TCP Performance in the Presence of Congestion and Corruption Losses

Andrei Gurtov

Master's Thesis
Department of Computer Science
UNIVERSITY OF HELSINKI

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Science	Dept. of Computer Science

Tekijä — Författare — Author
Andrei Gurtov

Työn nimi — Arbetets titel — Title
TCP Performance in the Presence of Congestion and Corruption Losses

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	December 2000	76 p.

Tiivistelmä — Referat — Abstract

The wireless environment of slow and lossy links presents a challenge for efficient data transport. We have performed an experimental evaluation of TCP in an emulated wireless environment. We consider a network model including a lossy wireless link and a last-hop router with a limited-size buffer. We have explored how well the state-of-art TCP perform, identified key reasons behind the behavior, and measured the effect of different optimizations. We experimented with TCP connections with different values of the initial window, receiver window, with or without SACK and New Reno over the emulated network with different error rates and buffer sizes. The experimental data is obtained with a state-of-art TCP implementation of the Linux operating system and a real-time network emulator Seawind. Our main result is a comparative study and analysis of different TCP optimizations.

Computing Reviews Classification:
C.2.1 (Network Architecture and Design): Wireless Communication,
C.4 (Performance of Systems)

Avainsanat — Nyckelord — Keywords
Wireless networks, mobile computing, performance, TCP, error losses

Säilytyspaikka — Förvaringsställe — Where deposited
Library of the Dept. of Computer Science, Report C–2000–

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

The number of nomadic users that access the Internet using wireless technology grows rapidly. Soon, in the upcoming era of mobile computing, every portable device will have a wireless interface and an IP address. With all advantages, mobile computing introduces an environment quite different from the one found in fixed networks, with limitations that come from physical properties of the wireless medium. The scarce radio bandwidth allows for a rather low link speed; miscellaneous external factors like fading of the radio signal may cause loss of data on the radio path. In a cellular radio network the mobility is accomplished by changing a cell that serves the user, according to the user's current location. The handover process may cause data losses and a drastic change in the available service, when the user moves from a less busy to a more occupied cell. Improving the service of wireless networks is a complex task bound by the amount of available radio resources. We believe that in the future, wireless connections will be widely used, but they will remain a different environment from wireline networks.

Many popular Internet applications including World-Wide Web (WWW), File Transfer Protocol (FTP) and email require reliable data delivery over the network. The Transmission Control Protocol (TCP) is the most widely used transport protocol for this purpose; traffic studies in the Internet report that the dominant fraction of the traffic belongs to TCP [TMW97]. TCP was designed and tuned to perform well in fixed networks, where the key functionality is to utilize the available bandwidth and avoid overloading the network. However, nomadic users want to run their favorite applications that are built on TCP over a wireless connection, as well. Packet losses due to transmission errors, a long latency and sudden delays occurring on the wireless link may confuse TCP and yield a throughput far from the available line rate. Optimizing TCP for a wireless environment has been an active research area for the last few years.

This thesis presents an experimental evaluation of TCP in an emulated wireless environment. We consider a network model including a lossy wireless link and a last-hop router with a limited-size buffer. Our goal is to explore how well the state-of-art TCP performs in this environment, what are the key reasons behind the behavior, and what is the effect of different TCP optimizations. We experiment with multiple error rates and router buffer sizes over TCP connections with different optimizations. In the experiments the network is represented with a real-time network emulator Seawind [AGKM98] and the real data communication using TCP. We have used the state-of-art TCP implementation of the Linux OS. Our main result is a comparative study of performance of different TCP optimizations. We also present a list of detected implementation faults, discuss anomalies in performance and give a detailed analysis of interesting cases.

The rest of the thesis is organized as follows: in Section 2 we describe the Transmission Control Protocol, the assumed network architecture, the properties of wireless links and review the related work. In Section 3 we give specific performance problems we focus on. Section 4 lists the relevant optimizations documented by IETF. Section 5 specifies the network and workload model. In Section 6 we present our measurement setup and in Section 7 we illustrate and analyze the results of our experiments.

2 Background and Related Work

2.1 Transmission Control Protocol

2.1.1 Overview

The Transmission Control Protocol (TCP) [Pos81, Bra89, APS99] is the most used transport protocol in the Internet. TCP provides applications with reliable byte-oriented delivery of data on the top of the Internet Protocol (IP). TCP sends user data in *segments* not exceeding the Maximum Segment Size (MSS) of the connection. MSS is negotiated during the connection establishment procedure known as the three-way handshake. To open a connection the client transmits a SYN segment, the server replies with its SYN and the client replies with a SYN-ACK segment. After that the connection is established and data can be transmitted in both directions. When all data is sent, the client and the server exchange FIN and FIN-ACK segments to terminate the connection.

Each byte of the data is assigned a unique sequence number. The receiver sends an acknowledgment (ACK) upon reception of a segment. TCP acknowledgments are cumulative; an ACK confirms all bytes up to the given sequence number. The sender has no information whether some of the data beyond the acknowledged byte has been received. TCP has an important property of *self-clocking*; in the equilibrium condition each arriving ACK triggers a transmission of a new segment. Normally, TCP does not acknowledge a received segment immediately, but waits for a certain time. If a data segment is sent during this time, the acknowledgment is “piggy backed” into it. Alternatively, another data segment can arrive, and the acknowledgment can confirm both received segments at once. However, TCP must not delay acknowledgments for more than half a second and should send an acknowledgment for every second received segment [APS99].

Data are not always delivered to TCP in a continuous way; the network can lose, duplicate or re-order packets. Arrived bytes that do not begin at the number of the next unacknowledged byte are called *out-of-order* data. As a response to out-of-order segments, TCP sends *duplicate acknowledgments* (DUPACK) that

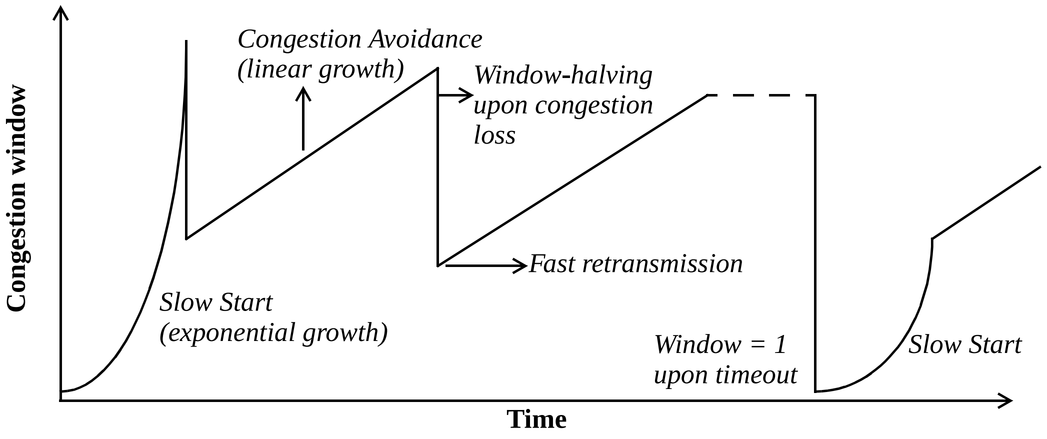

Congestion window
Congestion Avoidance (linear growth)
Window-halving upon congestion loss
Fast retransmission
Slow Start (exponential growth)
Window = 1 upon timeout
Slow Start
Time

Figure 1: Congestion control in TCP [Bal98].

curry the same acknowledgment number as the previous ACK. In combination with a retransmission timeout (RTO) on the sender side, ACKs provide reliable data delivery [Bra89]. The retransmission timer is set up based on the smoothed *round trip time* (RTT) and its variation. RTO is *backed off* exponentially at each unsuccessful retransmit of the segment [PA00]. When RTO expires, data transmission is controlled by the slow start algorithm described below.

To prevent a fast sender from overflowing a slow receiver, TCP implements the flow control based on a *sliding window* [Tan96]. In every acknowledgment, the receiver advertises to the sender the *receiver window*, the number of bytes allowed for transmission. The receiver window is always relative to the acknowledgment number. An arriving ACK allows more data to be sent by advancing the edge of the sliding window to the right. When the total size of outstanding segments, *segments in flight* (FlightSize), reaches the receiver window, the transmission of data is blocked until the sliding window advances or a larger receiver window is advertised. Advertising a window of zero bytes is legal and can be used to force the sender into the *persist mode*. In the persist mode the TCP connection is alive, but no new data can be sent until a non-zero receiver window is advertised.

Early in its evolution, TCP was enhanced by *congestion control* mechanisms to protect the network against the incoming traffic that exceeds its capacity [Jac88]. A TCP connection starts by sending out the *initial window* number of segments. The proposed congestion control standard allows the initial window of one or two segments [APS99]. During the *slow start* phase, the transmission rate is increased exponentially. The purpose of the slow start algorithm is to get the “ACK clock” running and to determine the available capacity in the network. A *congestion window* (cwnd) is a current estimation of the available capacity in the network. At any point of time, the sender is allowed to have no more segments outstanding than the minimum of the advertised and congestion window. Upon reception of an acknowledgment, the congestion window is increased by one segment, thus the sender is allowed to transmit the number of acknowledged segments plus one. This roughly doubles the congestion window per RTT (depending on whether delayed acknowledgments are in use) . The slow start ends when a segment loss is detected or when the congestion window reaches the *slow-start threshold* (ssthresh). When the slow start threshold is exceeded, the sender is in the *congestion avoidance* phase and increases the congestion window roughly by one segment per RTT. When a segment loss is detected, it is taken as a sign of congestion and the load on the network is decreased. The slow start threshold is set to the half of the current FlightSize. After a retransmission timeout, the congestion window is set to one segment and the sender proceeds with the slow start. Figure 1 shows a possible behavior of the congestion window for a TCP connection.

TCP recovery was enhanced by the *fast retransmit* and *fast recovery* algorithms to avoid waiting for a retransmit timeout every time a segment is lost [APS99]. Recall that DUPACKs are sent as a response to out-of-order segments. Because the network may re-order or duplicate packets, reception of a single DUPACK is not sufficient to conclude a segment loss. A threshold of three DUPACKs was chosen as a compromise between the danger of a spurious loss detection and a timely loss recovery. Upon the reception of three DUPACKs, the fast retransmit algorithm is triggered. The DUPACKed segment is considered lost and is retransmitted. At the same time congestion control measures are taken; the congestion window is halved. The fast recovery algorithm controls the transmission of new data until a non-duplicate ACK is received. The fast recovery algorithm treats each additional arriving DUPACK as an indication that a segment has left the network. This allows to inflate the congestion window temporarily by one MSS per each DUPACK. When the congestion window is inflated enough, each arriving

DUPACK triggers a transmission of a new segment, thus the ACK clock is preserved. When a non-duplicate ACK arrives, the fast recovery is completed and the congestion window is deflated.

New Reno [FH99] is a small but important modification to the TCP fast recovery algorithm. “Normal” fast recovery suffers from timeouts when multiple packets are lost from the same flight of segments [FF96]. New Reno can recover from multiple losses at the rate of one packet per round trip time. If during the fast recovery the first non-duplicate ACK does not acknowledge all outstanding data prior to the fast retransmit, such an ACK is called a *partial acknowledgment*. The New Reno algorithm is based on an observation that a partial acknowledgment is a strong indication that another segment was also lost. During the *recovery phase* New Reno retransmits the presumably missing segment and transmits new data if the congestion window allows it (the exact rule is given in Appendix A.1.1). The recovery phase ends when all segments outstanding before the fast retransmit are acknowledged or the retransmission timer expires.

2.1.2 Detection and Recovery of Corruption Losses

Here we describe possible events following a packet corruption on a wireless link. Normally, corrupted frames are detected and discarded by the link layer. However, some corrupted packet may be left undetected and delivered to the serial protocol running over the link. Two protocols are commonly used as a link-layer service for IP, the Point-to-Point Protocol (PPP) [Sim93] and the Serial Line Interface Protocol (SLIP) [Rom88]. PPP provides checksumming of the payload and is able to detect most corrupted frames. The predecessor of PPP, the SLIP protocol does not have error detection.

If a corrupted packet is delivered to the IP layer, the events following depend on which part of the packet was corrupted. A checksum used by IP protects only the header but not the payload of a datagram. Routers in the Internet are required to check only IP checksum, but not a checksum of the payload of an IP datagram. Hence, packets with corrupted IP headers are discarded at the first router.

If the TCP header or payload is corrupted, the packet is transmitted all the way through the Internet to the destination. Apparently, the transmission of corrupted packets through the Internet wastes resources. It is the task of the link protocols to detect and discard corrupted packets. A checksum used by TCP covers the TCP header, the payload and the pseudo-header composed of IP source and destination addresses and the length of TCP segment. The TCP checksum detects most of the corrupted packets, but still there is some chance that corrupted data can be delivered to applications [Pax97b]. TCP takes no actions upon a packet with an invalid checksum. Such packets are silently discarded. An example of the situation when a corrupted packet is undetected by PPP and is delivered to TCP can be found e.g. in [Lud00, p. 54].

Some link protocols do at least a limited number of attempts to recover a corrupted packet locally on the link. Neither PPP nor SLIP provide recovery from frame losses. IP does not have error recovery. Since TCP silently discards corrupted packets, the recovery procedure is the same whether a corrupted packet is delivered to TCP or not. Three DUPACKs or a retransmission timeout is used to detect a packet loss. Upon a detection, the packet is retransmitted and congestion control measures are taken.

2.1.3 Selecting the TCP Implementation

The TCP behavior is standardized by IETF and is described in RFCs. However, the standards leave many issues unspecified and TCP implementations differ in how they behave under similar conditions. For a long time, the reference implementation has been Reno TCP found in the Unix BSD4.3 operating system [WS95]. Modern TCP implementations differ significantly from Reno. The current family of BSD OSes is derived from Unix BSD4.4 with TCP-Lite implementation [Lud00].

For the baseline in our analysis we wanted to select a state-of-art TCP implementation that is both widely used in the Internet and has the source code available for analysis and modification. We chose Linux as a popular operating system with the source code available. Due to a large amount of independent developers interested in Linux, implementations of new features are quick to appear for Linux.

The TCP implementation in earlier versions of Linux had problems with conforming to standards [Pax97a]. We have detected, evaluated and corrected a number of misbehavior problems. We believe that after these fixes we obtained a TCP that behaves reasonably with regard to standards. A recent work gives the requirements for a TCP implementation to be used for TCP research [AF99]. Our baseline TCP (described in detail is Appendix A) satisfies these requirements. One useful option would be to run a part of the tests also with a current version of the BSD Unix and compare with results obtained with our TCP.

2.2 Network Environment

2.2.1 Properties of Wireless Links

A wide range of wireless technologies that exist today differ a great dial in their properties. Wireless in its original meaning refers to communication without wires, which could based on the radio medium, the infrared light or other means. In this thesis, we would use wireless to refer to radio waves. Furthermore, wireless networks that exist today differ considerably in their transmission rate and delay properties. Although Wireless LANs, satellite links and Wireless Wide Area Networks (WWAN) certainly share some common characteristics, they also have enough distinct properties to be taken as different environments for data communication. In this thesis we are focusing on WWAN, that is, cellular phone systems also capable of data transmission. Hence we furthermore limit the wireless term to refer to WWANs in our contents.

Many wireless links are slow, have high latency and may have high error rates. These link characteristics adversely affect the TCP performance. The line rate of a wireless link may not exceed some tens of kilobits per second. Such a link speed is typical also for dial-up modem users. For some wireless links, the line rate can vary over time, due to a change in the amount of radio resources assigned to the user. We do not consider links with changing bandwidth in this thesis, although such links may prove to be an interesting environment and worth studying in the future.

The *latency*, the propagation delay, of wireless links is typically high. The latency comes from the special transmission schemas and processing delays the network equipment. For example, the Global System for Mobile Communications (GSM) uses interleaving of data on the radio link to reduce the effect of error bursts, and this introduces a latency of 90 ms independent of packet size [Lud00]. Additional latency in using a GSM data service is caused by the modem link to the Internet Service Provider (ISP) and processing time within the GSM system. The total one-way latency in GSM sums up to 200-300 ms. Note, that we do not include the transmission delay into the link latency. Thus the *round-trip time* is defined as the sum of transmission and propagation delays in both directions.

Some wireless links impose a significant amount of data corruption due to transmission errors. The error rate depends on the current radio conditions and the strength of the channel coding schema. For example, in the transparent GSM data service the residual bit error rate (BER) of the link can be as high as 10^{-3} after the Forward Error Correction (FEC) [MP92]. Radio conditions can vary greatly. In the ideal conditions all packets are delivered correctly, and in the worst case nothing can be correctly sent over the link. Some links employ the Automatic Link Adaptation (ALA) to change the channel encoding strength in response to the change of the radio quality [MP92].

The delay-bandwidth product is an important characteristic of a network [Sta00]. It defines the minimum size of data in flight to utilize the available network bandwidth, the pipe capacity. Networks with a large delay-bandwidth product, for example including satellite links, demand special attention from the transport protocol. For example, the slow start phase of TCP can be time-consuming in such networks [ADG$^+$00]. In our environment, the delay-bandwidth product is small, close to one kilobyte. In the slow start, the pipe capacity is filled already after one-two RTTs .

2.2.2 Network Architecture

Rather than selecting one particular network architecture and developing a detailed model that would reflect the behavior of this network we attempt to build a generic model that would be suitable for all wireless networks with similar characteristics. We are interested in the issue how a nomadic user can use Internet services via a wireless network. In a scenario shown in Figure 2, the wireless network plays the role of an access network from the Internet point of view. It is also possible for a nomadic user to exchange data with another mobile user, so that two wireless links are present on the data path. We do not consider such configuration is this thesis, assuming that the access to a remote host in the Internet would be the dominant case.

The wireless link is often the bottleneck in the path of a data flow, because fixed networks are fast and reliable compared to the capabilities of the wireless link. When data packets flow from the relatively fast Internet to the slow wireless link they are buffered in the last-hop router which connects the wireless link to the Internet. This router plays a significant role in the end-to-end TCP performance because congestion data losses are most likely to happen at the bottleneck queue. A limited number of buffers can be allocated in the last-hop router per user. This buffer space is shared among connections of the same user, but there is no interference between the connections of different users. A similar network architecture was considered, when three buffers are available per user [SP98].

The wireless link in our environment imposes corruption losses. We assume that all data with transmission errors are detected and discarded at the wireless link. We also assume no error recovery and no variable delays on the link. We do not include the Internet into our environment in the rest of the thesis. Thus, different patterns of link errors is the only non-deterministic element in our environment.

We now discuss how existing wireless networks can be mapped to our generic model. The Global System for Mobile Communications (GSM) is a widely successful effort to build a WWAN system with millions of users in Europe and worldwide [MP92, Rah93]. GSM data, High Speed Circuit Switch Data (HSCSD),

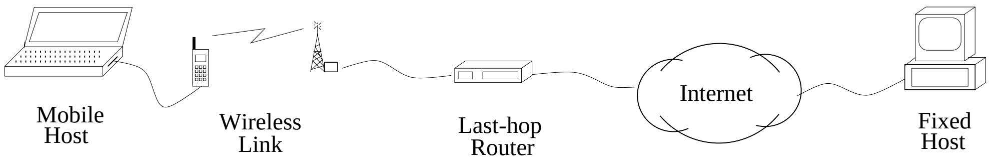


Figure 2: Network Architecture.

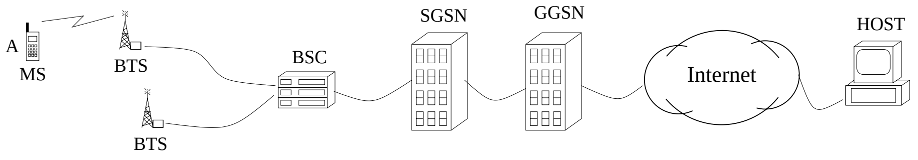


Figure 3: GPRS Data Transmission Path.

and General Packet Radio Service (GPRS) [BW97] are data transmission services offered by GSM. We believe they will be the dominant wireless data transmission services in the foreseeable future. GPRS is expected to provide a high speed packet data access suitable for a wide range of Internet services. The GPRS network is a complex system that consists of multiple nodes. However, for a fixed user in the Internet it is visible on the IP layer as a usual Internet subnetwork.

The GPRS data transmission is depicted in Figure 3. It maps well to our generic model shown in Figure 2. The Gateway GPRS Support Node (GGSN) acts like a router connecting the Mobile Station (MS) user to the Internet. The bottleneck queue is located in the Service GPRS Support Node (SGSN). Although, the link layer in GPRS will normally be operating in the reliable mode with a very low BER, an unreliable mode of operation is specified too. Transparent data delivery over a GPRS network with no link-level error correction might still be used as an inexpensive option [GSM98].

2.3 Related Work

Improving TCP performance over connections including a wireless link has been an active research area for a few years. An earlier attempt to classify existing solutions outlines three different categories: end-to-end proposals, split TCP and link-layer proposals [BSAK95].

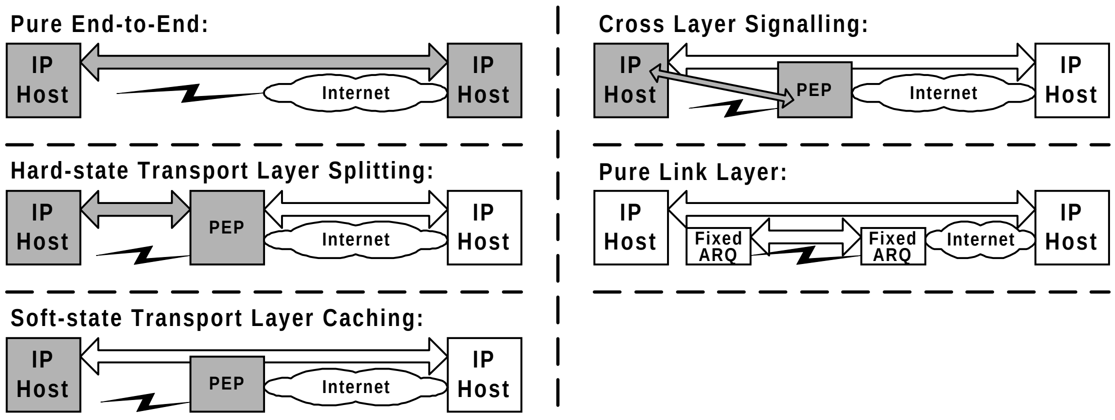


Figure 4: Approaches to improve TCP over wireless [Lud99].

A more recent work gives an excellent classification of approaches to improve the performance of TCP at a high level of corruption losses [Vai99]. In the most general approach, all methods either attempt to hide error losses from the sender or alternatively make the sender know the cause of a packet loss. The first group corresponds to the ideal network behavior, where errors are recovered transparently and without performance degradation visible for the user. The second group corresponds to the ideal TCP behavior, where TCP simply retransmits corrupted packets without taking any congestion avoidance measures. The ideal network or TCP behavior cannot be achieved, but methods attempt to approximate either of two. The next aspect is which part of the system needs to be modified to achieve the performance improvement. Changes can be made to the sender, receiver or to an intermediate node. A common agreement is that the legacy servers in the Internet cannot be modified, or it takes a long time until a change could become widely employed. An intermediate node can in some cases be modified. The implementation of the network stack in the mobile host can often be controlled and we are able to apply any changes there. Such changes, however, must be backward compatible not to harm interoperability. On the functional description level, methods are divided into the following groups: link-level mechanisms, split connection approach, TCP-aware link layer, TCP-unaware approximation of TCP-aware link layer, explicit notification, receiver-based discrimination, and sender-based discrimination.

Another recent work gives an excellent state-of-art classification that we would like to discuss here in detail [Lud99]. Five different categories of solution are illustrated in Figure 4. The pure transport layer solutions are based on modifications of TCP solely at the end points of a connection. This scheme retains the end-to-end TCP connection semantics, but enhances the TCP protocol to make it perform better in the wireless environment. We are working in this area. It is important, though, not to break the TCP standard mechanisms, such as the slow start and congestion avoidance, and tolerance to re-ordered packets. These mechanism are crucial to the stability of the Internet.

The "transport layer splitting" solutions argue that the properties of the wired and wireless links are so different, that they are best handled separately. The TCP connection from the fixed host is terminated at an intermediate node, a Performance Enhancing Proxy (PEP), and a special protocol is used for data delivery over the wireless link [BKG$^+$00]. The major advantage of PEPs is that areas of congestion and corruption losses are handled separately in an appropriate way. However, PEP violates the end-to-end semantics of the TCP protocol, because "faked" acknowledgments are sent to the fixed IP host before data are actually delivered to the destination. The proxy is said to maintain a hard state, since any data lost beyond it are not recovered by TCP.

The "transport layer caching" approach eliminates the problem of maintaining the hard state in the proxy. The loss of the soft state on the proxy can affect the performance, but does not prevent the end-to-end data delivery by TCP. The best-known implementation of the soft-state proxy concept is the Snoop protocol [BSK95]. Snoop examines packets in PEP in a way that allows to detect TCP segment losses and recover locally by retransmitting the cached segment. The main shortcoming of Snoop is low performance in the presence of a high level of congestion losses [Lud00].

Solutions based on soft-state cross-layer signalling inform the transport layer of specific events on the link layer. This category of solutions includes, for example, an explicit "bad-state" notification [BKPV96] , and an explicit loss notification [BPSK96]. Such methods are often difficult to implement because they require modifications both to PEP and to the TCP protocol. In addition, such methods do not typically work in the presence of the encryption provided by

IPsec [KA98].

Pure link layer solutions struggle to isolate the local problems of the wireless link from the higher layers. Many wireless links can recover from lost packets by using link-level retransmissions using the Automatic Repeat Request (ARQ) [Sta00]. Link errors are not visible to the upper layers, at the expense of variable delays in the data delivery. Some link-layer protocols provide *semi-reliable* data delivery, by performing only a small number of local retransmissions before discarding a packet. The current research favors highly persistent link-layer recovery [Lud00].

For certain types of traffic, for example real-time video, link layer recovery may be harmful since data must be delivered timely or not at all. The work [Lud99] introduces the concept of a *flow-adaptive link* which is capable to satisfy the Quality of Service (QoS) requirements of a data packet by changing, for example, the link retransmission policy. The QoS requirements of a packet are given to the link layer in the type-of-service octet in the IP header.

The standardization body for the Internet protocols, the Internet Engineering Task Force (IETF), is specifying various performance enhancements to TCP and is documenting the impact of problematic link-layer characteristics to the Internet protocols. State-of-art understanding of the issue is found in the recent Long Thin Networks (LTN) RFC [MDK$^+$00], and two Internet Drafts, End-to-end Performance Implications of Slow Links [DMKM00] and Links with Errors [DMK$^+$00]. Our goal is to provide experimental data on how well these enhancements actually perform in the presence of congestion and corruption losses.

In our environment, a natural issue is the achievable improvement when the sender is able to distinguish between congestion and corruption losses. In other words, for each packet loss, the TCP sender knows if the loss occurred due to congestion or due to corruption loss over the wireless link. A study [BV97] shows that the improvement depends on the ratio of congestion and corruption loss probabilities. The result is obtained from experiments and theoretical approximations as follows. A simple approximation for long range throughput T of TCP-Reno is from [MSMO97]:

$$T = \frac{MSS}{RTT} * \frac{C}{\sqrt{r}}$$

where MSS is the Maximum Segment Size, RTT is the Round Trip Time, C is a constant, r is the random loss rate.

The approximation omits other details of the recovery process, except the fact that TCP halves the congestion window at every packet loss. The most important omission is the effect of RTO on recovery. If a TCP connection experiences congestion losses with a rate r_c and corruption losses with a rate r_w, the approximation of its long range throughput is

$$T = \frac{MSS}{RTT} * \frac{C}{\sqrt{r_c + r_w}}.$$

Next, imaginary *Ideal TCP-Reno* is introduced that has perfect knowledge of the reason for a packet loss, and thus halves its congestion window only for congestion losses. The approximation of long range throughput for Ideal Reno is

$$T = \frac{MSS}{RTT} * \frac{C}{\sqrt{r_c}}.$$

The authors do not give a valid range of parameters for the estimation [BV97]. We believe that the approximation is only valid when r_w and r_c are of a few per cent.

The improvement of *Ideal TCP-Reno* over TCP is approximated as $T_{Ideal}/T_{TCP} = \sqrt{1 + r_w/r_c}$. We see that r_w/r_c is the main factor of how much better *Ideal TCP-Reno* can perform. The secondary factor affecting the *Ideal TCP-Reno* performance advantage is the bandwidth-delay product. When it is small, the congestion window stays small at the presence of error losses. In this case recovery using a retransmission timeout rather than using a fast retransmit is more likely when an error loss occurs. In such conditions, the performance improvement achievable by *Ideal TCP-Reno* is low.

A TCP implementation that achieves to some extent the performance of *Ideal TCP-Reno* can be based either on discrimination heuristics or explicit loss notifications. Attempts to use simple statistics on the round-trip time and throughput were not successful [BV97]. Proposals based on explicit loss notification are more promising and include Explicit Loss Notification to the Receiver (ELNR) [MV97], Explicit Loss Notification (ELN) and Explicit Bad State Notification (EBSN). However, all these proposals fall into the cross-layer signalling category and are difficult to deploy. In our work we do not consider TCP optimizations based on distinguishing between congestion and corruption losses.

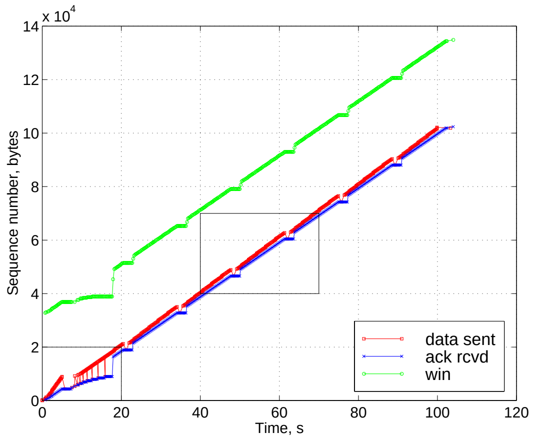
3 Problem Description

In this section we outline the specific problems of TCP over wireless links we focus on in the rest of the thesis.

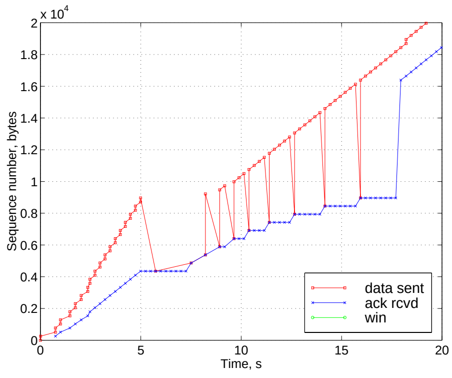
3.1 Congestion Losses

In this section we discuss the occurrence of congestion losses and their effect on TCP. We use the term *congestion* for the time period when many packet losses occur due to a buffer overflow, even in the case of a single connection. We first look at a typical TCP connection over a limited-size buffer, but in an error-free environment. Figure 5 shows a baseline TCP connection when the buffer space is limited to seven packets. Two phases of the connection are clearly visible. In the first phase, which lasts approximately 20 s, the connection starts up aggressively, creates congestion, loses a large number of packets and recovers them. We call this phase *the start-up buffer overflow* hereinafter. In the second phase, the connection proceeds smoothly with the periodic loss of a packet. This is referred to as the *steady state* of the connection. During this phase the connection goes through periodic *congestion avoidance cycles* following the linear increase – multiplicative decrease policy [Ste97]. In the beginning of the cycle, the FlightSize is increased by one MSS per RTT until the FlightSize reaches the size of the router buffer. When a single packet is lost due to the router buffer overflow and the loss is detected by the TCP sender, the FlightSize is halved and the cycle starts over.

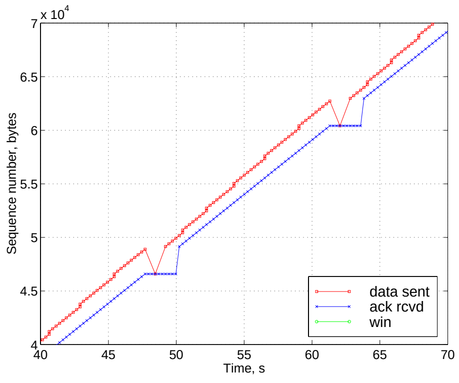
Start-up buffer overflow. Let us look at the start-up buffer overflow which is also known as the slow-start overshoot [MM96]. Figure 5(b) zooms on the start-up buffer overflow. Ten segments are lost and retransmitted. The important points to notice on the figure are: when congestion occurs, when the first packet loss is detected, and how segment losses are recovered. Questions about the start-up buffer overflow are “why does it happen”, “what is the negative effect”, and “how can it be prevented”. We provide the detailed analysis in Section 7.6



(a) Complete connection



(b) Start-up zoom



(c) Steady-state zoom

Figure 5: An error-free TCP connection over the router buffer of seven packets.

The optimal router buffer size. The maximum size of the queue in the router has a significant effect on the connection. A router buffer, which is too small, can result in a smaller FlightSize than needed by TCP to recover well from packet losses. The size, which is too large, leads to the heavy start-up buffer overflow and overbuffering. One paper has estimated 1.5*RTT*bandwidth as the optimal value for the buffer size [Lud00].

Overbuffering. The situation when significantly more packets are in flight than is required to fill the available network capacity is called *overbuffering*. Overbuffering does not necessarily cause congestion. If the number of packets injected into the network equals the number of packets leaving the network, no congestion take place. However, having a large number of packets buffered in the network has several drawbacks [Lud00]. If buffers in the network are full, there is no capacity left to accommodate traffic bursts. Some applications using TCP generate bursty traffic. In addition, the TCP protocol itself can inject packets in bursts. Another drawback is a poor service for interactive applications, because the end-to-end delay on the overbuffered path can be huge. Finally, the data in the network can become stale, when a user aborts the data transfer, for example using a stop button in a web browser. Due to these reasons overbuffering should be avoided.

Fair sharing of resources. Tail-drop routers are known to have problems with sharing the bandwidth between connections in a fair way [BCC^{+}98]. When two or more TCP connections share the same router buffer, one connection can starve while other connections monopolize the resources. This situation is referred to as *lock-out* and occurs due to timing effects. We would like to avoid this problem in our environment.

3.2 Corruption Losses

Performance problems of TCP at the presence of error losses are well known [BSAK95]. Upon a loss detection, TCP always reduces the transmission rate, as the reason of the packet loss, congestion or corruption, is not known.

When the level of error losses is low, they do not have a notable effect on the performance. At the moderate level of error losses, TCP underestimated the available network bandwidth. When the level of error losses is high, most of time the connection is idle waiting for a retransmission timeout to expire. In the worst case, the connection is terminated, when the maximum number of retransmission is exceeded. Not only the rate of the error losses is important, but also the burstiness [Lud00]. In general, TCP suffers more when errors are bursty rather when they are uniformly distributed. Recommendations for using the TCP algorithms and control parameters at the presence of error losses is given in [DMK$^+$00]. We have identified three patterns of error losses to be studied.

Single Errors. Normally, single-packet error drops do not have a significant effect on the TCP behavior, except for a few special cases. We will try to locate such interesting cases and analyze them.

Random errors. We will try to identify levels of the uniformly-distributed packet error rate when error losses have no effect on performance, when the link bandwidth is underestimated, when most time is spent in RTOs and when the connection is terminated. We will study how different TCP optimizations affect the performance for varying size of the router buffer and the error loss rate.

Burst errors. It is interesting to study the effect of burst errors on TCP. An error loss rate of one percent does not normally affect TCP performance, if uniformly distributed. However, the same error rate when errors occur in clusters can adversely impact performance. We expect TCP to perform badly during an error burst; also performance after the burst ends can be hampered.

3.3 OS-Related Problems

State-of-art TCP performance. Most of the related TCP research concentrated on evaluation of TCP performance of the Reno TCP implementation or an abstract TCP model in the *ns* simulator [PN98]. However, from the point of view of an end user, it is much more important how their currently installed operating system performs under the given conditions. The upcoming Linux version 2.4 differs from Reno in many ways. Thus, it is actual to evaluate how a state-of-art TCP implementation (our baseline TCP is defined in Appendix A) performs on wireless links.

Conformance of Linux. There is a stereotype among researchers about the TCP implementation in Linux, that it does not conform to standards. Indeed, earlier releases of the Linux kernel showed malicious behavior and were even named as an incoming danger to the Internet [Pax97a]. The Linux networking code has undergone significant changes since version 1.0, and a large number of independent developers have verified and improved Linux. Today, when Linux is widely used on Internet servers, it is actual to locate and fix the remaining inconsistencies with TCP standards produced by IETF.

3.4 Summary

We consider a network model including a lossy wireless link and a last-hop router with a fixed-size buffer. The Internet is not included in the study. We assume no variable delays on the link. We examine the end-to-end TCP performance using the state-of-art TCP implementation in Linux. The main problems we address are the start-up buffer overflow, overbuffering, optimal buffer size, fair sharing of resources, the effect of single, random and burst errors. A number of optimizations will be tested to study the effect on TCP performance in our environment.

4 Optimizations

In this section we discuss optimizations that possibly improve the TCP performance in our environment. First, appropriate values of the standard TCP control parameters are considered. Second, we describe two TCP extensions that optimize the protocol operation. Third, the active queue management in the router buffer is described. Finally, we list the factors that are relevant for our work, but are left for the future study.

4.1 TCP Control Parameters

4.1.1 Initial Window

The TCP protocol starts transmitting data in the connection by injecting the initial window number of segments into the network. The initial window of one or two segments is allowed by the current congestion control standard [APS99]. An experimental extension allows an increase of the initial window to three or four segments [AFP98]. However, the number of segments sent after RTO, the loss window, is fixed at one segment and remains unchanged.

The increased initial window size has the advantage of saving up to three RTTs from the connection time. It also decreases the time when the FlightSize of the connection is smaller than necessary to trigger the fast retransmit if a packet loss occurs. This decreases the probability of the connection experiencing RTOs. The increased initial window may have a possible disadvantage for an individual connection in an increased probability of a congestion loss in the connection start-up when the router buffer size is small. A study has been made to evaluate a connection with the initial window of four segments when the router buffer size is three packets [SP98]. The study shows that the four-packet start is no worse than what happens after two RTTs in the normal slow start with the initial window of two segments. Another simulation study has evaluated the effect of the increased initial window on the network [PN98]. The study concludes that the increased initial window size does not significantly increase congestion losses but improves the response time for short-living connections.

Using an increased initial window can be beneficial in our environment because of the high RTT of the wireless link and presence of error losses. We expect that the performance increases with increasing the initial window, but the improvement only affects the beginning of connections. In addition, interesting questions are, whether the number of RTOs is reduced and whether the start-up buffer overflow is worsened by the increased initial window.

4.1.2 Receiver Window

The amount of outstanding data, the FlightSize, is limited at any time of a connection by the minimum of the congestion window and the receiver's advertised window. The size of the receiver window is a standard control parameter of TCP [Pos81]. By advertising a smaller window the receiver can control the number of segments that the sender is allowed to transmit. The basic analysis of the effect of the receiver window on a protocol performance can be found e.g. in [Sta00].

If the receiver window is limited to an appropriate value that reflects the available network capacity, then congestion losses are prevented. The receiver rarely has any knowledge of the underlying network properties and current state. However, when a host knows that it is connected to a last-hop wireless link, it could limit the advertised window [DMKM00]. Limiting the receiver window also prevents excessive queueing in the network (overbuffering). Overbuffering occurs when the size of the router buffer is much larger than required to utilize the link.

It is interesting to examine whenever the limited receiver window prevents the start-up buffer overflow, whether error recovery is disturbed and what the appropriate size of the receiver window is for a given size of the router buffer. We expect that when the receiver window is limited to an appropriate value, TCP performance is improved, but the improvement only affects the beginning of connections and is more visible for a larger router buffer. When the receiver window is larger than appropriate, we expect TCP to perform similar to the baseline. The receiver window which is too small can adversely affect TCP performance.

4.1.3 Maximum Segment Size

The Maximum Segment Size affects TCP performance [Ste95, MDK$^+$00]. The Maximum Transfer Unit (MTU) of the network path imposes an upper limit for MSS; in certain cases using a smaller MSS is desirable. For example, with an MSS of 1024 bytes, each segment will occupy a 9600-bps link for almost a second. This is unacceptable for an interactive application, because a large file transfer packet can delay a small telnet packet for a time much longer than the human-perceptible delay. Links that rely on the end-to-end TCP error recovery also demand a small MSS. For a fixed BER, the probability of segment corruption increases with its size. On the other hand, the header overhead grows with a smaller MSS, especially in the absence of the TCP/IP header compression. A MSS value of 256 bytes for a 9600-bps link is often used as a compromise.

It is interesting to examine the effect of a larger MSS on the TCP congestion and error control. TCP grows the congestion window in units of segments, independently of the number of bytes acknowledged. Using a larger MSS allows a connection to complete the slow start phase faster. On the other hand, connections with a larger MSS may suffer more from RTOs. A larger segment size causes a larger RTT and thus a number of packets in flight grows slower than for a smaller MSS. There is smaller probability to have enough DUPACKs to trigger a fast retransmit. Furthermore, a larger RTO, especially after a back off, increases the recovery time.

4.1.4 Disabling Delayed Acknowledgments

Delayed acknowledgments is an important feature of TCP that can affect the performance of a connection. The basic information about delayed acknowledgments was given on page 3, a more detailed description can be found for example in [Ste95]. A common value for the delay of 200 ms is used by Linux. Furthermore, Linux TCP detects the situation when packets arrive less frequently than the delayed acknowledgment timeout and sends acknowledgments immediately upon reception of a segment, i.e. without waiting for 200 ms. When acknowledgments are delayed on a bulk data transfer, every second segment is normally ACKed. An

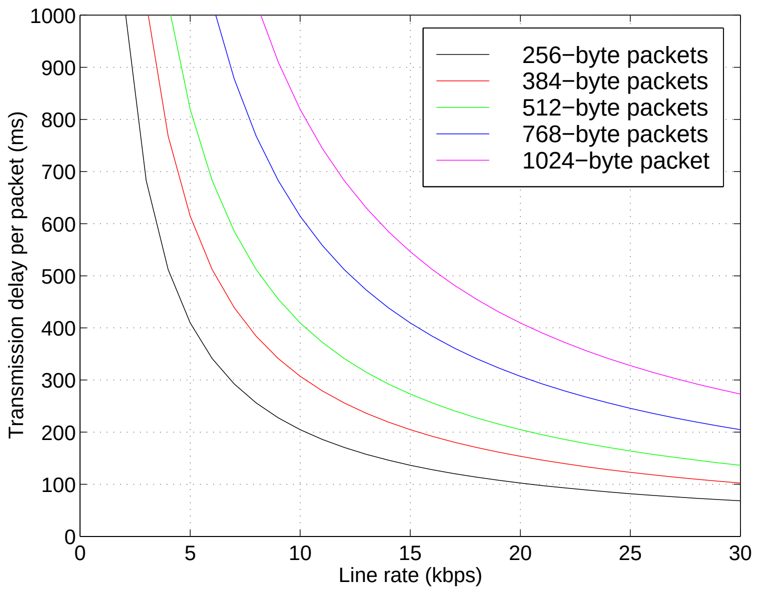


Figure 6: Length of the transmission delay for different packet sizes and line rates.

arriving ACK advances the sliding window and increases the congestion window; thus, a connection with delayed ACKs is less aggressive. This is especially visible in the slow start phase, because in the slow start each arriving ACK increases the congestion window by one segment.

We need to consider the implication of delayed acknowledgments on our tests. The transmission delay of the link corresponds to the interval at which packets arrive to the receiver. If the transmission delay is larger than the timeout for delaying acknowledgments, every packet is acknowledged. The value of the transmission delay depends on the line rate and MSS used on the connection. Figure 6 shows the transmission delay for packet sizes and line rates of interest to us. In our environment (MSS of 256 bytes and the line rate of 9600 bps) the transmission delay is longer than the timeout for delayed ACKs. Thus, each segment is acknowledged.

Linux introduces a new feature called “quick ACKs”. The idea is to disable delayed ACKs for the first n packets of the connection, where n is a configurable parameter. Acknowledging every packet at the beginning of the connection allows achieving the equilibrium state (congestion avoidance) in shorter time. On the other hand, such a policy could increase the probability of network congestion, as the sender transmits data more aggressively. Quick ACKs do not affect our tests, because at the 9600 bps bandwidth, every segment is ACKed anyway as explained above.

4.2 TCP Optimizations

4.2.1 Selective Acknowledgments

TCP acknowledgments are cumulative; an ACK confirms reception of all data up to a given byte, but provides no information whether any bytes beyond this number were received. The Selective Acknowledgment (SACK) option [MMFR96] in TCP is a way to inform the sender which bytes have been received correctly and which bytes are missing and thus need a retransmission. How the sender uses the information provided by SACK is implementation-dependent. For example, Linux uses a Forward Acknowledgment (FACK) algorithm [MM96]. Another implementation is sometimes referred to as “Reno+SACK” [MMFR96, MM96]. SACK does not change the semantics of the cumulative acknowledgment. Only after a cumulative ACK, data are “really” confirmed and can be discarded from the send buffer. The receiver is allowed to discard SACKed, but not ACKed, data at any time.

The FACK algorithm uses the additional information provided by the SACK option to keep an explicit measure of the total number of bytes of data outstanding in the network [MM96]. In contrast, Reno and Reno+SACK both attempt to estimate the number of segments in the network by assuming that each duplicate ACK received represents one segment which has left the network. In other words, FACK assumes that segments in the “holes” of the SACK list, are lost and thus left the network. This allows FACK to be more aggressive than Reno+SACK in recovery of data losses. In particular, the fast retransmit can be triggered already after a single DUPACK in FACK implementation if the SACK information in the DUPACK indicated that several segments were lost. In contrast, Reno+SACK will wait for three DUPACKs to trigger the fast retransmit.

A loss of multiple segments from a FlightSize of data often presents a problem for TCP [FH99]. As one option, the sender either have to retransmit outstanding segments using the slow start; most of the segments could be received correctly already and thus are unnecessarily retransmitted. As another option, the sender can recover by one segment per RTT as the cumulative acknowledgment number advances. In the presence of SACK, the sender knows exactly which segments were lost and thus can recover multiple segments per RTT without unnecessary retransmits. SACK TCP has been shown to perform well even at a high level of packet losses in the network [MM96].

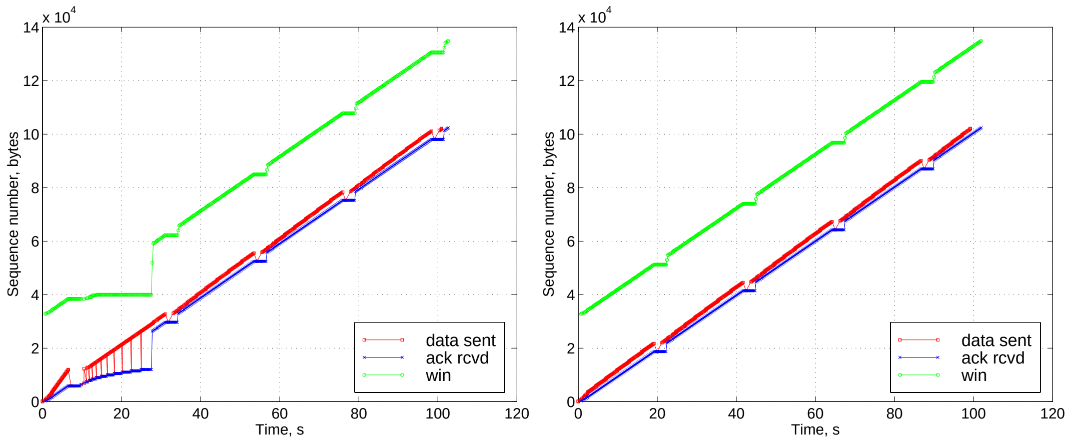
We expect SACK TCP to perform better than the baseline and other optimizations under all conditions. The difference will be most significant at a high level of error losses. It is interesting to examine whether SACK recovers well from the start-up buffer overflow.

4.2.2 Control Block Interdependence

A control block of a TCP connection maintains the connection state, round-trip time estimation, slow start threshold, maximum segment size, and other similar parameters. When a new connection is created, it has no idea what the properties of the underlying network path are, and it has to determine values of these parameters empirically. The performance of this new connection could be improved if it takes advantage of parameters obtained by earlier connections. TCP Control Block Interdependence (CBI) [Tou97] is the way to share the information between connections.

Figure 7 shows two subsequent connections to the same host in the presence of CBI. The second connection avoids the start-up buffer overflow, because the congestion control variables were initialized with values obtained by the first connection. To be exact, the slow start threshold (ssthresh) is set to an appropriate value so that TCP switches from the slow start to the congestion avoidance before the router buffer overflows.

To collect reasonable statistics we need to rerun the same test multiple times. Enabling CBI would make connections dependent on each other and disturb the



(a) The first connection (b) The second connection

Figure 7: Effect of CBI on TCP connections. Both connections are to the same host. The second is started after the first has been completed.

results. Also the effect of other optimizations cannot be easily observed in the presence of CBI. For these reasons, we had to disable CBI for our tests. However, we believe that CBI is a useful feature that improves TCP performance and should be widely used.

4.3 Active Queue Management

A method that allows routers to decide when and how many packets to drop is called the *active queue management*. The Random Early Detection (RED) algorithm is the most popular active queue management algorithm nowadays [FJ93]. A RED router detects incipient congestion by observing the moving average of the queue size. To notify connections about upcoming congestion, the router selectively drops packets. TCP connections reduce their transmission rate when they detect lost packets and congestion is prevented.

The RED algorithm solves two problems related to congestion losses: overbuffering and fair sharing of resources. RED is recommended as a default queue management algorithm in the Internet routers [BCC+98]. This is motivated by the statement that all available empirical evidence shows that the deployment of RED in the Internet would have substantial performance benefits. There are seemingly no disadvantages to using the RED algorithm, and numerous advantages [FJ93].

RED may not be useful in our environment. The major advantages of RED in providing fair sharing of resources and the low-delay service for interactive applications simply are not needed in the case of a single bulk data transfer. It is probable that RED does not prevent the start-up buffer overflows. Still, we would like to evaluate the effect of RED on TCP performance in our environment, because RED can improve the performance of two concurrent bulk connections and the algorithm is expected to be widely deployed in the Internet.

Here we provide some details about the RED algorithm for an interested reader. The algorithm contains two parts. The first part is to compute the moving average of queue size avg that determines the degree of burstiness allowed in the router queue. The second part is to determine the packet-dropping probability, given the moving average of the queue size. The general RED algorithm is shown in Figure 4.3. The moving average of the queue size is computed by a low-pass filter giving the current queue size a certain weight in the result. When the moving average is below the minimum threshold min_{th} no packets are dropped, and when it is above the maximum threshold max_{th}, every arriving packet is dropped. Between these boundary conditions, each packet is marked with a probability p_a that depends on the moving average. During congestion the probability that the router drops a packet from a connection is roughly proportional to the bandwidth share of that connection. By default the RED algorithm measures the queue size in packets, not in bytes.

4.4 Other Modifications

4.4.1 Timestamps

The TCP timestamp option [BBJ92] requires the sender to place a current timestamp and echo the most recent received timestamp into each transmitted segment. The timestamp option was introduced for protection against wrapped sequence

```
for each packet arrival
   calculate the moving average of the queue size $avg$
   if $min_{th} \leq avg < max_{th}$
      calculate probability $p_a$
      with probability $p_a$:
         drop the arriving packet
   else if $max_{th} \leq avg$
      drop the arriving packet
```

Figure 8: The general algorithm of the Random Early Detection (RED).

numbers. It can also be used to improve the RTT estimate collection. With timestamps, every received segment, also retransmitted, can be used as an RTT sample. The timestamp option occupies 12 bytes in each segment.

Several algorithms for improving TCP over wireless links are dependent on the timestamps. One example is the Eifel algorithm for the prevention of spurious retransmits [LK00]. Thus, it is actual to evaluate the effect of using timestamps in our environment. A better RTT estimate may be helpful to reduce the number of RTOs. However, due to time limits, we left TCP with timestamps for the future work.

We do not expect that timestamps would improve TCP performance. The overhead caused by a timestamp in every segment is too high for a small MSS. When timestamps are used the number of segments is larger than for the baseline.

4.4.2 Header Compression

Compressing TCP and IP headers can decrease the header overhead significantly. A widely used Van Jacobson (VJ) header compression [Jac90] is a proposed standard. The VJ compression is sensible to packet losses; a single-packet loss causes the full FlightSize to be dropped that forces TCP into RTO. A more recent header compression proposal [DNP99] supports an explicit request for a retransmission of an uncompressed packet, and thus does not have this drawback. In addition, the PPP protocol defines its own type of the header compression [ECB99]. Some TCP options, for example timestamps, prevent the header compression.

For a typical packet of 296 bytes, the overhead from TCP/IP headers is reduced from 40 to 3-5 bytes, or in other words from 13 % to 1 - 1.5 %. Reducing the overhead is especially important for connections with a small MSS.

We do not use any compression method in our tests. Using the header compression is problematic on links with errors [DMK$^+$00]. Also it would make the comparison of optimizations difficult, because the header compression cannot be applied to segments with a timestamp or SACK TCP option.

4.4.3 Explicit Congestion Notification

A packet loss serves TCP as an implicit notification of congestion. The Explicit Congestion Notification (ECN) is a complementary mechanism to the active queue management [FR99, RF99]. ECN provides means to notify a TCP connection of incipient congestion as an alternative to dropping packets. ECN uses bits in the packet header to indicate that this packet has passed through a congested router. The receiver echos the congestion indicator in ACKs. Upon reception of a congestion notification the sender must react in the same way as for a single dropped packet, that is reducing the transmission rate.

ECN has obvious advantages in avoiding unnecessary dropped packets (since there is actually free queue space to store them), avoiding excessive delays due to retransmissions and wasted bandwidth on the path from the sender to the router.

We have not used ECN in our tests. In future, it will be interesting to evaluate performance benefits of ECN. It is important not to treat a lack of ECN notification for a lost packet as a signal of a corruption loss. An ECN-capable packet can well be dropped by a non-ECN aware router or even by an ECN router under heavy congestion. Performing aggressive retransmissions in such a case is a network equivalent of “pouring gasoline on a fire” [Jac88].

5 Performance Model

5.1 Network Model

In this section we describe the network model for the network architecture depicted in Figure 2 on page 11. The network model is implemented in a real-time emulator. The model of downlink and uplink channels is shown in Figure 9. The last-hop router is modeled as a queue. The wireless link is modeled as a combination of the transmission and propagation delays; error losses are modeled as packet drops. The uplink and downlink directions in our model are independent.

In the downlink direction, packets arriving to the emulator are placed in the queue. The maximum queue length can be limited; when an overflow happens, packets are tail-dropped. The RED algorithm can be used to actively control the queue length. Packets are taken from the head of the queue one-by-one for "transmission" over the link. The length of the transmission delay is computed according to the line rate and the packet size. When the transmission delay for a packet is completed, the packet is moved to the propagation delay node. The length of the propagation delay is the same for all packets independently of packet size. Several packets can be in the propagation delay node simultaneously. Error losses are modeled by dropping packets after the propagation delay. If a packet was not dropped, it is sent out from the emulator.

On the uplink direction, the transmission and propagation delay nodes are used in the same way as for downlink. We assume no queueing in the uplink direction. With our workload model described in Section 5.2 the chance of two or more packets (i.e. acknowledgments) to be queued in the uplink direction is negligible. Error losses are modeled in the same way as for downlink.

We assume the link rate of 9600 bps and the propagation delay of 200 ms hereinafter. Our error model assumes that all corrupted packets are detected and discarded on the wireless link, that is, no corrupted packets are delivered to the IP layer. The packet drop probability is independent of the packet size. This may be considered inaccurate because, for example, the loss rate of small ACKs is the same as of large packets. However, this is the case, for example, when

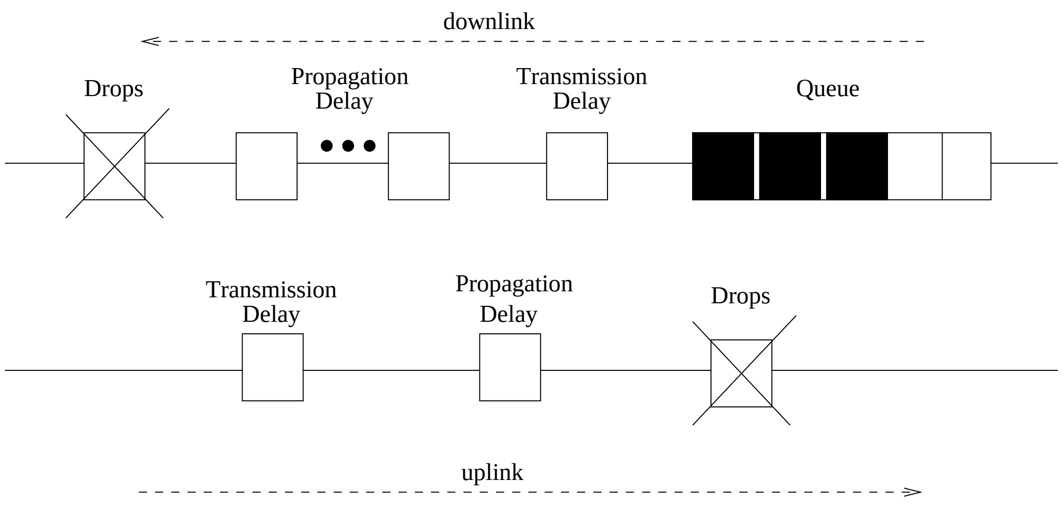


Figure 9: The model of downlink and uplink data channels.

acknowledgments are piggybacked to large data packets.

We do not attempt to include the Internet in the model, although the Internet is a part of our environment (Figure 2). Modeling of the Internet is hard because of its great heterogeneity and the rate at which its properties change [Pax97a]. Indeed, there is a tremendous number of different routes in the Internet, each one with its own characteristics that differ sometimes by several orders of magnitude. More importantly, no single route remains in a constant state. In our case the situation is more tractable than in general, because the wireless link is in most cases the bottleneck in the route between the mobile host and the fixed host in the Internet. Data packets traveling through the Internet experience a varying propagation delay and have a certain loss probability to congestion on the route. Thus for our purposes a very simple model that would take these factors into account would suffice. An alternative approach would be to perform tests when a fixed host is really located somewhere in the Internet. Our emulator tool allows for such kind of a setup.

Another different need for simulating the Internet comes from the obligation to prove that modifications to TCP improving its performance on wireless links do not have a negative impact on the performance in other environments and do not introduce a congestion danger on the network. This is much harder than building a simple model to reflect the effect of the Internet on a data packet traveling toward the mobile host. Since it is almost an intractable task to evaluate how a modified TCP would perform in all scenarios possible in the Internet, we should be conservative in what changes we can implement, and prefer improvements that have been already widely evaluated by the research community.

5.2 Workload Models

The type of workload used for evaluation of different solutions has a significant effect on the results. Important factors characterizing the workload are the behavior of individual connections, the number of simultaneous connections and their relative position in time. By the relative position in time we mean whether the connections are started at once, or a at a certain time interval. Below we briefly discuss existing types of workload, and outline the workload we have used. Workload parameters are given in detail later in Section 6.2.

Two major classes of connections are recognized: the bulk data traffic and the interactive traffic. Bulk data connections consist of a continuous flow of data packets of the maximum size allowed by the network. The interactive data traffic is of sporadic nature, small varying size packets are sent at irregular intervals. A typical example of a bulk connection is a file transfer using FTP, while a telnet application is an example of an interactive connection. These two traffic classes require different service from the network. For bulk data connections the latency and its variance are not very important, but the total throughput is. For interactive connections, extensive delays irritate the user.

A relatively recent addition to these two traditional traffic classes is the Hypertext Transfer Protocol (HTTP) protocol [BLFF96], which is both of the interactive nature, since the user is waiting for a web page to be displayed, and can transfer a considerable amount of data, for example, in images. This third class represents *transactional* traffic and in addition to HTTP also includes larger database queries. On the average, a duration of a single HTTP/TCP connection is short, often too short to get a valid picture of the network condition. Modern web browsers tend to generate a large amount of simultaneous connections. Such an approach can congest the network, because short connections tend to overestimate the available network capacity. Modeling HTTP requires constructing a complex model of request-reply interaction, which defines the number and size of retrieved objects.

A TCP connection between hosts A and B is a combination of two independent data flows, from A to B, and from B to A. In theory, A can transfer data to B in a bulk data transfer, while B to A as an interactive traffic. However, in practice most bulk data connections are unidirectional, that is, application data is only sent in one direction, and only the acknowledgments are sent in the other direction.

The next question is how many simultaneous TCP connections are present on the radio link. It is quite common for several connections to share the link simultaneously. For example a file transfer is proceeding in the background while the user is browsing the web. The last question is whether the concurrent connections are started at once, or in some time interval, for example 10 seconds.

In most of our tests we use a single unidirectional bulk data transfer as the workload. In the limited set of tests we use two such transfers. We chose such a workload type because it is commonly used, and it is simple to implement and analyze. We do not consider any “background” traffic that would compete with the workload under study. We believe that in mobile computing most data is transferred from the fixed host to the mobile. Indeed, a typical nomadic user is in most cases concerned with obtaining information rather than sending it. Examples of downlink-intensive applications are web browsing, a file transfer and email. The downlink direction is also more interesting from the modeling point of view, because the downlink is more problematic than the uplink. Thus we use downlink transfers in our tests.

In general, a sound TCP study should include a consideration of competing traffic [AF99]. Furthermore, both congestion sensitive flows (e.g. TCP) and congestion insensitive flows (e.g. a UDP video stream) should be studied. We draw most of our observations from experiments without competing traffic, that may seem somewhat limited. However, we aim our study at the specific environment and do not claim that the conclusions hold in general. Indeed, a typical mobile user would mostly have a single or a few TCP transfers concurrently. Resource demanding real-time traffic seems hardly reasonable on a slow wireless link.

5.3 Baseline TCP

In Section 2.1.3 we outline the requirements for a TCP implementation and stated our selection of the Linux OS. Appendix A lists fixes and improvements we have made to the Linux implementation. We refer hereinafter to *baseline TCP* as our TCP implementation with a fixed set of control parameters and algorithms.

Table 1: Features in the baseline TCP.

Feature	Availability
Fast Retransmit, Fast Recovery	ON
New Reno	ON
Initial Window Size, segments	2
SACK	OFF
MSS, bytes	256
Timestamps	OFF
Delayed Acks	ON
Advertised Window, kilobytes	32
PPP Compression	OFF
Control Block Interdependence	OFF

Recent studies of the Internet traffic indicate that both the New Reno algorithm and the Selective Acknowledgment (SACK) option are widely used nowadays [All00]. We have decided to include the New Reno algorithm into the baseline, but leave SACK as one of optimizations. This corresponds to the current practise and makes easier comparisons with the related work. Table 1 presents a list of relevant parameters that we assume in the baseline TCP if not mentioned otherwise. Appendix A gives more details and justification behind such a choice.

6 Experimental Design

6.1 Test Environment

In this section we describe our test environment: how the network model shown in Figure 9 on page 33 is realized in our emulator, and how the workload generator (TTCP, see the next section) is positioned. Figure 10 shows the protocol layering in our setup. The workload source, the Seawind emulator [AGKM98] and the workload sink are each located on a separate computer in Ethernet LAN (802.3). The test TCP traffic is encapsulated into a regular TCP/IP connection. Seawind runs on a normal Linux workstation, as it gets the test TCP traffic from a standard socket interface.

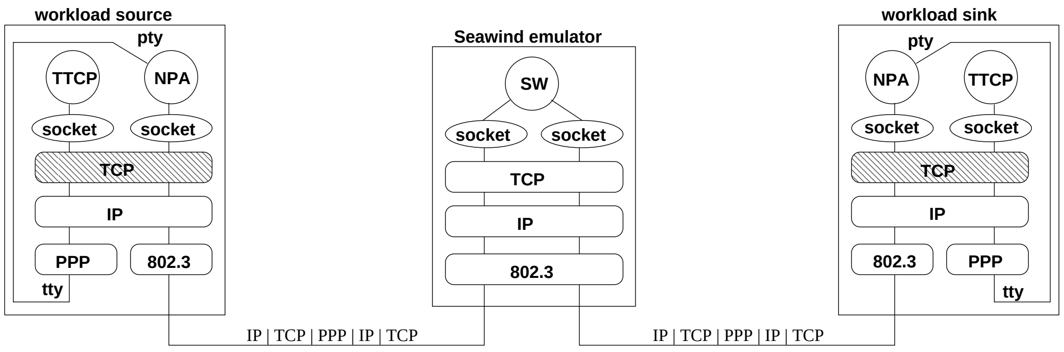


Figure 10: Seawind protocol stack. The modified TCP code is dashed.

The Seawind emulator implements the network model shown in Figure 9 by delaying and dropping TCP segments in real time. The downlink and uplink channels are parameterized independently. The maximum length of the queue in packets is controlled by a parameter; in the "unlimited" mode the length is only bound by the available memory. For the purpose of our experiments, it can be considered infinite. One packet is considered currently "in transmission" and is not counted into the queue limit. Through the rest of the thesis we assume that the "router buffer size" does not include this packet. Link errors can be emulated as a fixed pattern (e.g. 5th and 12th packets are dropped) or with a specified dropping probability per each packet.

The workload source and sink computers use the TCP implementation under study. The Point-to-Point (PPP) protocol is used as a link service for a TCP connection under study. This corresponds to a real-world situation, as most dial-up users employ PPP. We have disabled all kinds of header and data compression, as well as escaping of control characters in PPP (except the flag byte and the escape byte). The PPP/IP/TCP traffic is forwarded by the Network Protocol Adapter (NPA) via a TCP/IP connection to the Seawind emulator.

6.2 Test Network

We use a modified *TTCP* tool for generating traffic for TCP connections. TTCP is a popular public domain tool for testing the end-to-end throughput by sending a high volume of data over the network [Sti90]. TTCP is commonly used as a workload generator for bulk data transfers. We have made several extensions to TTCP to make it more suitable to our needs. In our tests we used 400 writes of 256-byte data blocks, which results in a 100-kilobyte transfer.

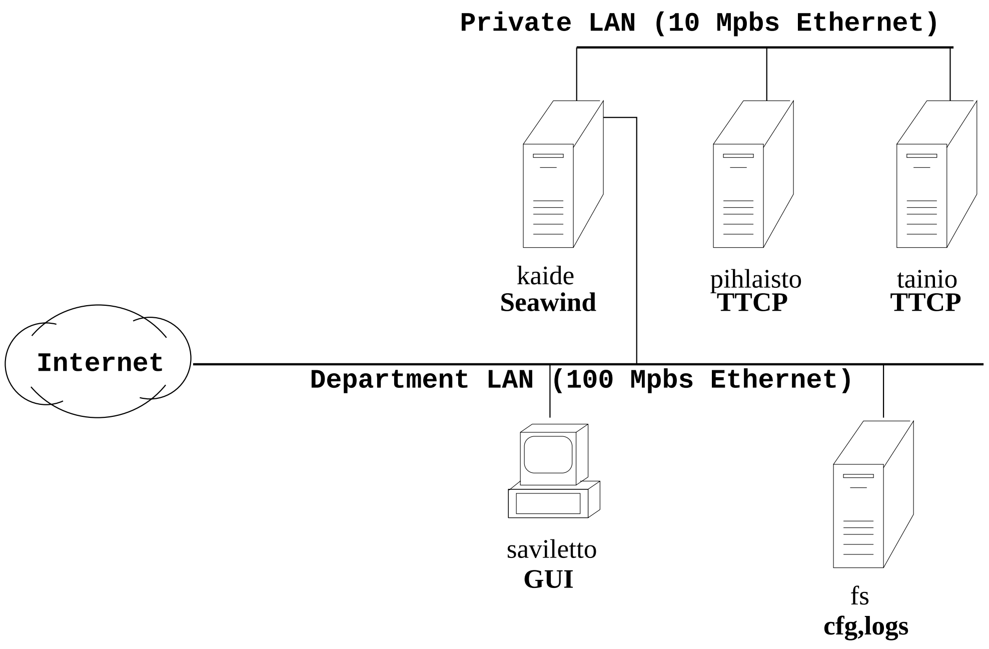


Figure 11: Test Network.

The Nagle algorithm in TCP does not allow transmission of segments smaller than MSS if such a segment is currently in flight [Nag84]. The Nagle algorithm could disturb our results. To prevent this, the size of data passed by TTCP for transmission with a single write is set to the MSS of the connection. We have also afterwards checked the minimum segment size in a connection trace. We could also have used a socket option *TCP_NODELAY* to disable Nagle altogether.

For our tests we have used a specially set up network, as shown in Figure 11. Test TCP connections are performed inside a private LAN, so that interference with other traffic is avoided. From computers in the test network, only *kaide* is connected to the department network. Other computers cannot directly communicate with other hosts but those located in the private LAN. Computers connected to the department LAN can use a file server *fs*. All machines in the test network are 400-MHz Celerons, running RedHat Linux 6.1.

TTCP runs on *pihlaisto* and *tainio* that are used as a traffic source or sink interchangeably. The Seawind emulator runs on *kaide*. The configuration and control of all tests can be done remotely from a computer in the department LAN that runs the user interface. We store configuration and log files in the *fs* server, so that they are accessible from *kaide* and any other computer in the department LAN.

We have used the Linux kernel version 2.2.14 on *kaide* that runs the emulator. In principle, any stable kernel version could be used, since we are not concerned with the details of kernel behavior here. The only requirement is the correct and timely execution of the emulator code. In contrast, *pihlaisto* and *tainio* are running the kernel with the modified TCP implementation as they are used for workload generation.

As all tests are done in a private LAN, the overhead of transmitting workload data from source to emulator to sink is minimal and predictable. In our tests we have ignored this overhead. In the future, it can be measured and substracted from a delay calculations in the emulator. We also have not run any tests where a workload generator is located in the Internet. It can be done in future, by making *kaide* forward packets between the private LAN and the Internet.

6.3 Measurement Data

The experimental data is collected from three sources: the tcpdump, the seawind log and the kernel log. Tcpdump captures a binary dump of the packets at the TCP sender and receiver. Seawind logs down the amount of delay imposed on each packet, the current queue size when a packet is enqueued, and events such as a packet drop due to exceeding a queue limit or as a result of a random error. The kernel log provides values of TCP internal variables, e.g. the congestion window (cwnd), the slow start threshold (ssthresh), the retransmission timeout (rto). We believe that the overhead caused by collecting the logs is negligible and does not affect our results.

For TCP bulk data connections the most important performance metric is the *throughput*. It is defined as the ratio of the amount of user data transfered during the connection over the connection elapsed time (taken at the sender from the first SYN until the last FIN-ACK). We compare the effect of different optimizations based on achieved throughput. In addition we summarize the following metrics for each test: the elapsed time (quartiles and median), throughput (median), number of TCP retransmission (median), number of packets dropped in Seawind (median). For the detailed analysis of some cases we can also produce the trace of the queue size in Seawind. Because of the large number of tests, it is important to use automated tools for the basic analysis. We have developed a set of scripts to produce the statistics and graphs of TCP connections.

The number of replications of a test that we could perform has been limited by time considerations. A typical test connection takes approximately two minutes to complete. Taking into account the broad parameter space, we could make 15 replications per each test. We have used common random numbers as a standard variance-reduction technique, see e.g. [LK91]. All comparison tests of different TCP optimizations at the presence of random error losses were run with the same sequence of packet drops. This allows obtaining statistically sound data with the limited number of replications.

We have extended the Linux kernel to print the values of internal TCP variables into the kernel log. In general, *tcpstats* is used for this purpose [Pad98]; it is the de-facto complement for tcpdump in the TCP analysis. Unfortunately, tcpstats is available only for BSD Unix OS. In the future, we plan porting tcpstats to Linux and developing an analysis tool similar to MultiTracer [LRK$^+$99]. The new tool will allow automatically combining and analyzing data from tcpdump, tcptrace and the Seawind emulator.

6.4 Test Cases

Here we define a scope and parameters of our test sets. To select interesting cases that are worth a detailed study, a large number of different optimizations with a smaller number of repetitions was run in preliminary tests.

Unlimited router buffer. In the first set of tests we examine the behavior of TCP in the environment with the unlimited router buffer. We assume the presence of constant bandwidth and propagation delay, but lack of any other adverse factors, such as variable delays, error or congestion losses. We will use the baseline TCP in this set of tests.

Optimal router buffer size. The purpose of this set of tests is to determine a range of router buffer sizes where TCP performs well on an error-free link. We will use the baseline TCP and the router buffer size of 3, 4, ..., 12, 15, 20, 40 packets.

Single-packet error losses. We will try to locate and analyze the cases when an error loss of a single segment has a significant effect on performance. We drop a segment at different places in a connection. In general, we expect single-segment errors to be recovered well by the fast retransmit algorithm without noticeable performance degradation. An interesting effect can be expected when error losses happen at the point of time when the router buffer overflows. Also errors at the beginning of a connection would interrupt the slow start and force the congestion avoidance. It will be interesting to find out whether this would reduce the throughput of the connection. We expect bad effects to be caused by retransmission timeouts. We will use the baseline TCP in this set of tests and a varying size of the router buffer.

Random errors losses. In our largest set of tests we study the effect of random error losses on the TCP performance. To determine the packet loss rates to experiment with, we took 1 % as the lower bound of interest. The non-congestion loss rate of less than 1 % is given as a condition for "normal" operation of the TCP protocol [Ste97]. Some of the related work has experimented with a broad range of loss probabilities (from 0.001 to 1) [MSMO97]. We selected the uniformly distributed loss rates of 2 %, 5 % and 10 %. An alternative and possibly better option would be to fix the Bit Error Rate (BER) and compute the loss probability separately per each packet based on its size. Unfortunately, our emulator tool does not yet allow this. Following a practice commonly used in the related work, the error rates are assigned to good, mediocre and poor radio link conditions. We have selected the router buffer sizes of 3, 5, 7, 10 and 20 packets.

Table 2 lists the optimizations we have experimented with. There is a large number of possible interesting combinations, for example the limited receiver window and the increased initial window. However, due to the time limits we did not tests the combinations. The receiver's advertised window size was limited by setting the receiver buffer with a socket option. Although we have used two and four kilobytes for the size of the receive buffer, the actual advertised window was slightly less in TCP traces. This is the reason for the values given in Table 2.

Not only the total throughput at the end of the connection is interesting, but also the throughput taken at different stages of a connection, e.g. when 15 kilobytes of data was sent. In this way we can estimate the behavior of shorter transaction-type connections. Due to concerns about the validity of such approach, no detailed conclusions on the transactional workload can be made. We could not run tests with the transactional workload because of time limits.

Burst error losses. We will perform a limited number of tests with the baseline TCP to study the effect of error bursts on TCP performance. We will experiment with three-packet and ten-packet router buffers. The length of the burst period is set to ten and twenty seconds and the loss rate during the burst to 20% and 40%. We will trigger an error burst in the middle (after 60 seconds) of an otherwise error-free connection.

RED. It is interesting to experiment with RED parameters to determine a set of values appropriate for our environment. The goal for a single connection is to prevent the start-up buffer overflow. For the case of two parallel connections, we also evaluate how fair the bandwidth is shared. We have selected three router buffer sizes of 10, 20, and 40 packets. We have omitted smaller sizes from our experiments, because the start-up buffer overflow problem is not severe for a small buffer. In addition, we do not expect RED to perform well for a small buffer size. The weight of the current queue size in calculating the moving average of the queue size should be rather high to detect the start-up buffer overflow. The maximum drop probability should not be high to prevent retransmission timeouts. During preliminary tests we have identified the following parameters to be interesting for the detailed experimentation: the queue weight of 0.2 and 0.4, the maximum drop probability of 0.05 and 0.1. The minimum and maximum thresholds are fixed at 25 % and 75 % correspondingly. We will use the baseline TCP

Table 2: TCP optimizations tested with random errors.

Label	iw segm.	win bytes	sack on/off	newreno on/off	mss bytes
baseline	2	32696	off	on	256
iw3	3				
iw4	4				
win2K		2048			
win4K		3840			
sack_on			on		
newreno_off				off	
mss536					536

over an error-free link in this set of tests.

7 Measurement Results and Analysis

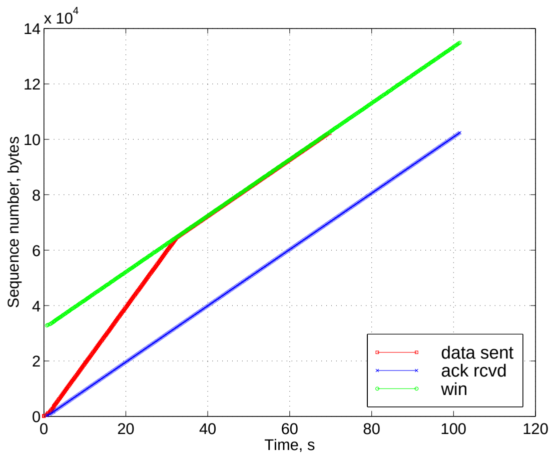
In this section we provide and analyze the most interesting cases found in the tests. The complete results of tests are given in Appendix B.

7.1 Unlimited Router Buffer

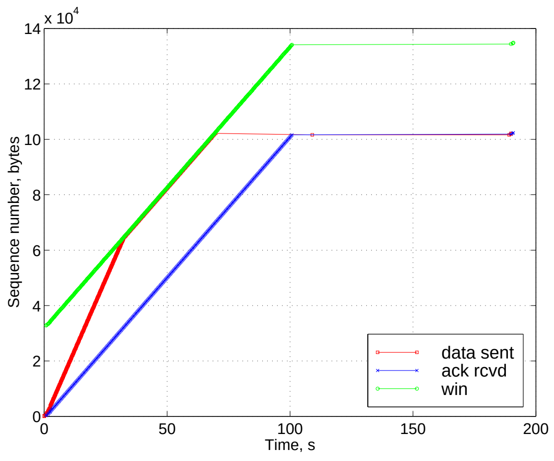
First, we examine the behavior of TCP over a router buffer of unlimited size. We assume the presence of constant bandwidth and the propagation delay, but no variable delays, error or congestion losses. Detailed results are given in Appendix B.1. Figure 12(a) shows the behavior of the baseline TCP under such conditions. The achieved throughput of 1002 bytes per second (Bps) is close to the maximum in this environment. Taking into account 20 bytes of TCP header, 20 bytes of IP header, 5 bytes of PPP header and 3 bytes of PPP trailer, a SYN segment of 56 bytes and SYN-ACK of 57 bytes, and the elapsed time of 102.15 seconds we get the raw throughput of 1192 Bps. This is very close to the line rate of 1200 Bps.

TCP behaves as expected; the FlightSize increases until the size of the receiver window is reached. When it happens, the FlightSize stays constant at the size of the receiver window; the congestion window does not affect the connection. When the FlightSize equals the receiver window (32 kilobytes), 127 segments are queued in the network. In our environment, where the FlightSize of a few segments is sufficient for utilizing the pipe capacity, this is undesirable for reasons discussed in Section 3.1 on page 19. We will add here that the measured RTT also includes the queuing time and thus is highly inflated with regard to the actual RTT of the link. It can be seen from Figure 12(a) that RTT reaches 30 seconds compared to less than a second in the beginning of the connection. A second connection started over the same link will experience a timeout because the time required to get an acknowledgment for the first packet would be greater than the initial RTO value (3 seconds). The second connection will have difficulties in obtaining a fair share of the link bandwidth, when the first connection has effectively blocked the link.

An adaptive link layer can change the strength of the radio channel coding



(a) No segment losses.



(b) Three last segments and the first retransmission are lost.

Figure 12: TCP behavior with the unlimited router buffer size.

when the number of transmission errors on the link changes [Lud00]. A stronger coding schema allows to reduce the packet loss rate over the link at the expense of the reduced line rate. A complete lack of segment losses creates the overbuffering problem and is not desirable in our environment. In order to keep the FlightSize at the optimal level, TCP needs a low rate of segment losses. Losses due to congestion at the router buffer and losses due to link errors are treated in the same way by TCP. Thus a link can provide a low level of error losses with a beneficial effect on TCP. For an adaptive link it means that the channel coding can be kept as weak as possible to maximize the line rate, leaving a low level of packet losses to be noticed and corrected by TCP.

7.2 Optimal Router Buffer Size

The purpose of this set of tests is to determine a range of router buffer sizes appropriate for TCP in our environment. We have used the baseline TCP and an error-free link and set the router buffer size to 3, 4, ..., 12, 15, 20, 40 packets. Detailed results are given in Appendix B.1. The achieved throughput is shown in Figure 13. We observe that throughput is good for buffer sizes in the range of 3 to 12 packets. A buffer size as small as three packets already allows filling the pipe capacity. Figure 14 shows the receiver-side trace of a TCP connection over the router buffer size of three packets. Segments are arriving with an interval equal to the transmission delay of the link. Periodic packet losses at the end of each congestion avoidance cycle do not affect the performance. This is because a packet loss occurs “before” the wireless link and no data is actually retransmitted over the wireless link. These losses are recovered well by the fast retransmit, and although the lost segment is delivered behind five other segments, a bulk transfer application does not notice the delay.

Variations in throughput for the buffer size of 3-12 packets have a simple explanation. If a congestion control cycle happens to occur at the end of a connection, it causes a retransmission timeout. Figure 15 shows that the connection suffers from the RTO for the six-packet router buffer, but does not for the five-packet buffer. The connection in Figure 15(b) simply does not have any more data to send when the periodic packet loss due to a router buffer overflow occurs. Three DUPACKs cannot arrive making the fast retransmit impossible. Whether RTO occurs or not for the given router buffer size depends on the amount of data sent over the connection.

Starting with the buffer size of 15 packets, performance decreases. The severity of the start-up buffer overflow increases with the buffer size. Some examples are given later in Section 7.4.2 on page 58. The start-up buffer overflow for a 15-packet buffer already lasts for 40 seconds. Thus, we can conclude that any buffer size in the range of 3-12 packets offers the adequate performance for a TCP connection in our environment.

7.3 Single-Packet Error Losses

Normally, single-packet error losses do not have a significant impact on TCP performance. We have identified the following special cases when a single-packet loss has a notable effect. We have not run a systematic set of tests for this section, but searched for interesting events in all test sets and experimented with dropping

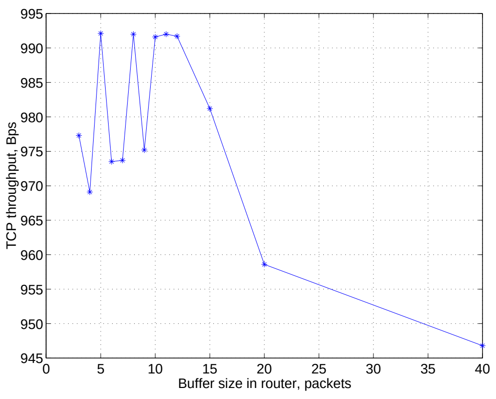


Figure 13: Throughput vs. buffer size for the baseline TCP over an error-free link.

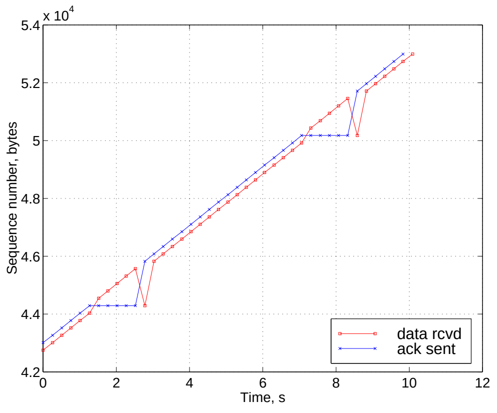
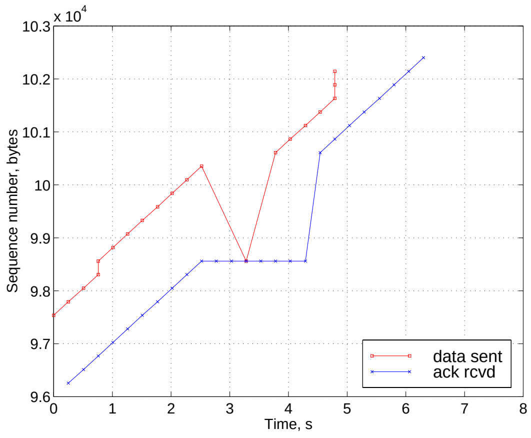
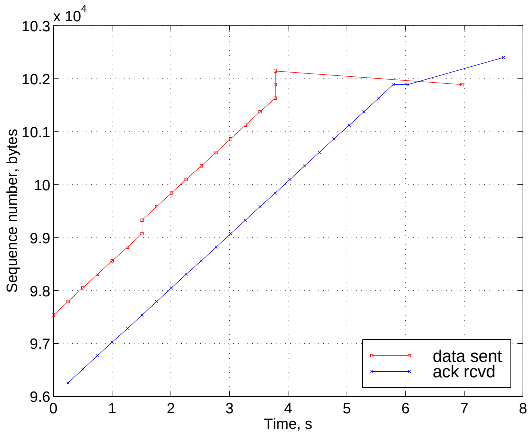


Figure 14: Even a buffer size of three packets allows filling the pipe capacity. The receiver-side trace.



(a) Buffer of 5 packets, the fast retransmit.



(b) Buffer of 6 packets, the RTO.

Figure 15: The congestion avoidance cycle causes periodic packet losses. If a loss is at the end of the connection, RTO occurs.

segments at the different phases of TCP connections. The cases concern a loss of:

a) the SYN, SYN-ACK, the first segment from the initial window,
b) a data segment when the number of packets in flight is less than four,
c) a retransmitted segment.

In all cases, the adverse effect is related to the retransmission timeout, because the fast retransmit is impossible in these situations.

In case a) a valid RTT sample has not yet been collected; the default value of three seconds is used for RTO. A usual timeout value without a back off based on RTT measurement is three to five seconds for our connections. Thus, the initial timeout value of three seconds is appropriate for our environment. Figure 16 shows an example where the first segment in the initial window of two segments is lost. The sender has to recover by a timeout to retransmit the lost segment. Furthermore, the retransmission is lost, and the second recovery attempt is made after back off of RTO as discussed in case c).

In case b) there are not enough packets in flight to reach the threshold of three DUPACKs to trigger the fast retransmit. The FlightSize of less than four segments occurs when a connection is limited by the congestion or receiver window, or does not have enough data to send. For a bulk data connection, the latter is only possible at the end of the transfer. Figure 15 shows an example of a timeout at the end of the connection. In case segment losses *do* suddenly happen on an overbuffered link (with a large router buffer), the recovery time is long, as shown in Figure 12(b) on page 46. Four last segments in a connection are lost (three original and the first retransmission). The retransmission of a lost segment happens only after 40 seconds after its loss.

The receiver window can be set smaller than four MSSs only in pathological cases. On the other hand, a small congestion window preventing the fast retransmit is a common case. In the beginning of a connection the congestion window is not yet large enough to allow fast retransmit. In another part of a connection, the congestion window may be small as a result of packet losses. Figure 25 on page 57 shows RTOs when the congestion window is too small to allow fast retransmits. However, the penalty of RTO recovery compared to the fast retransmit

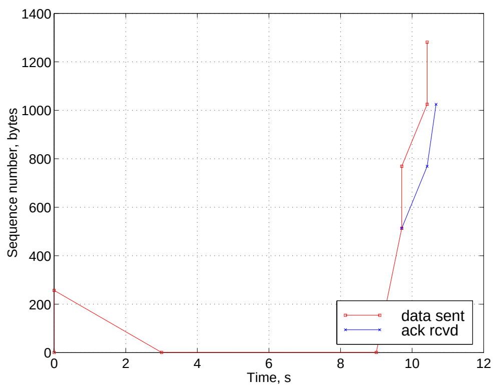


Figure 16: Loss of a packet in the initial window causes a timeout.

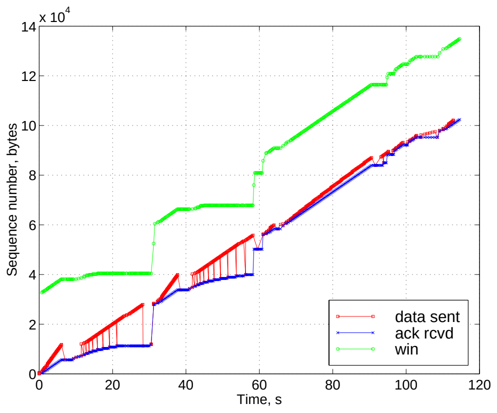


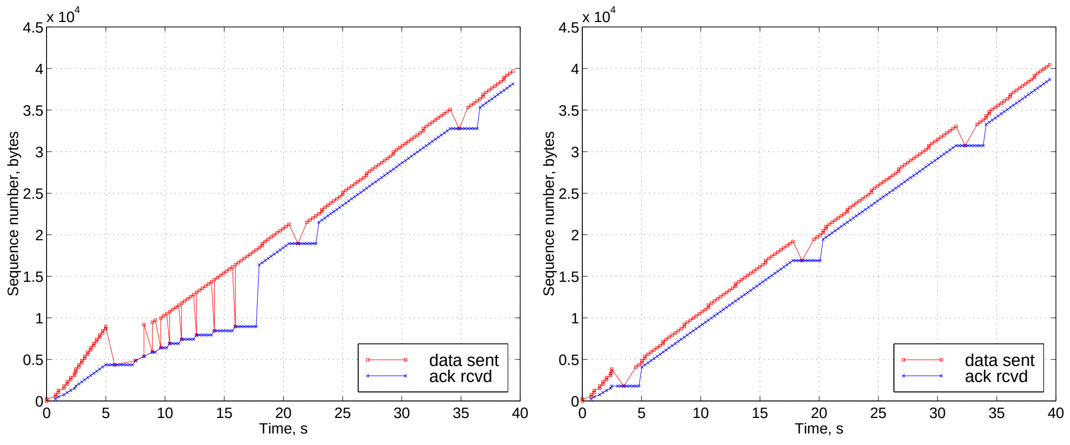
Figure 17: Repeated buffer overflows when the slow start threshold is set too high after RTO.

is not large in such a case.

In case c) a segment retransmitted by the slow start or by the fast retransmit is lost. Note that fast retransmits are not allowed for segments retransmitted by the slow start after RTO [APS99]. A loss of a segment retransmitted due to a timeout causes an RTO back off; the timeout value is doubled on each unsuccessful retransmission attempt. Figure 16 shows the back off of RTO from three seconds to six seconds at the second retransmission. The backed off RTO value can be as large as two minutes. The back off has a particularly bad effect if the original RTO value was inflated. In this case the backed off RTO can be a minute already at the second retransmission, as shown in Figure 12(b) on page 46. The connection time is doubled.

A loss of a fast-retransmitted segment or a segment retransmitted during the fast recovery phase by New Reno always leads to a timeout. This introduces a notable delay, as can be seen in Figure 24 on page 57. Another potential problem is shown in Figure 17. The loss of a retransmitted segment during the fast recovery phase causes a timeout. At this time, the FlightSize is huge. The slow start threshold is set to a half of the FlightSize, which is more than the FlightSize at the time when the first packet loss is detected. The slow start threshold, which is too high, overestimates the available network capacity and the second buffer overflow happens similar to the start-up buffer overflow. Such a bad behavior can continue through the connection lifetime. A correct interpretation of the congestion control standard [APS99] requires to halve the slow start threshold twice if a loss of a retransmitted packet is detected.

A *positive* effect of an error loss can be in the prevention of the start-up buffer overflows. Figure 18 shows two graphs, one without error losses and another when the eighth segment is lost due to an error. An error loss triggers a fast retransmit that decreases the transmission rate and limits the value of the slow start threshold early. The start-up buffer overflow is avoided, and the connection proceeds smoothly for its lifetime.



(a) No error losses (b) The 8th packet is lost

Figure 18: A single-packet error prevents the buffer overflow. Baseline TCP over the seven-packet router buffer.

7.4 Random Error Losses

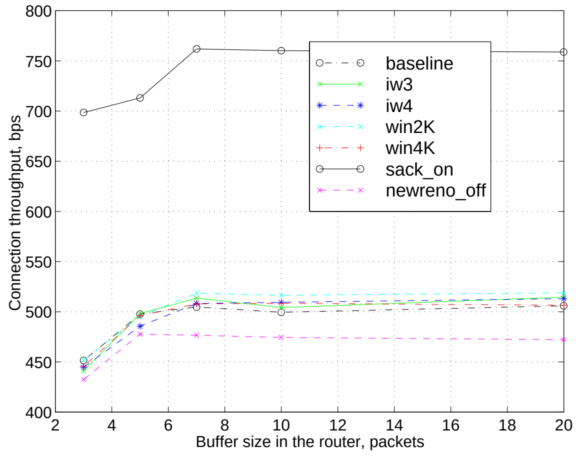
7.4.1 Throughput at the end of connections

In this section we compare the performance of TCP optimizations based on the achieved throughput at the end of a connection. Detailed results are given in Appendix B.2–B.8.

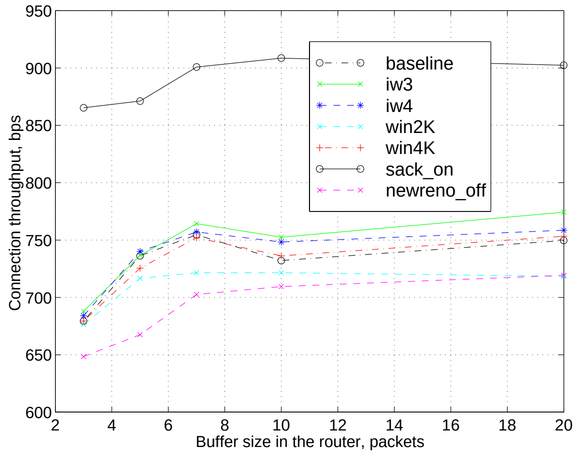
Good radio conditions. In this set of tests the error loss rate is relatively low, an average eight packets and eight acknowledgments are dropped in a connection. The performance picture we observed with different TCP optimizations is shown in Figure 19(a). The general trend is an increase of throughput from the buffer size of three packets up to ten packets for about 6 %. For the 20-packet buffer the throughput increases for a limited receiver window, decreases for a larger initial window, and stays the same for other optimizations. We explain it as follows. For smaller router buffer sizes, there is a higher probability of RTOs due to a short congestion avoidance cycle and due to a small FlightSize at the presence of error losses. For larger buffer sizes, the severity of the start-up buffer overflow increases. In general, all modifications and the baseline perform well; the achieved throughput is less than the maximum by only 3-15 %. Multiple error losses rarely occur in a single FlightSize of data; most losses are recovered well by a fast retransmit. Errors do have some effect: the throughput is less than in an error-free link as was shown in Figure 13 on page 48. The key performance problem is the start-up buffer overflows.

TCP with SACK provides better throughput than other modifications for all buffer sizes and about 7 % better than the baseline. The performance gain is achieved mostly by more efficient recovery from the start-up buffer overflow, as shown in Figure 20(a). Other modifications, in general, perform slightly better than the baseline. TCP with the increased initial window achieves 1-2 % better throughput than the baseline. The reason for the improvement is the prevention of RTOs when segments are lost in the very beginning of a connection, when the FlightSize may not yet be enough to trigger a fast retransmit. Figure 21 illustrates the idea. The initial window of four segments is better than of three segments for all buffer sizes but for a ten-packet buffer. There are two scenarios which explain this. In the first scenario, a larger FlightSize caused by the initial window of four segments triggers one additional DUPACK that resets the retransmission timer. A few more seconds are required before the RTO timer expires, Figure 22. In another scenario, the larger FlightSize causes another buffer overflow due to the slow start threshold, which is too high, as was shown in Figure 17 on page 50. The second buffer overflow increases the connection time by about ten seconds.

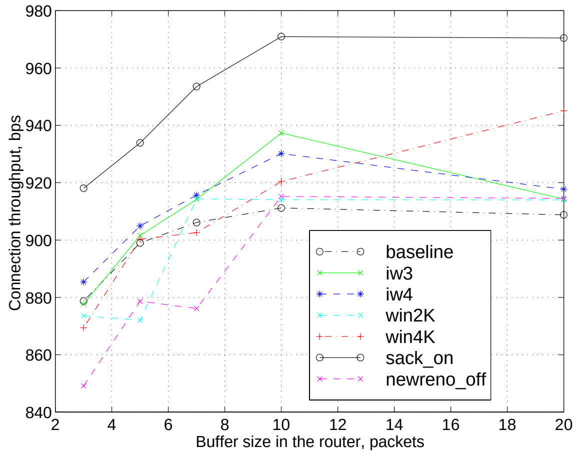
A limited receiver window yields worse throughput than the baseline for a



(c) Poor radio conditions (10 % error rate)

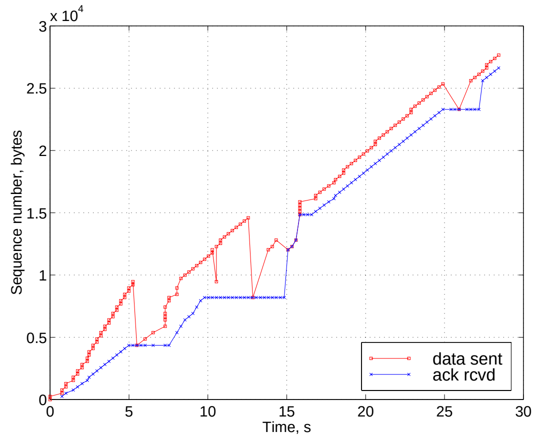


(b) Mediocre radio conditions (5 % error rate)

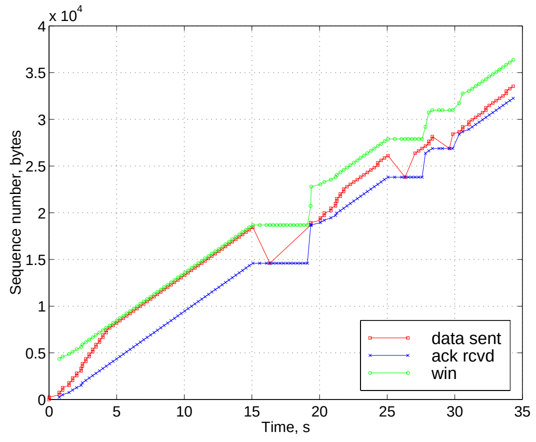


(a) Good radio conditions (2 % error rate)

Figure 19: Throughput of TCP at the end of connections. The average over 15 replications. Note the different scale.

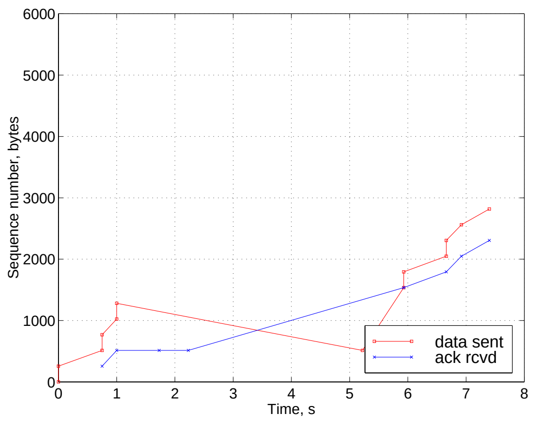


(a) Efficient recovery by SACK. The router buffer is seven packets.

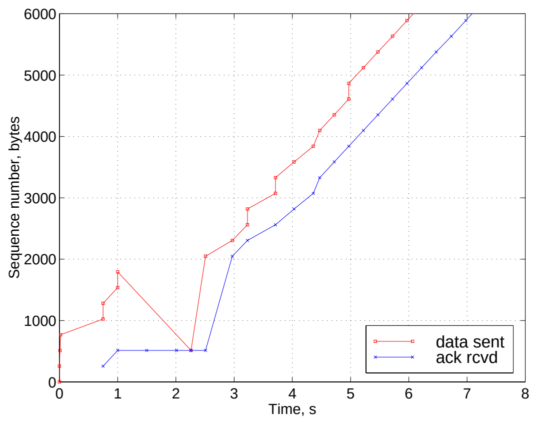


(b) Efficient prevention by a limited receiver window of four kilobytes. The router buffer is twenty packets.

Figure 20: Possible solutions to the start-up overflow of the router buffer.

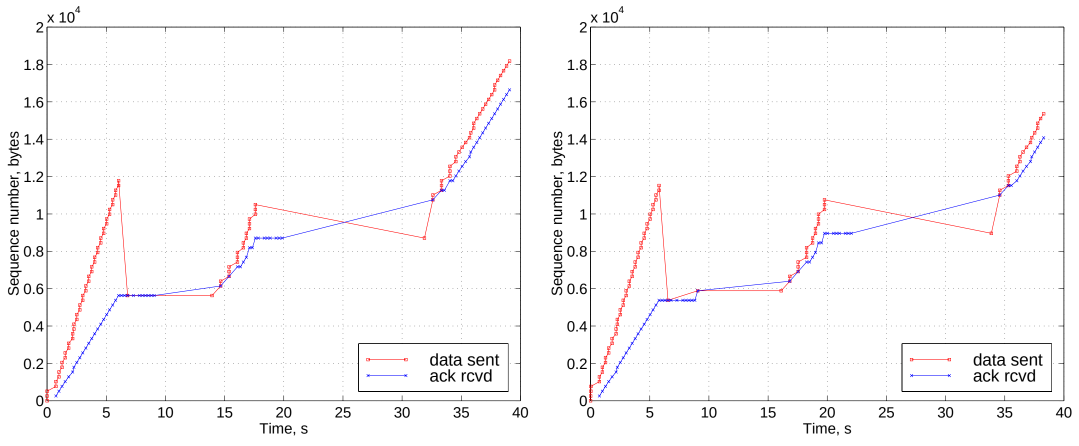


(a) The baseline TCP (initial window of two segments).



(b) Initial window of four segments.

Figure 21: The RTO in the connection start due to an error loss is present for the baseline, but is prevented by the increased initial window.



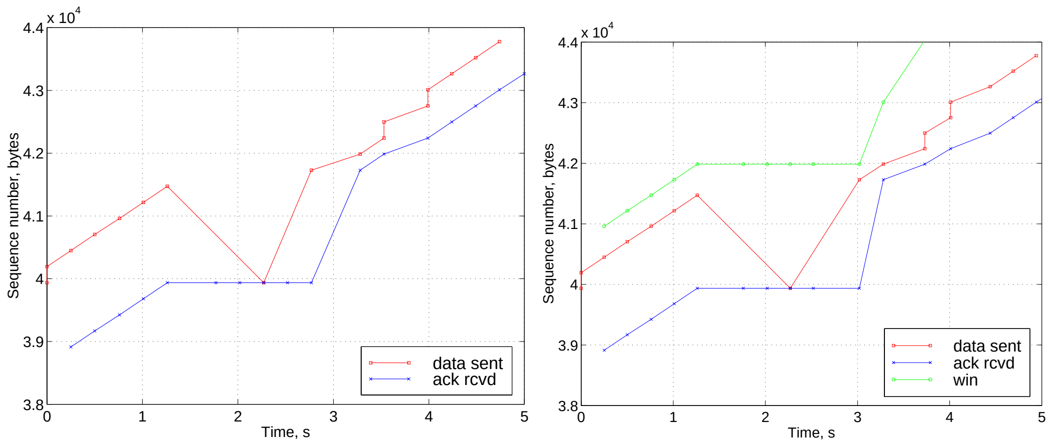
(a) Initial window of three segments.

(b) Initial window of four segments.

Figure 22: The larger initial window triggers a partial ACK that resets RTO and delays the recovery of other segments.

buffer size of up to ten packets. A smaller window size limits the FlightSize and decreases the buffer overflow in the connection startup. However, this advantage is overruled by disturbing the fast recovery procedure. This happens as follows. After a fast retransmit the baseline TCP transmits a new segment per each additional DUPACK during the fast recovery. A small receiver window prevents the transmission of new segments during the fast recovery as shown in Figure 23. For a larger buffer size, limiting the receiver window improves the throughput. With a larger buffer size, fast retransmits at the end of congestion avoidance cycles are less frequent and thus limiting the fast recovery at each fast retransmit has less effect. In contrast, the prevention of the start-up buffer overflow becomes increasingly important. Figure 20(b) on page 54 shows that the start-up buffer overflow is reduced to the minimum, and the fast recovery is not disturbed afterwards.

TCP without New Reno (and SACK) normally recovers by RTO when two or more segments are lost from the same flight. We noticed that it can have either a bad or a good effect on performance. When a few segments are lost, New Reno recovers faster than Reno. However, when a large number of segments are lost, the slow start recovers much faster than New Reno which is retransmitting only one segment per RTT. For a buffer size of ten and twenty packets Reno performs



(a) Receiver window of 32 kilobytes.

(b) Receiver window of 2 kilobytes.

Figure 23: The limited receiver window prevents sending new data after the fast retransmit.

better than New Reno, because a large number of segments lost in the start-up buffer overflow is recovered faster.

Mediocre radio conditions. In this set of tests the error loss rate is moderate, an average 20 packets and 20 acknowledgments are dropped in a connection. The performance picture we observed is shown in Figure 19(b). As a general trend, the throughput increases from the buffer size of three packets to the buffer size of seven packets. This is because for smaller router buffer sizes, there is a higher probability of RTOs due to a short congestion avoidance cycle and due to a small FlightSize at the presence of error losses. For a part of TCP optimizations the throughput drops slightly for a buffer size of ten packets, and increases again for the buffer size of 20 packets. We found that connections over the buffer size of ten packets are prone to timeouts after the fast retransmit, as shown in Figure 24.

The buffer size of 20 packets may actually have less frequent start-up buffer overflows than a smaller buffer size. This is because it takes a longer time to increase the FlightSize to a large enough value to cause the overflow. There is a higher probability of an error loss during this time. If an error loss occurs, the buffer overflow is reduced or prevented. SACK TCP achieves about 10 % less than the maximum throughput, other optimizations about 30 % less. Start-up buffer overflows are not frequent and less severe than in “good radio” tests. The frequency of TCP timeouts is moderate, one-two per connection. The back off of RTO has an especially bad effect on performance.

The effect of modifications is generally the same as in “good radio” tests. A small receiver window of two kilobytes performs worse than the baseline for all buffer sizes because it disturbs the fast recovery after the fast retransmit. As the amount of error losses is greater than in “good radio” tests, fast retransmits are more frequent. The distortion of fast recovery overrules the prevention of buffer overflows. A similar reasoning holds for Reno TCP, the more efficient recovery from buffer overflows is less important here than the recovery of multiple error losses. We observe that Reno performs worse than the baseline for all buffer sizes.

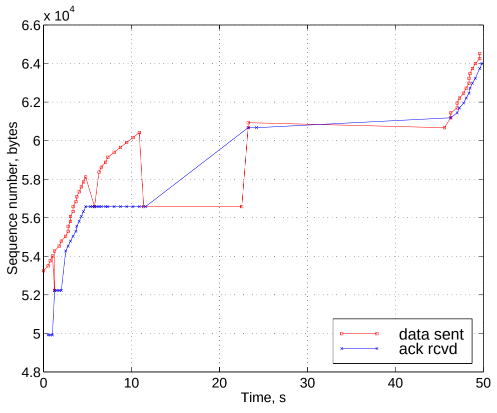


Figure 24: RTO occurs if the fast retransmitted packet is lost.

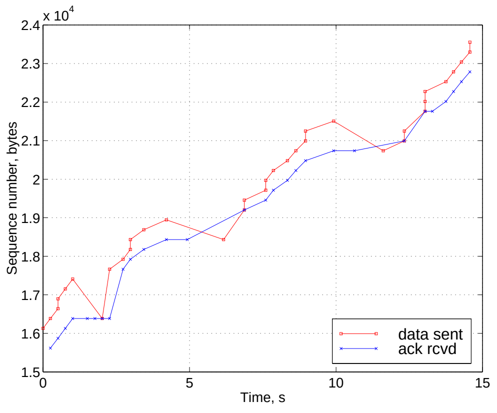


Figure 25: RTOs are not much worse than the fast retransmit when the FlightSize is small.

Poor radio conditions. In this set of tests there are 40 segments and 40 acknowledgments lost on the average in a connection. The performance picture we observed is shown in Figure 19(c). The throughput increases from the router buffer size of three packets up to seven packets. This is because for smaller router buffer sizes, there is a higher probability of RTOs due to a short congestion avoidance cycle and due to a small FlightSize at the presence of error losses. For buffer sizes above seven packets, the throughput stays nearly the same. This is because at the high level of error losses the FlightSize is small and thus start-up buffer overflows are not present for a larger buffer size. SACK throughput is about 25 % less than the maximum, other modifications achieve only about 50 % of the maximum. The main reason of the performance problem is the back off of RTO.

SACK TCP proceeds smoothly with a FlightSize of a few packets, timeouts are rare. The difference in performance among other TCP optimizations is not significant. Due to the small FlightSize the RTT on the link is close to the minimum and RTO is not inflated. This makes timeouts less severe than in a case of overbuffering. Figure 25 shows that the RTO recovery is not much worse than with a fast retransmit.

At a high level of error losses TCP tends to underestimate the available network capacity and to spend time in waiting for the RTO to expire. As all TCP optimizations but SACK utilize only half of the line rate, using two or more parallel connections could yield better total throughput (we have not yet run tests that can confirm or disprove this). Although each individual connection proceeds slowly, the total throughput for an application is improved. Such an approach can improve the performance of applications without any modifications to the baseline TCP. For example, many web browsers open several concurrent TCP connections which is beneficial at the presence of error losses.

Before we have modified Linux TCP to use all valid samples for resetting the backed off RTO and the counter of retransmissions as described in Appendix A.2, some of connections terminated before all data has been sent. We find it useful to increase the maximum number of retransmission attempts when operating over a lossy link.

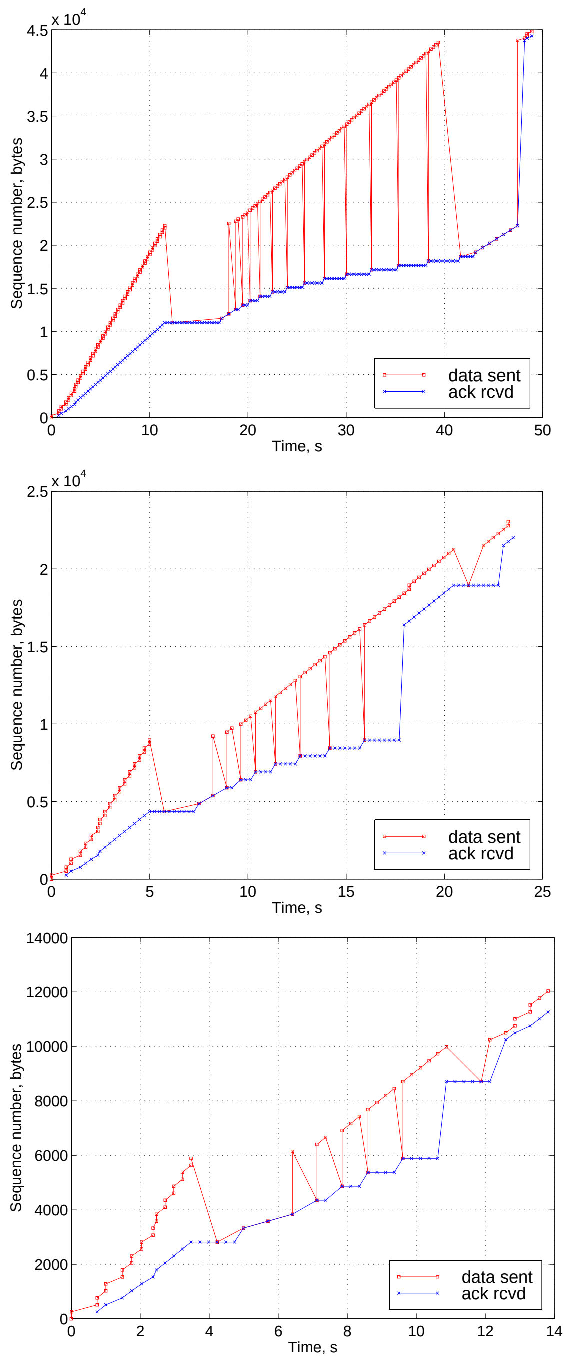
7.4.2 Throughput at the beginning of connections

In this section we evaluate the performance of TCP optimizations at the beginning of a connection. In such a way we can estimate the performance for a transactionlike traffic composed of shorter connections. We calculate throughput for n bytes based on the elapsed time until the nth byte is acknowledged. Our estimation is not fully accurate, because the segments just before n are affected by segments that carry data beyond n that are not present in a connection sending only n bytes. For example, Figure 15(b) on page 48 shows a timeout at the end of the connection. If some more data would be send over the connection, the timeout could be avoided.

We would like to select n in a way that would capture the effect of the start-up buffer overflow. This is not straightforward, as the number of bytes transmitted before the overflow and the length of the recovery from the overflow depends on the buffer size. Figure 26 shows the overflow and the recovery for the router buffer of three, seven, and twenty packets. The value of n of 15 kilobytes is appropriate, because it captures the overflow and a part of the recovery for a buffer size of twenty packets. Also it is not very distant from the end of the recovery for a three-packet buffer.

Figure 27(a) shows the throughput of a connection in “good radio” conditions with a 2 % packet loss rate after 15 kilobytes have been transfered. SACK performs well for all buffer sizes, about 15 % less than the maximum. The baseline TCP and the increased initial window TCP suffer from a long recovery and do not perform well for larger buffer sizes. In opposite, a limited receiver window prevents buffer overflows and performs increasingly well for larger buffers. TCP with disabled New Reno recovers faster than the baseline TCP for larger buffers.

Figure 27(b) shows throughput of a connection in “mediocre radio” conditions with a 5 % packet loss rate after 15 kilobytes have been transfered. At this error rate the prevention and recovery of buffer overflows becomes less important. We observe that the limited receiver window and Reno do not perform as well as in “good radio”. SACK performance is not worse than in “good radio” conditions. This is because SACK is able to avoid practically all RTOs in the beginning of connections. Using SACK in combination with the FACK algorithm allows to trigger the fast retransmit even when the FlightSize is less than four segments. TCP with the increased initial window performs slightly better than the baseline TCP because some timeouts due to error losses are avoided.



(c) 20-packet router buffer.

(b) 7-packet router buffer.

(a) 3-packet router buffer.

Figure 26: Buffer overflow at the beginning of connections. Note the different scale.

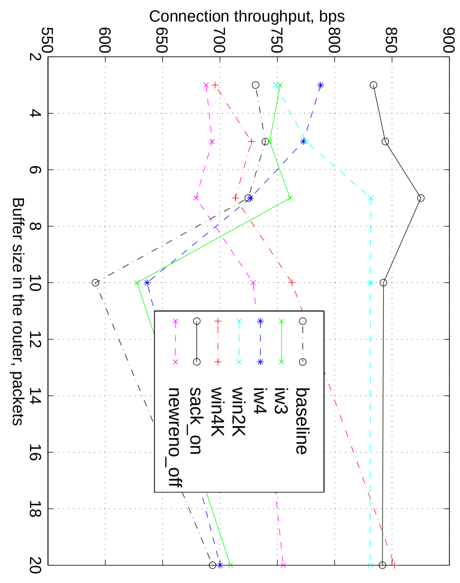
Figure 27(c) shows throughput of a connection in “poor radio” conditions with a 10 % packet loss rate after 15 kilobytes have been transfered. In this environment start-up buffer overflows are rare and the key problem is retransmission timeouts. SACK performance is good, about 25 % less than the maximum. Other modifications achieve only half of the possible throughput.

7.5 Burst Error Losses

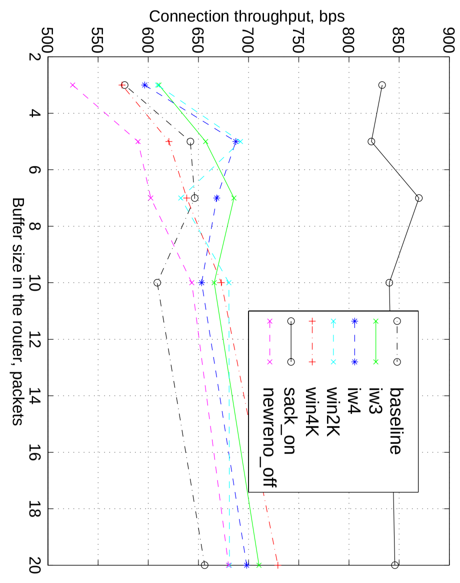
We have performed a limited number of tests with the baseline TCP to study the effect of an error burst on TCP performance. Tests were run with a three-packet and ten-packet buffers, burst length of ten or twenty seconds and a packet error loss rate of 20% and 40%. We triggered an error burst after 60 seconds from the connection start. There were no error losses other than during the error burst. Detailed results are given in Appendix B.9. In general, we found that TCP is prone to RTOs already at a 10-second burst with a loss rate of 20 %. This occurs because in the beginning of the burst the FlightSize is halved several times down to the threshold value of four packets; after that already a single-packet loss causes an RTO. We noticed that normally little or no data is transferred during an error burst. A back off of RTO to a value larger than the burst length causes the connection to be idle even after the error burst, when the link quality is perfect. We also noticed that New Reno is not very useful for recovery during an error burst, because of the high probability of a retransmission loss. The loss of a segment retransmitted by the New Reno algorithm anyway leads to a timeout.

For a three-packet buffer we have mostly observed several timeouts during a burst; the small FlightSize does not generate enough DUPACKs to trigger a fast retransmit. However, RTO is not inflated and even in the case of backed off RTOs transmission resumes shortly after the error burst ends. Because the FlightSize is small, some new segments that are not retransmissions can often be sent during the burst. Getting some new segments through is important, because an ACK for non-retransmitted data can be used to reset the backed off RTO to the normal estimate of RTT.

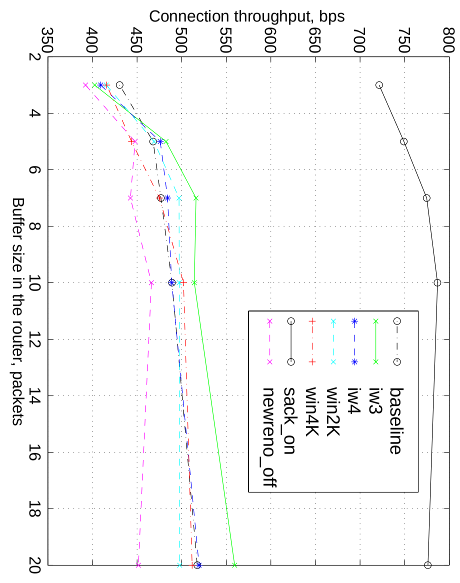
For a ten-packet buffer we have observed a number of varying combinations



(a) Good radio conditions (2 % error rate)



(b) Mediocre radio conditions (5 % error rate)



(c) Poor radio conditions (10 % error rate)

Figure 27: Throughput of TCP at the beginning of connections after 15 kilobytes was transfered. The average over 15 replications. Note the different scale.

of fast retransmits and RTOs. A connection often stays idle after the burst ends for a time of approximately half of the burst length. A large FlightSize at the time of the burst occurrence prevents sending new data during the burst. Thus, no valid RTT sample can be collected and a backed off RTO cannot be reset. In addition, RTO at the time of the burst is already inflated because of the queueing time. Using a smaller buffer size is beneficial in the environment where error bursts are present.

7.6 Random Early Detection

We have experimented with a broad range of parameters of the RED algorithm to determine a set of values appropriate for our environment. Using values of parameters recommended for “normal” routers [FJ93] is definitely inappropriate. This is because a “normal” router typically has a large number of simultaneous flows and connects high bandwidth links, while in our case only one or few connections exist and the link is slow.

We did not find parameter values that would give a performance improvement for a single connection. Detailed results are given in Appendix B.10. RED is not able to prevent the start-up buffer overflow, which is the main performance problem for a single connection. Calculating a packet marking probability based on a moving average of the queue size is not efficient for this purpose. TCP increases the transmission rate very rapidly during the slow start, but the moving average increases slower (Figure 28). A drop decision based on the increase of the moving average comes too late, because the queue overflow has already occurred. Furthermore, additional dropped packets would only harm the performance because the connection would not reduce its transmission rate anyway until the first packet loss due to the buffer overflow is detected.

For two simultaneous connections started at the interval of ten seconds using RED gives an improvement in throughput and fairness for larger buffer sizes. We use the fairness index as defined in [Jai91]. The performance of two baseline TCP connections over a drop-tail router is shown in Table 3. The explanation of the table format is given in Appendix B on page 85. Using RED with appropriate parameters gives an improvement of 10 % in throughput compared to the baseline for a twenty-packet and forty-packet buffer, as shown in Table 4. The fairness among the connections is slightly improved as well. Detailed results are given in Appendix B.12 and Appendix B.11.

Here we present the detailed analysis of the start-up buffer overflow shown in Figure 29. The segment marked *1,* is the last segment transmitted before the overflow is detected after the third DUPACK (*2*) for the lost segment (*3*). The number of segments between *1* and *2* is the FlightSize when a packet loss is detected, it is about twice as large as the router buffer. Approximately, every second segment from this flight is lost due to the buffer overflow. The time between points *2* and *3* shows the current RTT of the link, it is about six seconds. The number of segments between *3* and *5* is the FlightSize at the moment when the first loss occurs. Thus, the segment marked *5* is the latest segment to be dropped by an active queue management algorithm, so that a packet loss is detected before point *3*. When a loss of *5* would be detected, the FlightSize is not grown anymore and additional losses are prevented. The number of segments between points *6* and *7* is the minimum FlightSize to trigger the fast retransmit, four segments. Thus, segment *8* is the earliest segment of the connection, which loss would be recovered by the fast retransmit. The segments before that can be recovered only by the retransmission timeout.

Thus, if we drop a segment between points *5* and *6* we avoid the buffer overflow at the cost of a single packet drop. It is better to select a packet closer to point *5* to avoid underutilization of the link. A practical implementation of such a policy could define a soft queue limit in the router, for example ten packets. The hard limit can be two-three times larger than the soft limit. When the current queue size reaches the soft limit, a single packet is dropped. When the TCP sender detects a packet loss, it decreases the transmission rate and the buffer overflow is prevented. If the hard queue limit is reached, the router drops all arriving packets. Extending this algorithm to work well for a few concurrent connections requires a counter or a timer-based mechanism to determine when to drop another packet in case the load is not decreased, i.e. a previously dropped packet was not from the most aggressive connection. Some heuristics that favors connections with small packets can be implemented to protect interactive flows. The suggested algorithm

Table 3: Two baseline TCP connections over the drop-tail buffer.

Name	**throughput (min)**	**throughput (max)**	**throughput (avg)**	**throughput (fairness)**
buf_10	549.00	654.00	601.50	99.24
buf_20	428.00	600.00	514.00	97.28
buf_40	456.00	589.00	522.50	98.41

Table 4: Two baseline TCP connections over the RED buffer.

Name	**throughput (min)**	**throughput (max)**	**throughput (avg)**	**throughput (fairness)**
buf_10_maxp0.05_w0.2	448.00	662.00	560.00	96.72
buf_10_maxp0.05_w0.4	465.00	549.00	505.00	99.41
buf_10_maxp0.1_w0.2	453.00	548.00	517.50	99.00
buf_10_maxp0.1_w0.4	467.00	519.00	489.00	99.37
buf_20_maxp0.05_w0.2	492.00	577.00	531.00	99.46
buf_20_maxp0.05_w0.4	507.00	554.00	529.50	99.72
buf_20_maxp0.1_w0.2	485.00	579.00	527.00	99.60
buf_20_maxp0.1_w0.4	504.00	584.00	543.50	99.77
buf_40_maxp0.05_w0.2	503.00	609.00	564.00	99.20
buf_40_maxp0.05_w0.4	496.00	577.00	535.00	99.22
buf_40_maxp0.1_w0.2	491.00	569.00	532.00	99.20
buf_40_maxp0.1_w0.4	524.00	607.00	574.50	99.62

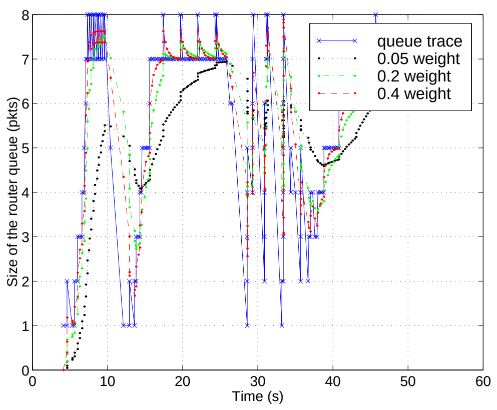


Figure 28: Trace of the router queue and its smoothed average. The baseline TCP over a 7-packet buffer.

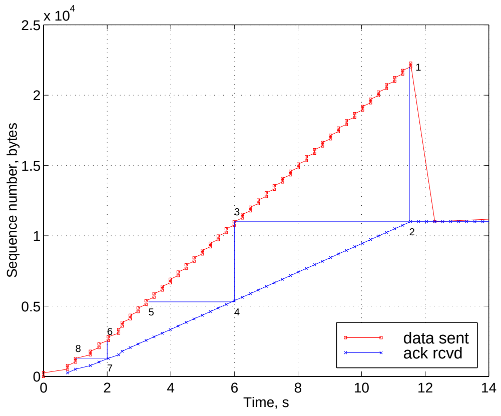


Figure 29: Analysis of the start-up buffer overflow. The baseline TCP over a 20-packet buffer.

is similar to the Dual Threshold Early Packet Discard [CH00].

7.7 Avoiding Multiple Fast Retransmits

In this section we discuss the issue of avoiding multiple fast retransmits in a more general environment than was assumed in the thesis. We have not run a systematic set of tests for this section, but a few specific tests. Fast retransmits are not allowed after a retransmission timeout until all retransmitted data are acknowledged [FH99]. A more careful version of this rule requires that at least one non-retransmitted segment is acknowledged. Most TCP implementations, including Linux and FreeBSD, use the less careful version of the rule, which only requires that all retransmitted segments are acknowledged. We have collected some empirical evidence suggesting that the more careful version should be implemented in all TCPs. The less careful version interprets DUPACKs for the last retransmitted segment as an indication of its loss and triggers the fast retransmit. Thus, the advantage of the less careful version is a quick recovery in case the last retransmitted segment was really lost. The more careful version, on the contrary, argues that in many cases DUPACKs for the last retransmitted are due to the out-of-order delivery of segments before the last retransmitted segment. Assuming the loss of the last retransmitted segment in such a case is a dangerous choice as it can lead to a spurious retransmission of many segments. The more careful version simply ignores DUPACKs for the last retransmitted segment.

We found two scenarios that show the necessity of the more careful version. In the first scenario, the retransmission timeout is caused by a long delay on the link [LRK$^+$99]. The timeout is spurious, since no data was actually lost. Unnecessary retransmitted segments generate a number of DUPACKs for the last received segments and trigger a false retransmit. For example Figure 30 shows a TCP connection in Unix FreeBSD 4.1 after a delay. The connection collapses after a spurious timeout, because of the several spurious retransmissions triggered by DUPACKs and an impatient retransmission timer. The Eifel algorithm is aimed to prevent unnecessary retransmissions after a spurious timeout [LK00].

In the second scenario, several adjacent segments are lost in the middle of

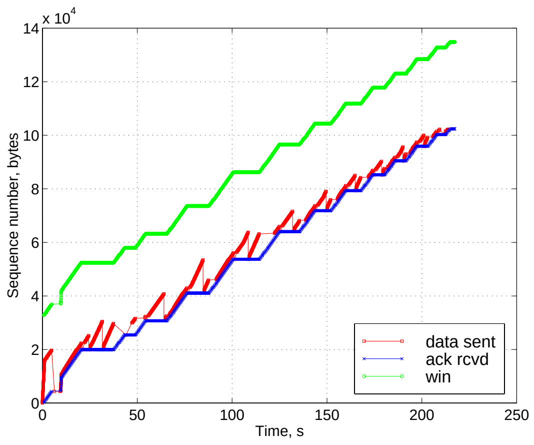


Figure 30: Spurious retransmissions after a delay in FreeBSD Unix.

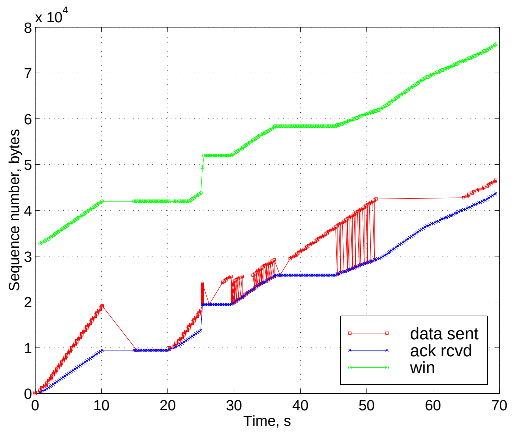


Figure 31: Spurious retransmissions after an error burst in Linux.

the flight. After a retransmission timeout, part of the segments are unnecessarily retransmitted. Figure 31 shows a Linux TCP connection in this situation. DUPACKs trigger a false fast retransmit twice. New Reno makes the situation worse by making more unnecessary retransmissions. Implementing the more careful version of the “bug fix” would prevent the invalid behavior in both scenarios.

8 Conclusion

The wireless environment presents a challenge for an efficient data transport over slow and lossy links. We have performed an experimental evaluation of TCP in an emulated wireless environment. We consider a network model including a lossy wireless link and a last-hop router with a limited-size buffer. We have explored how well the state-of-art TCP perform, identified the key reasons behind the behavior, and determined the effect of different TCP optimizations. We experimented with multiple error rates and buffer sizes over TCP connections with different values of the initial window, the receiver window, with or without SACK and New Reno. The experimental data is obtained with a state-of-art TCP implementation of the Linux operating system and a real-time network emulator Seawind. Our main result is a comparative study and analysis of different TCP optimizations.

Before we obtained what we call a “baseline TCP”, several bugs were fixed in the Linux TCP implementation. We reported some of them and we intend to make the patched kernel publicly available. We found our methodology particularly useful as it allows an easy comparison of different TCP implementations. For example, we ran and compared the same set of tests with different versions of Linux or BSD operating systems.

We use a simple model of a lossy wireless network connected to a fixed host via a router. Packets are lost on the wireless link due to transmission errors (corruption), and in the router due to a buffer overflow (congestion). Our workload model is a downlink unidirectional bulk data transfer of 100 kilobytes. The main problems we address are the start-up buffer overflow, overbuffering, optimal buffer size, fair sharing of resources, the effect of single, random and burst errors.

In the first set of tests we have shown that an unlimited buffer size in the router is not desirable. It creates the overbuffering problem and worsens the recovery from sudden data losses. We have determined a range of the router buffer sizes, of 3-12 packets, giving the optimal performance on an error-free link. We have located and analyzed the cases where a loss of a single packet significantly affects the performance of TCP. The bad effect appears when a packet loss leads to RTO, while the good effect appears in the prevention of the start-up buffer overflow.

Our largest group of tests is with random errors on the link. The error rate was set to 2, 5, and 10 % with a queue limit of 3, 5, 7, 10, and 20 packets. We studied how different TCP optimizations perform in such an environment. We found that the optimal buffer size to be 7-10 packets. Throughput of the baseline TCP is adequate at a 2 % error rate, but is only half of the line rate for a 10 % loss rate. TCP with SACK performed significantly better than other modifications under all conditions, especially at higher loss rates. The increased initial window gives slightly better throughput than the baseline. A small receiver window (2 kilobytes) decreases throughput, especially for smaller buffer sizes. A moderate window (4 kilobytes) is beneficial for larger buffers. Disabling New Reno is helpful at a low error rate and larger buffer sizes; in other cases it is worse. In general, our results are coherent with a previous evaluation of Reno, New Reno, and SACK TCP at the presence of error losses [FF96].

We have also studied what time it takes to transmit the first 15 kilobytes of data in the bulk data connections. In this way we can estimate the performance of a transaction-type traffic. The performance picture is different than for whole connections. The optimal buffer size varies with error rates and TCP optimizations. A limited receiver window and disabled New Reno are quite helpful at the low error loss rate. SACK performs better than other modifications in this case, as well.

We have studied the effect of an error burst on the TCP connection. Typically, little or no data gets through during the burst already at a 20 % packet loss rate. After the bursts ends, the transmission is resumed immediately, except when RTO was backed off several times during the burst. In the later case the connection is idle approximately for half of the burst length after the link quality returns to normal. The likelihood of the RTO back off is increased with the buffer size. This is because most packets sent during the burst are retransmissions, but not the new data. In such a case no valid RTT sample can be collected and RTO is more likely backed off several times. Thus a smaller router buffer size is preferable for a link where error bursts are possible.

We found that RED worsens the performance when only a single TCP connection is present. This is because the moving average of the queue size does not react timely to the start-up buffer overflow, and late packet drops only worsen the recovery. For two concurrent TCP connections, RED improves the throughput and the fairness among the connections, but only for large buffer sizes (20, 40 packets). We have provided the detailed analysis of the start-up buffer overflow and have suggested using a dual threshold drop policy to prevent it. However, its implementation and evaluation is left for future work. A deployment of the Explicit Congestion Notification (ECN) could make RED more attractive in our environment, because ECN avoids congestion-related losses. The implementation of ECN in our emulator and a performance evaluation is left for future work.

We have collected some empirical evidence suggesting that the more careful version of the “bug fix” for preventing multiple fast retransmits should be implemented in all TCPs. In the first scenario, multiple fast retransmits are caused by a long delay on the link and a spurious timeout and in the second scenario, by a loss of a block of segments in the middle of the flight. New Reno adversely affects the performance at the presence of multiple fast retransmits.

References

[ADG$^+$00] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Kruse, S. Ostermann, K. Scott, and J. Semke. Ongoing TCP research related to satellites. IETF RFC 2760, 2000.

[AF99] M. Allman and A. Falk. On the effective evaluation of TCP. *ACM Computer Communication Review*, 5(29), October 1999.

[AFP98] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. IETF RFC 2414, September 1998.

[AGKM98] T. Alanko, A. Gurtov, M. Kojo, and J. Manner. Seawind: Software requirements document. University of Helsinki, Department of Computer Science, September 1998.

[All00] M. Allman. A web server's view of the transport layer. *ACM Computer Communication Review*, 30(5), October 2000.

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

[Bal98] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, Computer Science Division, Univ. of California at Berkeley, Berkeley, CA, August 1998.

[BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance. IETF RFC 1323, May 1992.

[BCC$^+$98] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the Internet. IETF RFC 2309, April 1998.

[BKG$^+$00] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies. IETF Internet draft “draft-ietf-pilc-pep-05.txt”, November 2000. Work in progress.

[BKPV96] B. Bakshi, N. Krishna, D.K. Padhan, and N. Vaidya. A comparison of mechanism for improving performance of TCP over wireless links. In *ACM SIGCOMM*, Stanford, California, August 1996. ACM.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. IETF RFC 1945, May 1996.

[BPSK96] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. In *Proceedings of ACM SIGCOMM '96*, Stanford, CA, August 1996.

[Bra89] R. Braden. Requirements for internet hosts – communication layers. IETF RFC 1122, October 1989.

[BSAK95] H. Balakrishnan, S. Seshan, E. Amir, and Randy H. Katz. Improving performance of TCP/IP over wireless networks. In *Proceedings of the first annual international conference on Mobile computing and networking (MOBICOM 95)*, pages 2–11. ACM, 1995.

[BSK95] H. Balakrishnan, S. Seshan, and R. H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4), December 1995.

[BV97] S. Biaz and N. Vaidya. Using end-to-end statistics to distinguish congestion and corruption losses: A negative result. Technical Report TR97-009, Texas A&M University, August 9, 1997.

[BW97] G. Brasche and B. Walke. Concepts, services and protocols of the new GSM phase 2+ general packet radio service. *IEEE Communications Magazine*, pages 94–104, August 1997.

[CH00] R. Cohen and Y. Hamo. Balanced packet discard for improving TCP performance in ATM networks. Tel Aviv, Israel, March 2000.

[DMK$^+$00] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya. End-to-end performance implications of links with errors. Internet draft “draft-ietf-pilc-error-06.txt”, November 2000. Work in progress.

[DMKM00] S. Dawkins, G. Montenegro, M. Kojo, and V. Magret. End-to-end Performance Implications of Slow Links. IETF Internet draft “draft-ietf-pilc-slow-05.txt”, November 2000. Work in progress.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. IP header compression. IETF RFC 2507, February 1999.

[ECB99] M. Engan, S. Casner, and C. Bormann. IP header compression over PPP. IETF RFC 2509, February 1999.

[FF96] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, July 1996.

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

[FJ93] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FR99] S. Floyd and K. K. Ramakrishnan. A proposal to add explicit congestion notification (ECN) to IP. IETF RFC 2481, January 1999.

[GSM98] GPRS service description. ETSI GSM 03.60, August 1998.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM ’88*, pages 314–329, August 1988.

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

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, 1991.

[KA98] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). IETF RFC 2406, November 1998.

[LK91] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, second edition, 1991.

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

[LRK$^+$99] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A. Joseph. Multilayer 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 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 144–154, New York, May 1–4 1999. ACM Press.

[Lud99] R. Ludwig. A case for flow-adaptive wireless links. Technical Report CSD-99-1053, University of California, Berkeley, 1999.

[Lud00] R. Ludwig. *Eliminating Inefficient Cross-Layer Interactions in Wireless Networking*. PhD thesis, Aachen University of Technology, April 2000.

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

[MM96] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP Congestion Control. In *Proceedings of ACM SIGCOMM ’96*, October 1996.

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

[MP92] M. Mouly and M. Pautet. *The GSM System for Mobile Communications*. Europe Media Duplication S.A., 1992.

[MSMO97] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM Computer Communication Review*, 27(3), July 1997.

[MV97] M. Mehta and N. Vaidya. Delayed duplicate-acknowledgments: A proposal to improve performance of TCP on wireless links. Technical report, Texas A&M University, December 1997.

[Nag84] J. Nagle. Congestion control in IP/TCP internetworks. IETF RFC 896, January 1984.

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

[Pad98] V. N. Padmanabhan. *Addressing the Challenges of Web Data Transport*. PhD thesis, University of California at Berkeley, September 1998.

[Pax97a] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of the ACM SIGCOMM Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM-97)*, volume 27 of *Computer Communication Review*, pages 167–180, Cannes, France, September 14–18 1997. ACM Press.

[Pax97b] V. Paxson. End-to-end internet packet dynamics. In *ACM SIGCOMM '97*, pages 139–152, September 1997. Cannes, France.

[PN98] K. Poduri and K. Nichols. Simulation studies of increased initial TCP window size. IETF RFC 2415, September 1998.

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

[Rah93] M. Rahnema. Overview of the GSM system and protocol architecture. *IEEE Communications Magazine*, 31(4):92–100, April 1993.

[RF99] K. Ramakrishnan and S. Floyd. A proposal to add Explicit Congestion Notification (ECN) to IP. IETF RFC 2481, January 1999.

[Rom88] J. L. Romkey. Nonstandard for transmission of IP datagrams over serial lines: SLIP. IETF RFC 1055, June 1988.

[Sim93] W. Simpson. The point-to-point protocol (PPP). IETF RFC 1548, December 1993.

[SP98] T. Shepard and C. Partridge. When TCP Starts Up With Four Packets Into Only Three Buffers. IETF RFC 2416, September 1998.

[Sta00] W. Stallings. *Data and Computer Communications*. Prentice-Hall, sixth edition, 2000.

[Ste95] W. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, 1995.

[Ste97] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. IETF RFC 2001, January 1997.

[Sti90] R. H. Stine. FYI on a network management tool catalog: Tools for monitoring and debugging TCP/IP internets and interconnected devices. IETF RFC 1147, April 1990.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International, 1996.

[TMW97] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[Tou97] J. Touch. TCP control block interdependence. IETF RFC 2140, April 1997.

[Vai99] N. Vaidya. Tutorial on TCP for wireless and mobile hosts, 1999. Available at: http://cashew.cs.tamu.edu/faculty/vaidya/seminars/tcp-tutorial-aug99.ppt.

[WS95] G. Wright and W. Stevens. *TCP/IP Illustrated, Volume 2; The Implementation*. Addison Wesley, 1995.

A Baseline TCP

This appendix [1] gives the detailed description of the baseline TCP. The TCP implementation we use to execute the tests is based on Linux kernel *2.3.99-pre9*. The situation with Linux kernel was quite inconsistent since the final stable release *2.4* had not yet been published (and still has not). Many new patches came every week. We decided to take the *pre9* version and start working with it because we did not have the time to wait for the final release that we still might have to patch to achieve a TCP that works like we would wish. This section outlines the modifications we have made to the *pre9* kernel. We call this modified TCP version *Baseline TCP*, as it represents the standard behaving TCP that is outlined in [Pos81], [Bra89] and [APS99].

A.1 TCP parameters, options and settings

The TCP standards let the TCP implementations choose some of the parameters and for their own convenience. This section outlines the behavior of *Baseline TCP* in more detail. Because NewReno TCP modification is accepted as a possible fast recovery modification in [APS99], we have included it in the Baseline TCP as it represents the "best current practice".

A.1.1 NewReno TCP modification

When receiving a *partial ack* the TCP sender retransmits the following segment immediately. The question is, should the congestion window be suppressed. It is not clearly stated in [FH99] if a retransmissions is counted as a new transmitted segment which should be taken into account by lowering the `cwnd` by one SMSS. The alternative interpretation is that retransmissions do not count when calculating the new value for `cwnd`. In this case, a new segment may be transmitted in addition to the retransmitted one. We took the latter interpretation and so the Baseline TCP sends a new data segment after receiving a partial acknowledgment.

[1]Written chiefly by Panu T. Kuhlberg

After the TCP sender has received the ACK that acknowledges all segments up to and including the variable *recover*, the fast recovery period is ended. [FH99] gives two possible values for the new value of the cwnd: it can be set to ssthresh or *flightsize + SMSS*. We chose the latter alternative as it reduces the possible burst that may follow after the recovery period. After fast recovery is exited, the cwnd is raised by one SMSS upon every incoming ACK until ssthresh is reached, as in regular *slow start*. However, if the cwnd is exactly four segments, while the third duplicate acknowledgment arrives, the cwnd is *not* reduced after exiting the algorithm upon the new ACK that acknowledges all the four segments. Thus, after the recovery, the cwnd is retained, and congestion avoidance is used to further increase the cwnd. By doing this, the cwnd is not lowered beyond four segments, and the possibility to use fast retransmits is maintained[2].

The NewReno specification [FH99] describes a "bugfix". The question is, how to avoid multiple fast retransmits. Because the data sender remains in fast recovery until all of the data outstanding when fast retransmit was entered has been acknowledged, the problem of multiple fast retransmits can only occur after a retransmission timeout. After RTO, the highest segment sent during the recovery period is recorded to a new variable send_high. If the data sender receives three dupacks that do not cover send_high, fast retransmit is not triggered. Two different variants of exists for the "bugfix", called *Careful* and *Less Careful* [FH99]. The *Less Careful* variant triggers fast retransmit if the ACKs covers the variable send_high and the *Careful* variant enters fast retransmit only if the ACK covers *more* than send_high. Baseline TCP implements *Less Careful* variant of the "bugfix".

A.1.2 Recovery from RTO

Linux kernel was modified to implement "BSD style" RTO recovery[3].

[2]If the cwnd is less than four segments, there are not enough segments in the network that would produce three duplicate acks to trigger a fast retransmit.

[3]We call it BSD style, because Baseline TCP imitates the behavior that was used in the 4.4BSD-Lite version.

A.1.3 RTO calculation

The RTO calculations were not changed from original Linux kernel *2.3.99-pre9*. However, there are some occasions, where the RTO calculation is not accurate. Linux uses the `cwnd` as a parameter, when setting the RTO. Due to Intel Celeron's processor achitecture and undefined functionality in C-programming language conserning right shift operations, a `cwnd` that is multiple of 32, creates very high RTO values. When analyzing the tests, the effect of invalid RTOs were observed, and excluded from the test results.

A.1.4 Delayed acknowledgments

Baseline TCP makes use of *delayed acknowledgments*. The threshold for delaying an `ACK` is 200ms. Using a bandwidth of 9600bits/second, the time between the arrival of two consecutive data segments of size 296 bytes is more than 200ms. Therefore, in most of our tests each data segment is acknowledged separately. When using higher bandwidths, two segments are "quickacked" [4] in the beginning of the connection before the *delayed acknowledgments* are taken into use.

A.1.5 Receiver's advertised window

Due to implementation problems, Linux kernel 2.3.99-pre9 advertised a window of 32Kbytes in maximum even if the socket buffer size was bigger. We have not modified this in any way and therefore, Baseline TCP has a socket buffer of 64Kbytes of which 32 Kbytes is advertised. This feature does not affect the tests and the tests should be interpreted similarly as a "regular" TCP connection with a socket buffer and advertised window of 32 Kbytes. When we run tests with a reduced advertised window, the size announced is the size of the advertised window, not the size of the socket buffer.

[4] A term used to describe that each data segment is acknowledged separately

A.1.6 Disabling control block interdependence

Linux kernel *2.3.99-pre9* used control block interdependence for `ssthresh`, RTT and RTT variance. We disabled this feature and made it a sysctl option.

Table 5 summarizes the algorithms, parameters and their values used in Baseline TCP.

Table 5: Baseline TCP

Item	**Value and explanation**
Slow start	As defined in [APS99]
Congestion Avoidance	As defined in [APS99]
Initial window (IW)	Initial window of 2 segments
NewReno	As defined in [FH99] and Appendix A.1.1
`cwnd` after exiting NewReno	*flightsize* + *SMSS* (Appendix A.1.1)
NewReno "Bugfix"	*Less Careful* variant (Appendix A.1.1)
Delayed `ACK` threshold	200ms (Appendix A.1.4)
Quickacks	Two segments in the beginning of the connection
Advertised window (`rwnd`)	32Kbytes (Appendix A.1.5)
Control Block Interdependence	Disabled by default (Appendix A.1.6)
SACK	SACK option is disabled
Timestamps	timestamps are disabled

A.2 Implementation issues

This section describes the modifications which were made to Linux kernel version 2.3.99-pre9. There are two types of modifications: bug fixes and new TCP options added for the IWTCP project.

A.2.1 New TCP options

Linux provides a mechanism to set kernel-specific options at runtime. We added a set of new TCP options for the purposes of IWTCP. These options can be accessed in `/proc/sys/net/ipv4` in the Linux filesystem.

- **iwtcp_cbi.** Control Block Interdependence for congestion control variables was used in the unmodified Linux. We added this parameter to make Control Block Interdependence a user-selectable option.
- **iwtcp_iw.** This parameter can be used to set the initial congestion window in the beginning of the connection.
- **iwtcp_newreno.** Unmodified Linux used NewReno unconditionally. However, we added this option to follow the regular Reno congestion control policy instead of NewReno.
- **iwtcp_quickacks.** The parameter sets the number of quickacks used to quickly exit the early slow start phase. If the value is set to 0, the regular Linux-behavior is used. (i.e. number of quickacks is rwnd / (2 * MSS)).
- **iwtcp_srwnd_addr.** This parameter is used to activate the shared advertised window for connections originating from specified IP address. The user may specify the least meaningful octet of the peer IP address, for which the connections use shared advertised window. Only the connections from 10.0.0.* address family may be shared. This might not be the correct functionality for the real world (in which case the sharing should be done per device interface), but for the IWTCP purposes we decided to follow the above mentioned logic when deciding whether to share the advertised window or not.

  The TCP receiver calculates the advertised window following the standard procedures, but after the calculation it checks whether the sender's IP address was same than what specified with this parameter. In such case, the receiver calculates the current amount of shared advertised window and sets minimum of the original and shared window to the TCP window advertisement field.

- **iwtcp_srwnd_size.** This parameter specifies the size of the advertised window in bytes to shared among the connections originating from the IP address specified by *iwtcp_srwnd_addr* parameter. For the sharing purposes, our modification keeps track of the number of connections open from the specified source address. When a connection sharing the window receives data, the available space in the window is decreased by the amount of data received. When application reads data from a connection sharing the window, available space in the window is increased by the amount of data read by the application. The size of the window advertisement for each acknowledgement is $min(real\_wnd, avail\_shared/connection\_count)$, where *real_wnd* is the calculated window which would normally be advertised, based on the available buffer size for the socket, *avail_shared* is the amount of shared window space currently available and *connection_count* is number of connections sharing the window.

  If there are new connections opened to share the advertised window, the available window for old connections would decrease, because *connection_count* would increase. However, the advertised window will not be shrinked in such a case, but if a connection was advertising more than its share, no new window space will be advertised when new data arrives. This way the connection's advertised window will gradually decrease when new data arrives.

- **iwtcp_rto_behaviour.** With this parameter the user may choose from three policies of how to act when retransmission timeout occurs. *LINUX (1)* is the unmodified Linux behavior, which allowed new data to be sent while retransmitting the segments from retransmission queue. In particular, duplicate ACKs increased `cwnd`, which made this possible. *HOLDCWND (2)* holds the `cwnd` value as 1 during the transmission from post-RTO retransmission queue. *BSD (3)* is the default used in the IWTCP performance tests, named after BSD because it mimics the BSD style go-back-N behavior when RTO expires. This is achieved by making to alternations to the *LINUX* style: the duplicate ACKs do not increase the `cwnd` when retransmitting from post-RTO retransmission queue, and only the number of originally sent packets is compared to `cwnd` when deciding on whether to send new data. Original Linux compared the sum of original transmissions and retransmissions to the `cwnd`.

A.2.2 Bug fixes

Following fixes were made to the Linux kernel version 2.3.99-pre9 before running the IWTCP performance tests.

- Linux keeps the data received or to be transmitted in data blocks called *sk_buffs*. Each *sk_buff* has over 100 bytes of control data in addition to the segment data. Additionally, Linux allocates a fixed size memory block (usually 1536 bytes) for each IP packet it receives, instead of using the actual MTU in allocation requests.

  The user may limit the amount of memory allocated for each connection by setting socket options for sending and receiving socket buffer size. If the MTU is significantly smaller than the size of the fixed memory block allocated, the socket buffer limits will be reached, even though the amount of actual data received is significantly smaller. However, Linux uses the amount of actual data received for the basis of receiving window advertisements, which causes the receiver to advertise more it is allowed to receive when the MTU size is small. As a result, if the Linux receiver gets more segments than it has allocated space in its buffers, it discards all packets in its current out-of-order queue.

  As this behavior was not acceptable, we modified the TCP code to use actual data size in sending and receiving buffer allocations instead of the fixed predefined size.

- When exiting from fast recovery, unmodified Linux sender set `cwnd` to the value of `ssthresh`. In many situations, this caused a burst of `ssthresh` packets, harmful in environments with limited last-hop buffer space. We fixed this to set $max(packets\_in\_flight, 2)$ to `cwnd` when exiting fast recovery. *packets_in_flight* is the amount of unacknowledged packets in network, including retransmissions.

- The unmodified Linux forced the minimum advertised segment size to be 536 bytes by default (unless changed by sysctl `route/min_adv_mss`). We changed this to be 256 bytes.

- When a burst of segments arrives, Linux does not acknowledge every second segment violating SHOULD in RFC [FH99]. The reason for this may be treating segments of the size less than 536 bytes as a not full sized segments independently on the MSS of the connection.

- The unmodified Linux did not reduce the congestion window when partial ACKs were received during fast recovery, as required in [FH99]. We fixed this to decrease the congestion window by the amount of new data acknowledged with the partial ACK. After decreasing `cwnd`, it is increased by one. As a result, one new segment is transmitted in addition to the first unacknowledged segment next to the one acknowledged with partial ACK.

- Unmodified Linux did not parse TCP option field for incoming segments unless it was about to send some options. This made, for example, SACK unusable. We fixed it to parse option fields for all incoming segments.

- Linux grows the congestion window above the receiver window. This can lead to bursts and should not be done.

- Unmodified Linux did not use an ACK that confirms both a retransmitted and a new segment to collect an RTT sample. It is possible to collect a valid RTT sample in this situation (i.e. there is no contradiction to Karn's algorithm) and it is quite helpful for reseting backed off RTO. We fixed it.

- Linux uses a single variable *seq_high* for two purposes instead of two recommended variables [FH99]. The the variable *recover* should be used for New Reno, while the variable *send_high* should be used for preventing Fast Retransmits after RTO. Mixing those two variables leads to a nonconformant behavior for example when several packets are dropped in the middle of the current FlightSize.

B Measurement Data

Percent values refer to quantiles, for example 50 % is the median. In tests names, buf$x$ gives the limit on the router buffer length on downlink in packets; err$0.y$ gives the error loss rate. The zero value means the unlimited buffer and no error losses correspondingly. For error burst tests, len$z$ gives the length of the burst period in seconds. For RED tests, max$p$ gives the maximum drop probability and w gives the weight of the current queue size in the moving average of the queue size. For tests with two parallel connections, (min) and (max) in the column header refers to the slower and faster connection.

B.1 Optimal Router Buffer Size

Test name	**elapsed time 25%**	**elapsed time 50%**	**elapsed time 75%**	**throughput 50%**	**rexmt pkts 50%**	**dropped pkts 50%**
buf_3	104.61	104.64	104.65	979.00	23.00	23.00
buf_4	105.62	105.63	105.64	969.00	22.00	22.00
buf_5	103.18	103.20	103.21	992.00	19.00	19.00
buf_6	105.25	105.26	105.27	973.00	18.00	18.00
buf_7	105.24	105.26	105.26	973.00	18.00	18.00
buf_8	103.19	103.20	103.21	992.00	16.00	16.00
buf_9	104.99	105.01	105.33	975.00	18.00	18.00
buf_10	103.19	103.20	103.21	992.00	17.00	17.00
buf_11	103.20	103.21	103.21	992.00	17.00	17.00
buf_12	103.20	103.20	103.20	992.00	18.00	18.00
buf_15	104.33	104.35	104.36	981.00	20.00	20.00
buf_20	106.64	106.65	106.67	960.00	24.00	24.00
buf_40	108.08	108.10	108.11	947.00	44.00	44.00
buf_infinite	102.06	102.06	102.07	1003.00	0.00	0.00

B.2 Baseline TCP

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	102.06	102.07	102.08	1003.00	0.00	0.00
buf3_err0.1	200.16	220.38	256.60	465.00	69.00	96.00
buf3_err0.05	139.38	147.08	156.85	696.00	41.00	58.00
buf3_err0.02	113.22	115.76	119.08	885.00	30.00	36.00
buf5_err0.1	173.25	210.69	236.13	486.00	57.00	90.00
buf5_err0.05	130.73	136.83	144.18	748.00	33.00	51.00
buf5_err0.02	109.79	111.84	118.54	916.00	25.00	29.00
buf7_err0.1	167.87	199.54	238.34	513.00	53.00	87.00
buf7_err0.05	127.16	129.57	139.41	790.00	29.00	47.00
buf7_err0.02	107.88	111.69	115.44	917.00	21.00	27.00
buf10_err0.1	169.92	193.33	236.52	530.00	52.00	84.00
buf10_err0.05	129.14	139.54	153.29	734.00	27.00	46.00
buf10_err0.02	106.90	109.04	117.67	939.00	20.00	26.00
buf20_err0.1	171.13	195.65	230.96	523.00	50.00	84.00
buf20_err0.05	124.32	139.33	144.09	735.00	27.00	44.00
buf20_err0.02	104.22	108.10	121.30	947.00	12.00	22.00

B.3 Initial Window of Three Segments

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	101.85	101.85	101.86	1005.00	0.00	0.00
buf3_err0.1	205.43	219.37	271.17	467.00	70.00	96.00
buf3_err0.05	140.34	146.58	151.63	699.00	42.00	59.00
buf3_err0.02	113.51	115.51	121.36	887.00	29.00	35.00
buf5_err0.1	172.72	189.93	250.51	539.00	58.00	90.00
buf5_err0.05	130.62	135.51	141.63	756.00	33.00	51.00
buf5_err0.02	109.17	112.36	118.09	911.00	25.00	30.00
buf7_err0.1	168.48	200.12	240.79	512.00	54.00	88.00
buf7_err0.05	127.60	131.70	138.05	778.00	30.00	48.00
buf7_err0.02	107.54	112.11	115.28	913.00	21.00	27.00
buf10_err0.1	168.70	201.90	237.07	507.00	52.00	85.00
buf10_err0.05	128.91	135.07	142.99	758.00	27.00	46.00
buf10_err0.02	105.66	107.30	112.35	954.00	20.00	26.00
buf20_err0.1	166.94	195.88	244.36	523.00	50.00	84.00
buf20_err0.05	124.96	130.19	138.12	787.00	26.00	45.00
buf20_err0.02	103.96	107.37	117.10	954.00	13.00	22.00

B.4 Initial Window of Four Segments

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	101.85	101.85	101.86	1005.00	0.00	0.00
buf3_err0.1	203.07	229.19	279.66	447.00	66.00	97.00
buf3_err0.05	138.21	150.45	154.93	681.00	41.00	57.00
buf3_err0.02	112.96	115.05	120.33	890.00	28.00	35.00
buf5_err0.1	178.56	204.26	245.33	501.00	57.00	90.00
buf5_err0.05	132.52	137.31	142.66	746.00	33.00	49.00
buf5_err0.02	108.98	111.08	114.66	922.00	24.00	30.00
buf7_err0.1	166.64	202.37	236.02	506.00	54.00	89.00
buf7_err0.05	129.85	132.23	138.47	774.00	31.00	49.00
buf7_err0.02	107.86	111.65	113.00	917.00	23.00	28.00
buf10_err0.1	165.37	202.99	232.31	504.00	52.00	86.00
buf10_err0.05	129.01	134.87	141.86	759.00	27.00	46.00
buf10_err0.02	105.64	108.66	113.55	942.00	21.00	26.00
buf20_err0.1	166.13	199.06	229.85	514.00	50.00	84.00
buf20_err0.05	126.35	132.40	144.68	773.00	26.00	45.00
buf20_err0.02	103.96	107.40	117.98	953.00	14.00	22.00

B.5 Receiver Window of 2048 bytes

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	102.05	102.06	102.07	1003.00	0.00	0.00
buf3_err0.1	194.57	221.59	250.09	462.00	63.00	94.00
buf3_err0.05	137.60	141.77	163.42	722.00	39.00	57.00
buf3_err0.02	114.26	114.75	122.44	892.00	26.00	33.00
buf5_err0.1	184.90	202.98	236.01	504.00	53.00	85.00
buf5_err0.05	132.69	137.76	152.76	743.00	29.00	45.00
buf5_err0.02	113.41	117.45	119.82	872.00	16.00	22.00
buf7_err0.1	172.81	195.75	216.38	523.00	52.00	84.00
buf7_err0.05	131.50	138.69	155.41	738.00	23.00	40.00
buf7_err0.02	107.34	109.06	116.57	939.00	8.00	14.00
buf10_err0.1	172.50	202.04	215.98	507.00	52.00	84.00
buf10_err0.05	131.53	138.76	155.59	738.00	23.00	40.00
buf10_err0.02	107.35	109.07	116.75	939.00	8.00	14.00
buf20_err0.1	172.60	195.94	216.24	523.00	52.00	84.00
buf20_err0.05	131.51	138.75	161.00	738.00	24.00	40.00
buf20_err0.02	107.37	109.07	116.55	939.00	8.00	14.00

B.6 Receiver Window of 3840 bytes

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	102.06	102.06	102.07	1003.00	0.00	0.00
buf3_err0.1	201.52	222.85	265.76	460.00	67.00	96.00
buf3_err0.05	139.25	151.16	156.46	677.00	41.00	58.00
buf3_err0.02	114.01	115.71	121.59	885.00	31.00	35.00
buf5_err0.1	174.32	213.56	236.03	479.00	57.00	89.00
buf5_err0.05	133.86	138.06	144.58	742.00	33.00	52.00
buf5_err0.02	109.62	112.79	115.95	908.00	25.00	29.00
buf7_err0.1	169.69	201.06	225.36	509.00	53.00	87.00
buf7_err0.05	127.99	132.52	140.50	773.00	28.00	46.00
buf7_err0.02	109.00	112.65	118.02	909.00	19.00	25.00
buf10_err0.1	171.04	193.11	222.01	530.00	51.00	84.00
buf10_err0.05	133.84	138.93	143.98	737.00	26.00	44.00
buf10_err0.02	105.82	108.07	115.69	947.00	13.00	20.00
buf20_err0.1	171.25	199.95	221.88	512.00	50.00	84.00
buf20_err0.05	123.60	136.42	148.20	751.00	23.00	40.00
buf20_err0.02	104.71	105.81	110.43	968.00	7.00	14.00

B.7 SACK Enabled

Test name	elapsed time 25%	elapsed time 50%	elapsed time 75%	throughput 50%	rexmt pkts 50%	dropped pkts 50%
buf0_err0	102.05	102.05	102.07	1003.00	0.00	0.00
buf3_err0.1	135.87	146.72	161.20	698.00	61.00	96.00
buf3_err0.05	115.08	117.37	122.15	872.00	45.00	61.00
buf3_err0.02	109.59	111.90	113.23	915.00	43.00	48.00
buf5_err0.1	125.54	137.31	152.45	746.00	50.00	87.00
buf5_err0.05	112.20	116.58	121.07	878.00	36.00	52.00
buf5_err0.02	107.28	109.12	112.71	938.00	37.00	39.00
buf7_err0.1	123.21	133.37	141.91	768.00	45.00	83.00
buf7_err0.05	110.60	112.93	115.82	907.00	29.00	48.00
buf7_err0.02	104.48	105.41	109.53	971.00	28.00	31.00
buf10_err0.1	123.84	133.12	144.63	769.00	45.00	83.00
buf10_err0.05	109.76	111.48	114.22	918.00	25.00	46.00
buf10_err0.02	104.17	105.03	105.57	975.00	24.00	28.00
buf20_err0.1	123.31	133.18	140.93	769.00	44.00	83.00
buf20_err0.05	109.77	112.01	113.67	914.00	23.00	44.00
buf20_err0.02	103.69	104.11	106.22	983.00	12.00	22.00

B.8 New Reno Disabled

Test name	**elapsed time 25%**	**elapsed time 50%**	**elapsed time 75%**	**throughput 50%**	**rexmt pkts 50%**	**dropped pkts 50%**
buf0_err0	102.06	102.07	102.07	1003.00	0.00	0.00
buf3_err0.1	200.94	237.13	268.68	432.00	68.00	100.00
buf3_err0.05	145.29	156.40	161.32	655.00	40.00	59.00
buf3_err0.02	115.43	121.25	126.11	845.00	31.00	35.00
buf5_err0.1	181.91	214.43	244.02	478.00	58.00	90.00
buf5_err0.05	140.53	149.11	166.59	687.00	33.00	51.00
buf5_err0.02	112.18	114.02	122.63	898.00	29.00	28.00
buf7_err0.1	186.71	211.88	246.83	483.00	56.00	89.00
buf7_err0.05	135.39	145.82	158.19	702.00	31.00	47.00
buf7_err0.02	112.15	116.28	120.79	881.00	27.00	26.00
buf10_err0.1	184.39	212.19	250.38	483.00	54.00	86.00
buf10_err0.05	136.75	146.53	152.11	699.00	27.00	46.00
buf10_err0.02	108.59	109.64	112.02	934.00	30.00	25.00
buf20_err0.1	184.26	212.03	243.26	483.00	51.00	84.00
buf20_err0.05	131.85	143.37	151.64	714.00	27.00	44.00
buf20_err0.02	107.49	109.94	115.70	931.00	20.00	22.00

B.9 Burst Error Losses

Test name	**elapsed time 25%**	**elapsed time 50%**	**elapsed time 75%**	**throughput 50%**	**rexmt pkts 50%**	**dropped pkts 50%**
buf3_len10_err0.2	113.16	114.04	117.94	898.00	28.00	30.00
buf3_len10_err0.4	119.10	120.28	121.02	851.00	29.50	31.00
buf3_len20_err0.2	119.95	121.93	129.63	840.00	31.00	33.50
buf3_len20_err0.4	127.75	137.27	138.18	746.00	31.00	32.00
buf10_len10_err0.2	111.83	112.90	116.91	906.50	20.50	24.50
buf10_len10_err0.4	117.12	117.52	119.36	871.50	24.00	25.00
buf10_len20_err0.2	120.22	127.14	132.29	805.00	23.00	24.00
buf10_len20_err0.4	133.89	134.40	136.70	762.00	25.00	26.50

B.10 One Connection Over the RED Buffer

Test name	**elapsed time 25%**	**elapsed time 50%**	**elapsed time 75%**	**throughput 50%**	**rexmt pkts 50%**	**dropped pkts 50%**
buf10_maxp0.05_w0.2	113.14	114.60	119.30	893.00	32.00	37.00
buf10_maxp0.05_w0.4	111.75	112.90	116.75	907.00	30.00	33.00
buf10_maxp0.1_w0.2	114.52	118.18	127.41	866.00	35.00	37.00
buf10_maxp0.1_w0.4	114.44	116.17	120.59	881.00	29.00	34.00
buf20_maxp0.05_w0.2	106.78	108.53	115.57	943.00	22.00	26.00
buf20_maxp0.05_w0.4	104.66	108.03	115.77	948.00	19.00	26.00
buf20_maxp0.1_w0.2	107.67	118.43	126.49	864.00	20.00	27.00
buf20_maxp0.1_w0.4	105.18	109.45	122.65	935.00	18.00	25.00
buf40_maxp0.05_w0.2	104.39	108.08	116.83	947.00	11.00	20.00
buf40_maxp0.05_w0.4	103.93	110.32	114.58	928.00	10.00	17.00
buf40_maxp0.1_w0.2	104.53	110.32	118.66	928.00	12.00	20.00
buf40_maxp0.1_w0.4	104.20	106.36	110.99	963.00	12.00	18.00

B.11 Two Connections Over the Drop-Tail Buffer

Test name	**elapsed time (min) 25%**	**elapsed time (min) 50%**	**elapsed time (min) 75%**	**elapsed time (max) 25%**	**elapsed time (max) 50%**	**elapsed time (max) 75%**	**rexmt pkts (min) 50%**	**rexmt pkts (max) 50%**
buf_10	78.24	78.25	78.27	93.23	93.24	93.25	15.00	19.00
buf_11	82.53	82.55	82.56	93.23	93.24	93.26	16.00	21.00
buf_12	87.52	87.52	87.53	93.43	93.43	93.44	16.00	22.00
buf_15	83.48	83.49	83.50	103.86	103.87	103.88	3.00	29.00
buf_20	85.33	85.34	85.37	119.55	119.57	119.60	2.00	27.00
buf_3	93.36	99.25	100.06	100.05	102.50	109.92	28.50	36.50
buf_4	69.80	71.76	75.86	104.97	106.65	108.90	14.00	28.00
buf_40	86.96	86.97	86.98	112.20	112.21	112.23	3.00	42.00
buf_5	93.76	94.12	94.43	94.96	97.54	98.98	17.50	23.00
buf_6	94.01	94.51	95.63	97.96	101.73	103.93	12.00	31.00
buf_7	92.35	93.40	93.53	105.08	105.19	105.43	11.00	37.00
buf_8	87.12	87.89	95.49	95.79	96.27	97.37	14.50	34.50
buf_9	91.37	93.53	96.24	99.67	102.50	110.83	37.00	44.00
buf_infinite	52.36	52.37	52.37	93.37	93.38	93.38	0.00	5.00

B.12 Two Connections Over the RED Buffer

Test name	**elapsed time (min) 25%**	**elapsed time (min) 50%**	**elapsed time (min) 75%**	**elapsed time (max) 25%**	**elapsed time (max) 50%**	**elapsed time (max) 75%**	**rexmt pkts (min) 50%**	**rexmt pkts (max) 50%**
buf10_maxp0.05_w0.2	64.87	77.30	91.94	111.20	114.16	121.86	19.00	33.00
buf10_maxp0.05_w0.4	86.70	93.21	98.25	105.55	110.01	113.52	17.00	29.00
buf10_maxp0.1_w0.2	81.84	93.45	99.00	107.27	113.11	115.79	21.00	32.00
buf10_maxp0.1_w0.4	80.38	98.62	101.64	106.11	109.57	119.05	21.00	30.00
buf20_maxp0.05_w0.2	78.61	88.75	95.27	100.16	104.01	110.33	9.00	25.00
buf20_maxp0.05_w0.4	85.37	92.35	94.67	96.62	101.03	105.10	11.00	19.00
buf20_maxp0.1_w0.2	84.92	88.48	96.27	100.31	105.54	115.62	13.00	25.00
buf20_maxp0.1_w0.4	80.72	87.73	94.96	95.79	101.51	106.38	10.00	18.00
buf40_maxp0.05_w0.2	72.29	84.11	92.92	97.18	101.87	103.47	6.00	9.00
buf40_maxp0.05_w0.4	79.80	88.69	94.90	96.03	103.19	109.60	6.00	12.00
buf40_maxp0.1_w0.2	84.43	90.06	94.29	96.76	104.22	111.02	7.00	11.00
buf40_maxp0.1_w0.4	74.65	84.36	93.60	95.57	97.79	103.47	7.00	11.00