned parameters.

We primarily test the following four different variants of TCP:

- **none**: The standard ns-2 Sack1 TCP that does not use any explicit information about hand-offs.
- **slowstart**: The TCP sender gets a notification of vertical hand-off and sets congestion window to one MSS after vertical hand-off, after which it continues in slow-start.
- **qs**: The TCP sender gets a notification of a vertical hand-off and makes a new Quick-Start request in response. The TCP sender also makes a Quick-Start request at the beginning of the connection in this variant.

- **qsthresh**: Like *qs*, but the TCP sender sets a slow-start threshold to the value received in the Quick-Start response. After an approved Quick-Start response the congestion window and slow-start threshold have the same size.

7.4.2 Connection Startup

Figure 7.1 shows the connection startup throughput with EGPRS and WLAN link. The horizontal axis shows the length of a TCP connection (file size) and vertical axis the TCP throughput for the given amount of data. In these runs only a single TCP connection is used and no vertical hand-off between the access technologies occurs.

A couple of observations can be made from the graphs. First, the basic Quick-Start appears to slightly improve the connection start-up performance. Second, the graphs show the devastating effect of slow-start overshoot on performance, especially for moderate-sized transfers (file size roughly 200KB) on a Wireless LAN link. The graphs also show that setting the slow-start threshold based on the Quick-Start response effectively avoids the performance degradation caused by the slow-start overshoot.

Figure 7.2 shows the time-sequence diagrams for one of the cases in Figure 7.1 (file size = 188KB), where the WLAN throughput is at its worst level. The figure shows why the normal TCP performs badly: the TCP sender stays in slow-start, shooting the bottleneck queue full of packets, until it gets three duplicate ACKs as an indication of the first packet loss due to overflow of the bottleneck queue. The sender makes one retransmission, but it is not able to avoid a retransmission timeout because a significant number of packets have been lost during the slow-start overshoot and the small file size prevents the receiver from getting enough data to trigger the duplicate acknowledgments required to allow SACK recovery to proceed. Retransmission timeout is an expensive operation due to the minimum RTO value of one second, causing serious performance degradation.

The *qsthresh* variant, on the other hand, avoids the slow-start overshoot because it moves to congestion avoidance immediately after the Quick-Start phase. One can see that there are no packet losses, and the TCP connection is finished in 520 ms, versus the 1600 ms in the normal TCP case.

7.4.3 Vertical Hand-off

Figure 7.3 illustrates TCP hand-off performance from EGPRS to WLAN link with the make-before-break hand-off. The x-axis indicates the time of the hand-off, measured from the beginning of the connection. The top figure illustrates the

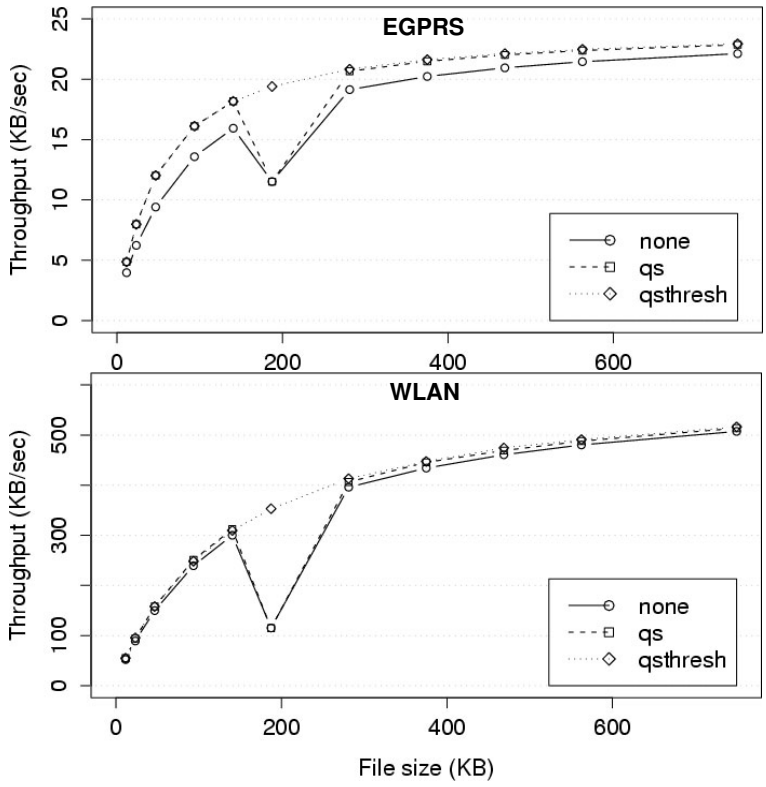


Figure 7.1: Throughput without hand-offs with different file sizes.

total long-term throughput of the TCP connections. The middle figure illustrates the number of packet losses in a 13-second test run, and the bottom figure shows the amount of data transmitted in a 3-second period following the hand-off, thus showing the TCP efficiency immediately after the hand-off event.

In these simulations the regular TCP suffers from packet reordering: packets traveling through a WLAN link arrive to the receiver before the packets sent earlier to the much slower EGPRS link, appearing as out-of-order segments that trigger duplicate acknowledgments at the receiver. Duplicate acknowledgments, in turn, trigger unnecessary fast retransmissions at the sender.

A few observations can be made from the figures. First, the basic Quick-Start suffers from bad performance when the connection has lasted more than 7 seconds before the hand-off occurs. This happens because the EGPRS link queue has become full, and a slow-start overshoot follows, causing loss of tens of packets. The TCP sender with the basic Quick-Start yields poor hand-off performance as

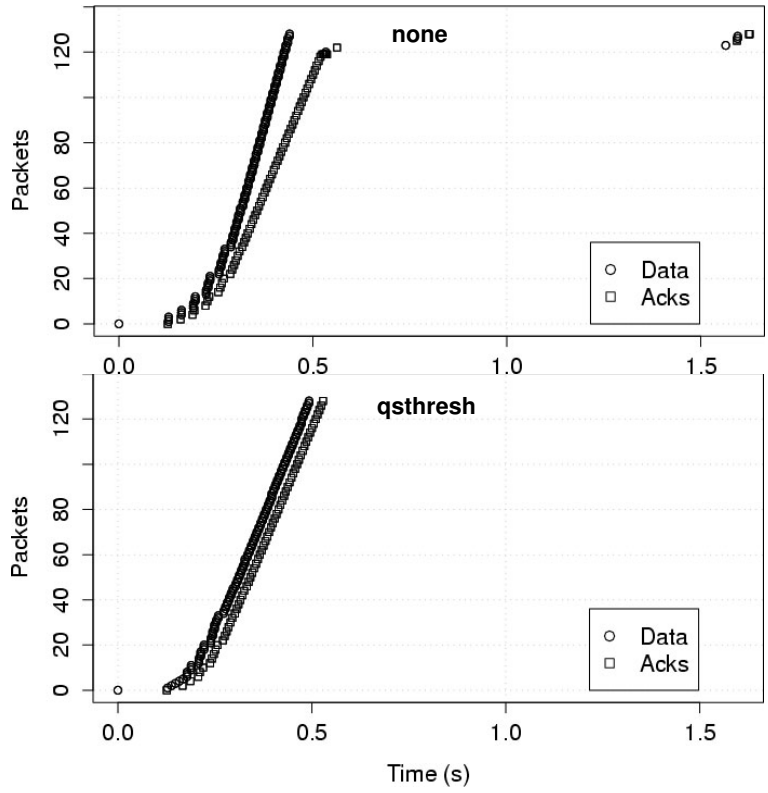


Figure 7.2: Slow-start overshoot with standard TCP and effect of qsthresh.

it further worsens the severity of the slow-start overshoot by continuing in slow-start after the hand-off, forcing the TCP sender to wait for a costly retransmission timeout to recover. In many other cases the use of the basic Quick-Start also results in several packet losses, even though the path was unutilized when the Quick-Start request was made. The *slowstart* variant performs slightly worse than the regular TCP, because the wireless link is utilized less effectively when slow-start is employed after the hand-off.

A second observation can be made on *qsthresh*. Most packet losses due to buffer overflow can be avoided with *qsthresh*, because the TCP sender is in congestion avoidance for most of the time after the initial round-trip time, and in particular after the hand-off. Therefore, the problems regarding hand-off performance and throughput can be avoided.

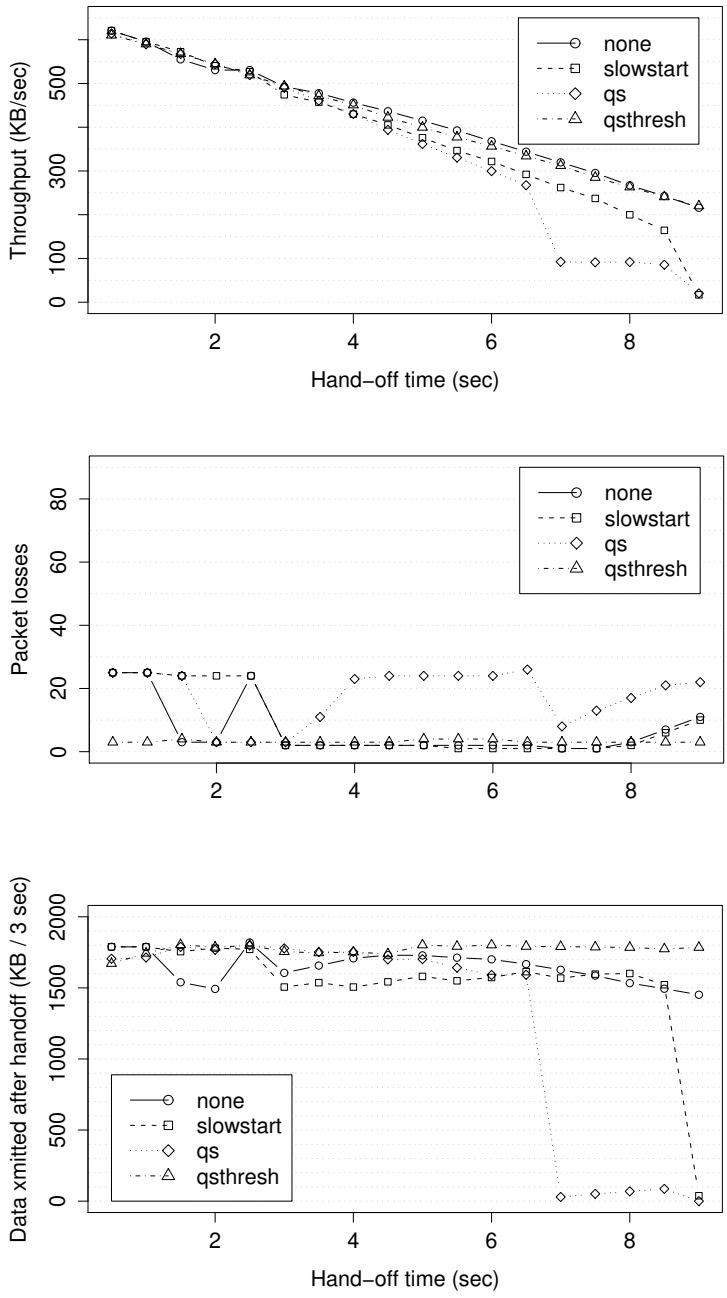


Figure 7.3: Make-before-break hand-off from EGPRS to WLAN.

Figure 7.4 illustrates the throughput of the whole data transfer, the number of packet losses during the transfer, and the amount of data transmitted in a 3-second period following the hand-off in case a break-before-make hand-off occurs from a WLAN link to a EGPRS link. The connection from the WLAN link is lost 500 milliseconds before the EGPRS link is up for transmission. All data sent from the wireless access router is lost during that time period. The figure shows that although the long-term throughput is roughly similar with different variants, both *qs* and *qsthresh* substantially improve the transmission performance after the hand-off. The *qsthresh* variant also has two to three times less packet losses than the other variants.

In the break-before-make hand-off scenario the slow-start overshoot after the hand-off is not a problem, because roughly one window's worth of segments is lost in any case due to the period of disconnection before the hand-off completes, and there is nothing the TCP sender can do to prevent this.

We also conducted simulations on WLAN to EGPRS make-before-break hand-offs. On some of the hand-off scenarios we observed packet losses that were caused due to inappropriately large congestion window, which was valid on the WLAN link, but too large for the EGPRS link. Other phenomena seen in these simulations were spurious TCP retransmission timeouts caused by a sudden increase of round-trip time after the hand-off.

7.5 Summary

In this chapter we investigated the possible benefits of using Quick-Start after a vertical hand-off that can occur with mobile multi-radio terminals. Quick-Start significantly improves the start-up performance of a connection, and it can be used to quickly resolve the correct path capacity after the vertical hand-off by using an explicit cross-layer notification to trigger the Quick-Start. However, we observed that packet losses due to the slow-start overshoot have a significant effect on connection performance on high-latency links such as EGPRS. We proposed an enhanced response to Quick-Start that also sets the slow-start threshold based on the approved Quick-Start request, thus extending the use of a Quick-Start response to limit the growth of the sending rate beyond the path capacity. This enhancement resulted in excellent results in our network environment. It should be noted that limiting the slow-start threshold may also have a negative performance effect, for example if the approved Quick-Start request does not cover the full bottleneck link capacity. Therefore, an interesting future research topic would be to explore alternative mechanisms, such as Limited Slow-Start [53].

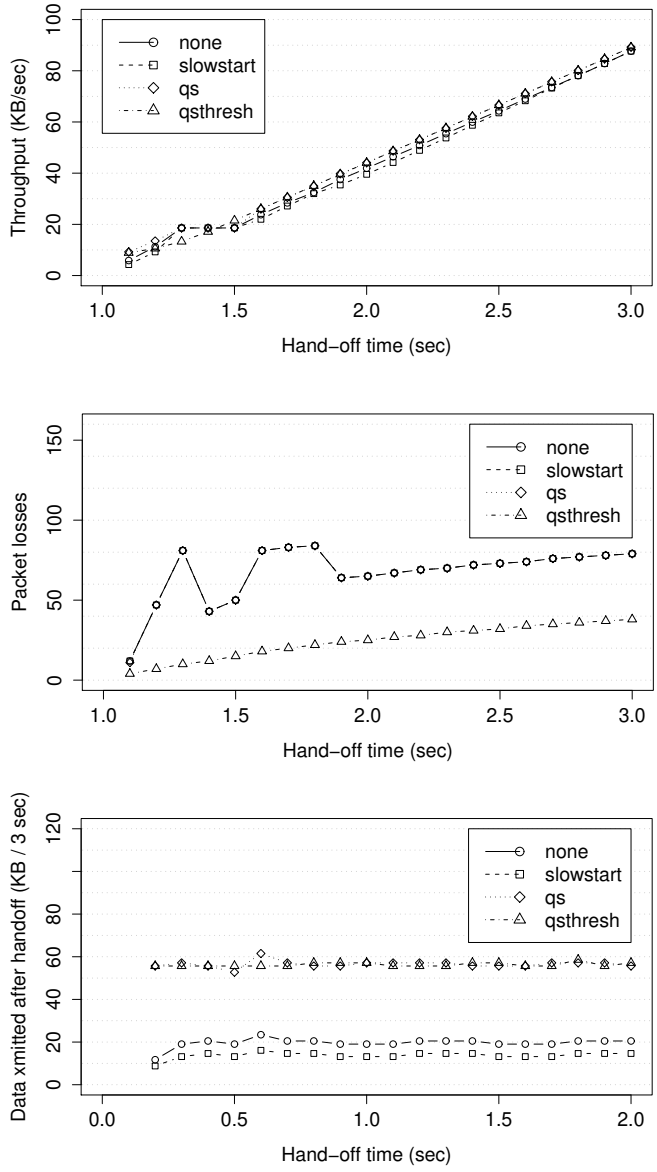


Figure 7.4: Break-before-make hand-off from WLAN to EGPRS.

While Quick-Start has a lot of potential, there are many challenges for its deployment in the Internet, such as various types of IP tunnels, or misbehaving hosts trying to exploit Quick-Start [54]. Therefore, we believe it is likely that for now Quick-Start would be useful in short-range network communication such as in enterprise intranets, or in wireless operator networks, where the challenges can be more easily controlled and dealt with. Considering that many of the wireless operator services, including Web proxies, are located in the operator's local network domain, there are benefits in introducing Quick-Start locally in these environments.

Because it is quite uncertain whether Quick-Start will ever be deployed in the worldwide Internet, its usefulness may seem limited also in wireless networks, since most services are provided outside the local operator domain. As discussed in Section 7.3, the same limitation applies to any other scheme that aims to speed up the TCP congestion control and therefore needs a permission from all routers on the network path. As we have shown that Quick-Start can bring significant performance advantages to TCP over wireless links, it seems interesting to find ways to go around this limitation. An inappropriate option would be to try to directly "cheat" the Quick-Start mechanism, for example by having a middlebox close to the egress of the operator network that modifies the contents of the Quick-Start Response option to make it seem that Quick-Start Request sent by the wireless host was approved in a case where some of the routers did not process the option. This would be a congestion control violation and therefore strongly discouraged.

A second possibility would be to place a split-connection proxy to process all TCP traffic at the wireless operator egress. The proxy would split an end-to-end connection into a part between the wireless host and the proxy, and to the part between the proxy to the other host in the Internet, in a similar way done, for example, in I-TCP [13]. If the communication path between the proxy and the fixed host would have relatively short round-trip delays compared to the delays on the wireless part of the connection, applying Quick-Start on the wireless part would help to improve the TCP performance. Having a Quick-Start proxy in the operator network would seem a relatively straight-forward deployment path, but also problematic, because the split-connection proxies are known to have several problems involved with them [24].

Even if the incorrect TTL Diff indicated that there were routers that did not process the Quick-Start Request, the incoming rate information could still be useful in avoiding the slow-start overshoot. From the reduced value of the Quick-Start Request option the sender knows that one of the routers on the connection path has indicated that it has a preferred upper bound on the transmission rate, so there is no reason to continue in slow-start beyond that limit, even if the TTL Diff value

was incorrect. If slow-start overshoot can be avoided, a number of packet losses can be prevented, which might be a good enough incentive for the end hosts and key routers to implement the Quick-Start option. On the other hand, while in this chapter we applied a single evaluation algorithm at the router, it might be more efficient to have separate evaluation algorithms for instantly available bandwidth that can be admitted for Quick-Start, and for the recommended upper bound for slow-start. Therefore it might be justifiable to have separate option values for the two uses. We leave these considerations to be investigated by future work.

CHAPTER 8

Conclusions and Future Work

This dissertation has proposed and thoroughly investigated mechanisms for improving TCP performance in network environments with challenging transmission delay behavior, such as the GPRS networks. We have primarily focused on three problems in TCP performance in these environments: (i) spurious retransmission timeouts caused by the sudden delay spikes in lower layer packet transmission, (ii) improving the slow-start, that utilizes the link capacity inefficiently in the beginning of connections on high delay-bandwidth paths, and (iii) quickly finding an appropriate sending rate after a vertical hand-off between two different link technologies. We supplemented our analysis with a thorough description of the Linux TCP implementation that was used in many of the experiments conducted for this work.

This dissertation described and analyzed the *Forward RTO-Recovery (F-RTO)* algorithm that can be applied at the TCP sender to detect spurious retransmission timeouts and thus avoid unnecessary retransmissions and congestion control actions that a spurious retransmission timeout would cause. We analyzed F-RTO in different network scenarios to validate its robustness in different kinds of networks and evaluated different alternatives for responding to spurious retransmission timeout, and showed that F-RTO is effective in avoiding the negative effects of spurious retransmission timeouts. We also discussed a SACK-based enhancement of F-RTO and a few limitations F-RTO has.

As with many other TCP problems, it is difficult to quantify how severe problem spurious retransmission timeouts are in live wireless networks. We have referred to earlier research that analyzed link behavior in a GPRS network and observed spurious timeouts in the measurements, but the link behavior depends on many factors that are unknown or difficult model, such as the network configuration used by the operator, or the movement patterns of the mobile device.

However, several network device vendors have had interest in solving the problem of spurious timeouts in some way, which indicates that they have observed similar behavior also in their networks using GPRS and other wireless technologies. Although there are alternative ways to improve performance on spurious retransmission timeouts, it appears that many of the big operating system vendors have chosen F-RTO as their solution. In addition to the Linux implementation made by the author, there already are several commercial implementations of F-RTO. Some companies have also requested F-RTO to be made a Proposed Standard in the IETF. Finally, we note that it is important that F-RTO detects spurious timeouts using TCP's own mechanisms, and can therefore be useful also in other contexts than wireless networks. Equally important is that we believe F-RTO to not harm TCP performance in any case, even in networks where spurious retransmission timeouts are a rare phenomenon.

The second part of this dissertation evaluated the Quick-Start algorithm, a cooperative effort between the TCP end-hosts and the routers to quickly establish the available bandwidth on the network path and thus instantly find appropriate values for TCP's congestion control parameters. We evaluated various different router algorithms and settings, and discussed the possible deployment and security threats such a scheme may have.

While we acknowledge that deployment of Quick-Start in real networks is very challenging, we believe this work is a useful contribution to the ongoing discussion on the next generation of network resource and congestion control. There have been increasing number of proposals in the EU and USA to take actions on revising the Internet architecture to better accommodate today's needs that were not envisioned when the core protocols were designed. One of the features that are under pressure to be changed is the current congestion control model that is based on the use of minimal information, being slow to react to rapid changes in the path characteristics and relying on the honesty of the end hosts that might have conflicting interests to send faster than what the congestion control rules allow. If a change becomes possible, it needs to be carefully designed, because the core Internet protocols have had tendency to last at least for a duration of one human generation. We believe that the lessons learned during this work, for example related to deployability and trustworthiness of the explicit network congestion information, are helpful when considering the future network congestion control algorithms.

The third part of this dissertation applied the Quick-Start algorithm in the context of vertical hand-offs between wireless technologies such as GPRS and WLAN. These environments are challenging for TCP, because they have very different bandwidth and delay characteristics, causing TCP's slowly converging

congestion control parameters to have inappropriate values after a vertical hand-off. With the Quick-Start algorithm applied after the vertical hand-off, significant performance improvements were achieved.

Because wireless networks often have specific problems, there are benefits to have such enhancements and mitigations that could be implemented on the wireless host or at the wireless network. While we showed Quick-Start to be effective, it is a demanding mechanism due to the requirement of being supported by every router on the connection path. On the other hand, in order to instantly start sending at a high rate there needs to be some procedure to ensure that the flow does not cause severe network congestion. We believe this is not possible without having some information from all of the routers on the connection path that have the potential to get congested. We briefly discussed about the possibility to use a Quick-Start proxy close to the wireless link, and further investigation of the benefits and costs of such arrangement could be an useful topic of future work.

While the above-mentioned technologies have been evaluated in the context of TCP, the same principles can be applied to other transport protocols. For example, F-RTO can be applied to *Stream Control Transmission Protocol (SCTP)* [153], which is a new transport protocol with similar algorithms to TCP. Quick-Start can also be used with SCTP, and to establish a correct sending rate in *Datagram Congestion Control Protocol (DCCP)* [99] with its two congestion control profiles, window-based congestion control [59], and TCP-friendly rate control [60, 55].

The research presented in this dissertation could be followed up by different research topics:

- **Revising TCP's retransmission algorithm.** F-RTO and the many other proposals to improve TCP's performance are incremental modifications to TCP's base algorithms. After being appended with a number of such algorithms in the past decades, one could claim that today's TCP implementations and specifications are patchy chunks of code, and it might be useful to try to invent a completely new retransmission algorithm that works better in today's heterogeneous network environment, without having to carry the legacy of TCP. An interesting question is, would this new algorithm be totally different from the current TCP algorithms, or would it end up rather similar.

- **Investigating the range of explicit congestion control mechanisms.** Quick-Start is a small modification to TCP to employ explicit cross-layer communication between the TCP end-hosts and the network. Recently there have been other related proposals from just slightly extending ECN to deploying a full-fledged explicit congestion control protocol. A compar-

ative analysis of the powerfulness of different mechanisms would be useful to gain knowledge about the possibilities of different approaches and the challenges such mechanisms have to face.

- **Investigating a common framework for future cross-layer communication mechanisms.** When investigating Quick-Start, we identified certain deployment and security challenges involved with it, as discussed in Chapter 6. It seems possible that many of these challenges are common to a wider range of similar explicit mechanisms. Therefore a possibly useful exercise would be to seek for a common framework for in-band explicit light-weight signaling, to be used as a basis in specifying the future revisions of the Internet protocols, for example related to IP tunneling.

A bigger issue behind the individual topics discussed in this dissertation and the future work items listed above is how strictly future Internet design should still stick to the traditional end-to-end principle and end-host-based congestion control that has been applied in TCP and the other transport protocols. In this dissertation we investigated one mechanism that is purely a TCP-sender-based solution, and another mechanism that requires collaboration from the network routers. There are many recent research ideas that require active participation by the network to support efficient data transfer, but a careful consideration should be taken regarding the compromises it might cause to network scalability and robustness.

The possibilities to improve TCP and other transport protocols in the current Internet architecture just by making end-host modifications are limited. Therefore the author would like to encourage the future research to be ambitious in challenging the current assumptions in the Internet design, and fearlessly exploit radical ideas in the search of substantial advances to Internet communication technologies for the future generations.

References

[1] 3GPP. System Architecture Evolution (Release 7). 3GPP TR 23.882 V1.2.3, June 2006.

[2] N. Abramson. The ALOHA System – Another Alternative for Computer Communications. In *1970 Fall Joint Computer Conference*, volume 37 of *AFIPS Conference Proceedings*, pages 281–285, Houston, TX, USA, November 1970.

[3] N. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31(2):119–123, March 1985.

[4] T. Alanko, M. Kojo, H. Laamanen, M. Liljeberg, M. Moilanen, and K. Raatikainen. Measured Performance of Data Transmission Over Cellular Telephone Networks. *ACM SIGCOMM Computer Communication Review*, 24(5):24–44, 1994.

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

[6] M. Allman. TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465, February 2003.

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

[8] M. Allman and E. Blanton. Notes on Burst Mitigation for Transport Protocols. *ACM SIGCOMM Computer Communication Review*, 35(2):53–60, April 2005.

[9] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, October 2002.

[10] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *Proceedings of ACM SIGCOMM '99*, pages 263–274, Cambridge, MA, USA, September 1999.

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

[12] M. Baker, X. Zhao, S. Cheshire, and J. Stone. Supporting Mobility in MosquitoNet. In *Proceedings of the USENIX 1996 conference*, San Diego, CA, USA, January 1996.

[13] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 136–143. IEEE, May 1995.

[14] H. Balakrishnan, H.S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of ACM SIGCOMM '99*, pages 175–187, Cambridge, MA, USA, September 1999.

[15] H. Balakrishnan and S. Seshan. The Congestion Manager. RFC 3124, June 2001.

[16] H. Balakrishnan, S. Seshan, and R. H. Katz. Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks. *Wireless Networks (Springer)*, 1(4):469–481, December 1995.

[17] J.C.R. Bennett, C. Partridge, and N. Shectman. Packet Reordering Is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, December 1999.

[18] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. IETF RFC 1945, May 1996.

[19] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of ACM SIGCOMM '94*, pages 212–225, London, UK, August 1994.

[20] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 32(1):20–30, January 2002.

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

[22] E. Blanton and M. Allman. On the Impact of Bursting on TCP Performance. In *Proceedings of the Workshop for Passive and Active Measurement*, March 2005.

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

[24] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

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

[26] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989.

[27] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205, September 1997.

[28] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected areas in Communications*, 13(8):1465–1480, October 1995.

[29] G. Brasche and B. Walke. Concepts, services and protocols of the new GSM phase 2+ general packet radio service. *IEEE Communications Magazine*, 35(8):94–104, August 1997.

[30] L. Breslau, S. Jamin, and S. Shenker. Comments on the Performance of Measurement-Based Admission Control Algorithms. In *IEEE Infocom 2000*, volume 3, pages 1233–1242, Anaheim, CA, USA, March 2000.

[31] B. Briscoe, A. Jacquet, C. Di Cairano-Gilfedder, A. Salvatori, A. Soppera, and M. Koyabe. Policing Congestion Response in an Internetwork Using Re-Feedback. In *Proceedings of ACM SIGCOMM 2005*, pages 277–288, Philadelphia, PA, USA, August 2005.

[32] K. Brown and S. Singh. M-TCP: TCP for Mobile Cellular Networks. *ACM SIGCOMM Computer Communication Review*, 27(5):19–43, October 1997.

[33] R. Caceres and L. Iftode. The Effects of Mobility on Reliable Transport Protocols. In *14th International Conference on Distributed Computer Systems*, pages 12–20, Poznan, Poland, June 1994. IEEE.

[34] R. Caceres and L. Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environment. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[35] J. Cai and D. J. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, 35(10):122–131, October 1997.

[36] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of ACM Mobicom 2001*, pages 287–297, Rome, Italy, July 2001.

[37] V.G. Cerf and R.E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions of Communications*, 22(5):637–648, May 1974.

[38] D. D. Clark. Window and Acknowledgement Strategy in TCP. RFC 813, July 1982.

[39] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of ACM SIGCOMM '88*, pages 106–114, Stanford, CA, USA, August 1988.

[40] B.P. Crow, I. Widjaja, J.G. Kim, and P.T. Sakai. IEEE 802.11 Wireless Local Area Networks. *IEEE Communications Magazine*, 35(9):116–126, September 1997.

[41] S. Dawkins, G. Montenegro, M. Kojo, and V. Magret. End-to-end Performance Implications of Slow Links. RFC 3150, July 2001.

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

[43] L. Dimopoulou, G. Leoleis, and I. Venieris. Fast Handover Support in a WLAN Environment: Challenges and Perspectives. *IEEE Network*, 19(3):14–20, May 2005.

[44] L. Dryburgh and J. Hewett. *Signaling System No. 7 (SS7/C7): protocol, architecture, and services*. Cisco Press, 2005.

[45] D. Duchamp and N. Reynolds. Measured Performance of a Wireless LAN. In *Proceedings of the 17th Conference on Local Computer Networks*, pages 494–499. IEEE, September 1992.

[46] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614, September 2006.

[47] P. Eronen (ed.). IKEv2 Mobility and Multihoming Protocol (MOBIKE). RFC 4555, June 2006.

[48] W. Eddy and Y. Swami. Adapting End Host Congestion Control for Mobility. Technical Report CR-2005-213838, NASA Glenn Research Center, September 2005.

[49] B. Carpenter (editor). Architectural Principles of the Internet. RFC 1958, June 1996.

[50] G. Fairhurst and L. Wood. Advice to link designers on link Automatic Repeat reQuest (ARQ). RFC 3366, August 2002.

[51] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3):5–21, July 1996.

[52] S. Floyd. Congestion Control Principles. RFC 2914, September 2000.

[53] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742, March 2004.

[54] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782, January 2007.

[55] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proceedings of ACM SIGCOMM 2000*, pages 43–56, Stockholm, Sweden, August 2000.

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

[57] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004. Standards Track.

[58] S. Floyd and E. Kohler. Internet Research Needs Better Models. *ACM SIGCOMM Computer Communication Review*, 33(1):29–34, January 2003.

[59] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341, March 2006.

[60] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342, March 2006.

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

[62] J. Geier. *Wireless LANs – Implementing Interoperable Networks*. Macmillan Technical Publishing, first edition, 1999.

[63] A. Gurtov and J. Korhonen. Effect of Vertical Handovers on Performance of TCP-Friendly Rate Control. *ACM Mobile Computing and Communications Review*, 8(3):73–87, July 2004.

[64] A. Gurtov and R. Ludwig. Evaluating the Eifel Algorithm for TCP in a GPRS Network. In *Proceedings of European Wireless '02*, February 2002.

[65] A. Gurtov and R. Ludwig. Response to Spurious Retransmission Timeouts. In *Proceedings of IEEE Infocom 2003*, volume 3, pages 2312–2322, San Francisco, CA, USA, March 2003.

[66] A. Gurtov, M. Passoja, O. Aalto, and M. Raitola. Multi-Layer Protocol Tracing in a GPRS Network. In *Proceedings of IEEE Vehicular Technology Conference*, volume 3, pages 1612–1616, Vancouver, Canada, September 2002.

[67] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861, June 2000.

[68] W. Hansmann and M. Frank. On Things to happen during a TCP Handover. In *Proceedings of the 28th Annual Conference on Local Computer Networks, LCN'03)*, pages 109–118, Bonn, Germany, October 2003.

[69] R. Hermann. IEEE 802.11 Wireless LAN Standard: A Technical Tutorial. Technical Report RZ 3186, IBM Research, November 1999.

[70] S. Hirata, A. Nakajima, and H. Uesaka. PDC Mobile Packet Data Communication Network. In *Proceedings of 4th IEEE International Conference on Universal Personal Communications*, pages 644–648, November 1995.

[71] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, June 1995.

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

[73] H. Honkasalo, K. Pehkonen, M. Niemi, and A. Leino. WCDMA and WLAN for 3G and Beyond. *IEEE Wireless Communications*, 9(2):14–18, April 2002.

[74] H. Huang and J. Cai. Improving TCP Performance during Soft Vertical Handoff. In *Proceedings of the 19th International Conference on Advanced In formation Networking and Applications (AINA'05)*, volume 2, pages 329–332, March 2005.

[75] IEEE Computer Society. IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 4: Further Higher Data Rate Extension in the 2.4 GHz Band. IEEE Std 802.11g-2003, June 2003.

[76] IEEE-SA Standards Board. Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. ANSI/IEEE Std 802.11, 1999 Edition (R2003), March 1999.

[77] IEEE-SA Standards Board. Supplement to IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band. IEEE Std 802.11b-1999 (R2003), September 1999.

[78] IEEE-SA Standards Board. Supplement to IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: High-speed Physical Layer in the 5 GHz Band. IEEE Std 802.11a-1999 (R2003), September 1999.

[79] H. Inamura, G. Montenegro, R. Ludwig, A. Gurtov, and F. Khafizov. TCP over Second (2.5G) and Third (3G) Generation Wireless Networks. RFC 3481, February 2003.

[80] J. Ioannidis, D. Duchamp, and G. Q. Maguire, Jr. IP-based Protocols for Mobile Internetworking. In *Proceedings of ACM SIGCOMM '91*, pages 235–243, Zürich, Switzerland, September 1991.

[81] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

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

[83] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. E-mail to the end-to-end interest mailing list, 30th April 1990, April 1990.

[84] H. Jiang and C. Dovrolis. Passive Estimation of TCP Round-Trip Times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, July 2002.

[85] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, June 2004.

[86] H. Kaaranen, A. Ahtiainen, L. Laitinen, S. Naghian, and V. Niemi. *UMTS Networks – Architecture, Mobility and Services*. Wiley, 2001.

[87] P. Karn. MACA - A New Channel Access Method for Packet Radio. In *Proceedings of the 9th ARRL Computer Networking Conference*, London, Ontario, Canada, 1990.

[88] P. Karn and C. Partridge. Improving Round-Trip Estimates in Reliable Transport Protocols. In *Proceedings of ACM SIGCOMM '87*, pages 2–7, New York, NY, USA, August 1987.

[89] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of ACM SIGCOMM 2002*, pages 89–102, Pittsburgh, PA, USA, August 2002.

[90] D. Katz. IP Router Alert Option. RFC 2113, February 1997.

[91] J. Kempf, et. al. Problem Statement for Network-Based Localized Mobility Management (NETLMM). RFC 4830, April 2007.

[92] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, November 1998.

[93] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[94] F. Khafizov and M. Yavuz. Running TCP over IS-2000. In *Proceedings of IEEE ICC 2002*, volume 5, pages 3444–3448, April 2002.

[95] S. Khurana, A. Kahol, and A.P. Jayasumana. Effect of Hidden Terminals on the Performance of IEEE 802.11 MAC Protocol. In *In Proceedings of 23rd Annual Conference on Local Computer Networks (LCN '98)*, pages 12–20, October 1998.

[96] L. Kleinrock and S. Lam. Packet Switching in a Slotted Satellite Channel. In *AFIPS Conference Proceedings*, volume 42, pages 703–710, National Computer Conference, New York, NY, June 1973.

[97] D. Knisely, S. Kumar, S. Laha, and S. Nanda. Evolution of Wireless Data Services: IS-95 to cdma2000. *IEEE Communications Magazine*, 36(10):140–149, October 1998.

[98] E. Kohler. Generalized Connections in the Datagram Congestion Control Protocol. Internet draft "draft-kohler-dccp-mobility-02", June 2006. Expired.

[99] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, March 2006.

[100] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a Wireless Network Emulator. In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, pages 151–166, Aachen, Germany, September 2001. VDE Verlag.

[101] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko. An Efficient Transport Service for Slow Wireless Links. *IEEE Journal on Selected Areas In Communications*, 15(7):1337–1348, September 1997.

[102] 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.

[103] R. Koodli. Fast Handovers for Mobile IPv6. RFC 4068, July 2005.

[104] J. Korhonen, O. Aalto, A. Gurtov, and H. Laamanen. Measured Performance of GSM HSCSD and GPRS. In *Proceedings of the IEEE International Conference on Communications*, volume 5, pages 1330–1334, Helsinki, Finland, June 2001.

[105] J. Korhonen, S. Park, J. Zhang, C. Hwang, and P. Sarolahti. Link Characteristic Information for IP Mobility Problem Statement. Internet-Draft "draft-korhonen-mobopts-link-characteristics-ps-01.txt", June 2006. Expired.

[106] S. Kunniyur. AntiECN Marking: A Marking Scheme for High Bandwidth Delay Connections. In *Proceedings of IEEE ICC '03*, volume 1, pages 647–651, Anchorage, Alaska, May 2003.

[107] S. Ladha, S. Baucke, R. Ludwig, and P.D. Amer. On Making SCTP Robust to Spurious Retransmissions. *ACM SIGCOMM Computer Communication Review*, 34(2):123–135, April 2004.

[108] J. Lehenkari and R. Miettinen. Standardisation in the construction of a large technological system – the case of the Nordic mobile telephone system. *Telecommunications Policy (Elsevier)*, 26(3–4):109–127, April 2002.

[109] R. Love, A. Ghosh, W. Xiao, and R. Ratasuk. Performance of 3GPP High Speed Downlink Packet Access (HSDPA). In *Proceedings of IEEE 60th Vehicular Technology Conference*, volume 5, pages 3359–3363, September 2004.

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

[111] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM SIGCOMM Computer Communication Review*, 30(1):30–36, January 2000.

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

[113] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A. Joseph. Multi-Layer Tracing of TCP over a Reliable Wireless Link. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27, pages 144–154, New York, May 1999.

[114] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM SIGCOMM Computer Communication Review*, 30(3):17–27, July 2000.

[115] J. Manner and M. Kojo. Mobility Related Terminology. RFC 3753, June 2004.

[116] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *Proceedings of ACM SIGCOMM '96*, pages 281–291, Palo Alto, CA, USA, October 1996.

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

[118] A. Medina, M. Allman, and S. Floyd. Measuring Interactions Between Transport Protocols and Middleboxes. In *ACM SIGCOMM/USENIX Internet Measurement Conference*, Taormina, Sicily, Italy, October 2004.

[119] M. Meyer. TCP Performance over GPRS. In *IEEE Wireless Communications and Networking Conference (WCNC)*, volume 3, pages 1248–1252, September 1999.

[120] Microsoft. Performance Enhancements in the Next Generation TCP/IP Stack. Microsoft TechNet article, available at: http://www.microsoft.com/technet/community/columns/cableguy/cg1105.mspx, November 2005.

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

[122] NS Simulator. URL http://www.isi.edu/nsnam/ns/.

[123] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of ACM SIGCOMM 1998*, pages 303–314, Vancouver, Canada, September 1998.

[124] V. Padmanabhan and R. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In *IEEE Globecom Internet Mini-conference*, Sydney, Australia, November 1998.

[125] C. Partridge and A. Jackson. IPv6 Router Alert Option. RFC 2711, October 1999.

[126] C. Partridge, D. Rockwell, M. Allman, R. Krishnan, and J. Sterbenz. A Swifter Start for TCP. Technical Report 8339, BBN Technologies, 2002.

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

[128] C. Perkins. IP Encapsulation within IP. RFC 2003, October 1996.

[129] C. Perkins (ed.). IP Mobility Support for IPv4. RFC 3344, August 2002.

[130] J. Postel. User Datagram Protocol. IETF RFC 768, August 1980.

[131] J. Postel. Internet Control Message Protocol. RFC 792, September 1981.

[132] J. Postel. Internet Protocol. IETF RFC 791, September 1981.

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

[134] P. Nikander R. Moskowitz. Host Identity Protocol (HIP) Architecture. IETF RFC 4423, May 2006.

[135] M. Rahnema. Overview of the GSM system and protocol architecture. *IEEE Communications Magazine*, 31:92–100, April 1993.

[136] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

[137] L. Roberts. Extensions of Packet Communication Technology to a Hand Held Personal Terminal. In *1972 Spring Joint Computer Conference*, volume 40 of *AFIPS Conference Proceedings*, pages 295–298, Atlantic City, NJ, USA, May 1972.

[138] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[139] J. Rosenberg, H. Schulzrinne, and G. Camarillo. The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP). RFC 4168, October 2005.

[140] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[141] P. Sarolahti. Performance Analysis of TCP Improvements for Congested Reliable Wireless Links. Master's thesis, University of Helsinki, Department of Computer Science, Series of Publications C, No. C-2001-8, February 2001.

[142] P. Sarolahti. Congestion Control on Spurious TCP Retransmission Timeouts. In *Proceedings of IEEE Globecom 2003*, volume 2, pages 682–686, San Francisco, CA, USA, December 2003.

[143] P. Sarolahti, M. Allman, and S. Floyd. Determining an Appropriate Sending Rate Over an Underutilized Network Path. *Computer Networks (Elsevier)*, 51(7):1815–1832, May 2007.

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

[145] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: an Enhanced Recovery Algorithm for TCP Retransmission Timeouts. *ACM SIGCOMM Computer Communication Review*, 33(2):51–63, April 2003.

[146] P. Sarolahti, J. Korhonen, L. Daniel, and M. Kojo. Using Quick-Start to Improve TCP Performance with Vertical Hand-offs. In *Proceedings of IEEE Local Computer Networks conference (LCN 2006), Wireless Local Networks workshop*, pages 897–904, Tampa, FL, USA, November 2006.

[147] P. Sarolahti and A. Kuznetsov. Congestion Control in Linux TCP. In *Proceedings of Usenix 2002/Freenix Track*, pages 49–62, Monterey, CA, USA, June 2002.

[148] S. Schütz, L. Eggert, W. Eddy, Y. Swami, and K. Le. TCP Response to Lower-Layer Connectivity-Change Indications. Internet-Draft "draft-schuetz-tcpm-tcp-rlci-01", March 2007. Work in progress.

[149] S. Schütz, L. Eggert, S. Schmid, and M. Brunner. Protocol Enhancements for Intermittently Connected Hosts. *ACM SIGCOMM Computer Communication Review*, 35(3):5–17, July 2005.

[150] E. Seurre, P. Savelli, and P.-J. Pietri. *EDGE for Mobile Internet*. Artech House, 2003.

[151] M. Stemm and R.H. Katz. Vertical Handoffs in Wireless Overlay Networks. *Mobile Networks and Applications (Springer)*, 3(4):335–350, January 1998.

[152] W. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, 1995.

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

[154] R. Stewart, et. al. Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration. Internet-Draft "draft-ietf-tsvwg-addip-sctp-17.txt", November 2006. Work in progress.

[155] P. Stuckmann and J. Franke. Performance Characteristics of the Enhanced General Packet Radio Service for the Mobile Internet Access. In *Proceedings of 2nd International Conference on 3G Mobile Communication Technologies*, pages 287–291. IEEE, March 2001.

[156] S. Sundarrajan and J. Heidemann. Study of TCP Quick-Start with NS-2. Unpublished report, University of South California, 2002.

[157] Y. Swami and K. Le. Decorrelated Loss Recovery (DCLOR) Using SACK Option for Spurious Timeouts. Internet draft "draft-swami-tsvwg-tcp-dclor-07.txt", February 2006. Expired Internet-Draft.

[158] Y. Swami, K. Le, and W. Eddy. Lightweight Mobility Detection and Response (LMDR) Algorithm for TCP. Internet-Draft "draft-swami-tcp-lmdr-07.txt", February 2006. Work in progress.

[159] K. Tan and Q. Zhang. STODER: A Robust and Efficient Algorithm for Handling Spurious Retransmit Timeouts in TCP. In *Proceedings of IEEE Globecom 2005*, volume 6, pages 3692–3696, November 2005.

[160] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International, third edition, 1996.

[161] J. Touch. TCP Control Block Interdependence. RFC 2140, April 1997.

[162] B. Tuch. Development of WaveLAN, an ISM Band Wireless LAN. *AT&T Technical Journal*, 72(4):27–37, July 1993.

[163] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One More Bit Is Enough. In *SIGCOMM 2005*, pages 37–48, Philadelphia, PA, USA, August 2005.

[164] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. In *Proceedings of IEEE Infocom 2004*, volume 4, pages 2514–2524, Hong Kong, China, March 2004.

[165] G. Xylomenos, G. Polyzos, P. Mähönen, and M. Saaranen. TCP Performance Issues over Wireless Links. *IEEE Communications Magazine*, 39(4):52–58, April 2001.

[166] G. Xylomenos and G.C. Polyzos. TCP and UDP Performance over a Wireless LAN. In *Proceedings of IEEE Infocom '99*, volume 2, pages 439–446, New York, NY, USA, March 1999.

[167] K. Yamamoto, H. Suzuki, N. Ishikawa, A. Hokamura, K. Sekiguchi, and Y. Suwa. Effects of F-RTO and Eifel Response Algorithms for W-CDMA and HSDPA networks. In *Proceedings of Wireless Personal Multimedia Communications(WPMC)'05*, September 2005.

[168] X. Yang. IEEE 802.11n: Enhancements for Higher Throughput in Wireless LANs. *IEEE Wireless Communications*, 12(6):82–91, December 2005.

[169] R. Yavatkar and N. Bhagawat. Improving End-to-end Performance of TCP over Mobile Internetworks. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 146–152, December 1994.

[170] M. Yavuz and F. Kahfizov. TCP over Wireless Links with Variable Bandwidth. In *Proceedings of the IEEE Vehicular Technology Conference (VTC'02 Fall)*, volume 3, pages 1322–1327, September 2002.

[171] P. Young. *Recursive Estimation and Time-Series Analysis*, pages 60–65. Springer-Verlag, 1984.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from:
Kumpula Science Library, P.O. Box 64, FIN-00014 University of Helsinki, FINLAND.

A-2003-5 J. Kurhila: Considering individual differences in computer-supported special and elementary education. 135 pp. (Ph.D. thesis)

A-2003-6 V. Mäkinen: Parameterized approximate string matching and local-similarity-based point-pattern matching. 144 pp. (Ph.D. thesis)

A-2003-7 M. Luukkainen: A process algebraic reduction strategy for automata theoretic verification of untimed and timed concurrent systems. 141 pp. (Ph.D. thesis)

A-2003-8 J. Manner: Provision of quality of service in IP-based mobile access networks. 191 pp. (Ph.D. thesis)

A-2004-1 M. Koivisto: Sum-product algorithms for the analysis of genetic risks. 155 pp. (Ph.D. thesis)

A-2004-2 A. Gurtov: Efficient data transport in wireless overlay networks. [B 141 pp. (Ph.D. thesis)

A-2004-3 K. Vasko: Computational methods and models for paleoecology. 176 pp. (Ph.D. thesis)

A-2004-4 P. Sevon: Algorithms for Association-Based Gene Mapping. 101 pp. (Ph.D. thesis)

A-2004-5 J. Viljamaa: Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. 206 pp. (Ph.D. thesis)

A-2004-6 J. Ravantti: Computational Methods for Reconstructing Macromolecular Complexes from Cryo-Electron Microscopy Images. 100 pp. (Ph.D. thesis)

A-2004-7 M. Kääriäinen: Learning Small Trees and Graphs that Generalize. 45+49 pp. (Ph.D. thesis)

A-2004-8 T. Kivioja: Computational Tools for a Novel Transcriptional Profiling Method. 98 pp. (Ph.D. thesis)

A-2004-9 H. Tamm: On Minimality and Size Reduction of One-Tape and Multitape Finite Automata. 80 pp. (Ph.D. thesis)

A-2005-1 T. Mielikäinen: Summarization Techniques for Pattern Collections in Data Mining. 201 pp. (Ph.D. thesis)

A-2005-2 A. Doucet: Advanced Document Description, a Sequential Approach. 161 pp. (Ph.D. thesis)

A-2006-1 A. Viljamaa: Specifying Reuse Interfaces for Task-Oriented Framework Specialization. 285 pp. (Ph.D. thesis)

A-2006-2 S. Tarkoma: Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching. 198 pp. (Ph.D. thesis)

A-2006-3 M. Lehtonen: Indexing Heterogeneous XML for Full-Text Search. 185+3pp.(Ph.D. thesis).

A-2006-4 A. Rantanen: Algorithms for ^{13}C Metabolic Flux Analysis. 92+73pp.(Ph.D. thesis).

A-2006-5 E. Terzi: Problems and Algorithms for Sequence Segmentations. 141 pp. (Ph.D. Thesis).