Performance analysis of TCP enhancements for congested reliable wireless links

Pasi Sarolahti

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

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

Tekijä — Författare — Author

Pasi Sarolahti

Työn nimi — Arbetets titel — Title

Performance analysis of TCP enhancements for congested reliable wireless links

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level M.Sc. Thesis	Aika — Datum — Month and year December 2000	Sivumäärä — Sidoantal — Number of pages 80 p. + Appx.

Tiivistelmä — Referat — Abstract

In this thesis we present a performance analysis of using TCP over a slow wireless link with a persistently reliable link layer protocol. A last-hop router with a limited buffer space is located between the wireless link and the fixed network. A real-time software emulator is used for modelling the wireless link and the last-hop router. By using the emulator we can have the control over the link characteristics and use a real protocol stack of the operating system in the performance tests. We present the related work and the main problems observed with the reliable wireless links, which are spurious retransmission timeouts and the congestion at the last-hop router. We compare the performance of the selected baseline TCP implementation with the SACK TCP and a TCP with increased initial congestion window size. Additionally, the performance implications of using a RED queue management algorithm at the last-hop router is studied. A detailed analysis is presented to explain the benefits and the problems of the baseline TCP and the different TCP enhancements. We introduce a mechanism for limiting the number of outstanding packets in the network by defining an upper limit for the TCP advertised window size and sharing the available advertised window space between the parallel TCP connections over the wireless link. By using the shared advertised window promising performance improvements are achieved in our environment.

Computing Reviews Classification:
C.2.2 (Network protocols)
C.4 (Performance of Systems)

Avainsanat — Nyckelord — Keywords
Wireless communication, mobile computing, performance, TCP, congestion

Säilytyspaikka — Förvaringsställe — Where deposited
Library of the Dept. of Computer Science, Report C–2001–

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Internet has traditionally been a combination of fixed local area subnetworks. These networks provide transfer rates of several Megabits per second and reliable, low-delay transport. As the Internet has become more popular, the subnetworks have operated relatively well, but the congested routers have become critical points. Therefore, congestion control was adopted as the main principle in the design of the *Transmission Control Protocol (TCP)* [Pos81, APS99] which is used by most of the networking applications in the Internet. TCP is an end-to-end protocol, offering the connected host a simple virtual connection to the destination. The TCP endpoints have very limited mechanisms to get information about the connection path between the endpoints. Therefore TCP makes some assumptions about the connection path. One such assumption is that if packets are dropped along the connection path, it is because a congested router could not process them. In this situation the TCP sender slows down its sending rate to help the congested router recover.

Lately the number of wireless hosts connected to the Internet has been increasing. With the new packet radio techniques, such as *General Packet Radio Service (GPRS)* [BW97, CG97], the wireless Internet hosts are about to become even more popular. Wireless links have very different behaviour than fixed, broadcast-based local area networks. Wireless links are slower than the fixed ones and they are much more vulnerable to the conditions in the natural environment, causing higher and more variable bit error rates.

In this thesis we concentrate on TCP transmission over slow wireless links with highly variable delays. In addition to the slow transmission rate and high latency typical to the slow wireless links, we inspect the length of the packet queue in the last-hop router buffer between the wireless link and the fixed network. Routers need buffers to tolerate and smooth the bursts of data arriving from the network. If the router buffer size is exceeded, packets are dropped and the TCP sender is required to slow down its transmission rate. Because the fixed network delivers the packets at a higher rate to the last-hop router than what the router can transmit to the slow wireless link, the router is prone to congestion and usually several packets are dropped because the router buffer overflows.

The poor performance of TCP over wireless links is a well known fact [CI94, BPSK96, KRL$^+$97]. Although many of the performance studies inspect the problems caused by the packet corruption on the link, it is not the concern in our environment, because we assume the link layer to offer reliable service to the higher protocol layers. Instead, the reliable link layer causes the packet delays to be variable and unpredictable. Because TCP uses timers as the basis of doing retransmissions, excessive delay may trigger an unnecessary retransmission, which has negative implications on the TCP performance.

A large amount of research has been done on improving the TCP behaviour over wireless links. Suggested improvements for TCP can be roughly divided in two categories. An efficient way to improve the TCP performance is to split the TCP connection in a wireless and a wired part and separate the two sections of the connection path by a *proxy* [BKG$^+$00]. However, these solutions might violate the end-to-end principles of TCP and require changes and TCP-level intervention in the middle of the connection path. Therefore they are not suggested for general use.

Another approach to improve the TCP performance is to modify the behaviour of TCP end points without splitting the connection. In this thesis we concentrate on this kind of improvements and inspect their influence on the TCP performance when transmitting data over a wireless link. In addition to the selected TCP enhancements, we inspect the performance implications of using active queue management at the last-hop router.

We present the wireless environment and the characteristics we are assuming for the wireless link in Section 2. In Section 3 we discuss the known problems of using TCP over slow wireless links, especially in the presence of congestion and variable delays. We introduce some of the suggested TCP improvements to overcome the problems presented. The test arrangements for our performance tests are described in Section 4. We describe how we use a real-time software emulator for modelling the wireless link and the last-hop router and discuss the validity of the model we are using. We select a set of test cases to be tested with the baseline TCP implementation we have chosen and analyse the results of the test runs in Section 5. After pointing out the problems with the baseline TCP, we present the performance analysis for the selected TCP enhancements in Section 6. Finally, we discuss the benefits and the problems caused by the TCP enhancements and conclude our work in Section 7.

In addition to the main text, we provide a number of appendeces[1] to give some additional information for the reader. In the appendeces we provide the test results of all performance tests run, description of the state-of-the-art TCP congestion control algorithms, description of the baseline TCP implementation we are using, description of the TCP enhancements we analyse in our work, and description of the Seawind software emulator we use to model the wireless link and the last-hop router.

[1]The appendeces are jointly written by Andrei Gurtov, Panu Kuhlberg, and Pasi Sarolahti.

2 Wireless Communication Environment

In this section we describe the wireless communication environment we are assuming in this thesis. There are two factors in the wireless environment which need to be considered separately: the architecture of the wireless access network and the properties of the wireless link.

2.1 Network architecture

Figure 1 shows the high level view of the network components which are involved when a mobile host is communicating with a host in a fixed network on the Internet. The connection path consists of a last-hop wireless link and a number of wired links through various fixed subnetworks. This kind of organisation is used on wireless *local area networks* (W-LANs) and wireless *wide area networks* (W-WANs, in which category e.g. the widely used *Global System for Mobile communications (GSM)* [MP92] belongs). *Internet Engineering Task Force (IETF)* has considered the problems of the Internet communication in this kind of environment and propsed a number of solutions for the problems [MDK$^+$00]. The PILC working group in IETF has discussed the impact of different link characteristics further [DKMK00, DMK$^+$00, KFT$^+$00].

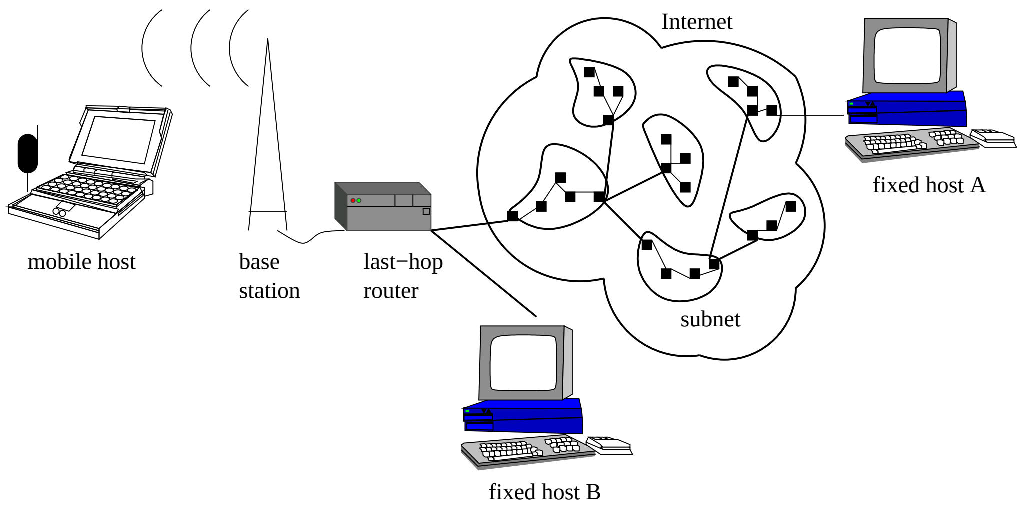

Figure 1: Architectural elements in a system with wireless hosts

The mobile host communicates with the *base station* through the air interface using a wireless radio link, which is prone to data corruption. The base station is connected to a last-hop router, which is the last network element before the wireless link having buffers to store the network packets. The actual functionality carried out in the last-hop router may vary according to the wireless technology used. For example, in the GPRS system the last-hop router corresponds to the *Serving GPRS Support Node (SGSN)* [BW97].

The last-hop router is connected to the traditional fixed internetwork in which the fixed host communicating with the mobile host is located. Properties and problems in the fixed network are drastically different from the ones of the wireless link. Fixed network usually transfers data at much higher speeds than the wireless link. Unlike with wireless links, data corruption very seldomly occur in the fixed network. However, because the traffic characteristics are highly unpredictable in the internetwork, estimating round-trip time and the other connection parameters might not be easy. Packet losses are also a frequently seen phenomenon because of congestion at the routers. Because the fixed host is usually a *WWW* or *ftp* server, most of the data is transmitted downlink, from the fixed server to the mobile client.

The sender may also be located in the same network with the last-hop router. This is a common case, for example, if an organisation provides a *World-Wide Web (WWW) proxy* for the users. As can be seen in Figure 1, the communication path between fixed sender B and the mobile host is much simplier and involves less intermediate hops than the path between sender A and the mobile host. Because of the short communication path, the traffic from sender B arrives at the router at a constant rate, but the traffic from sender A is likely to be much more unpredictable in terms of delay and packet arrival rate. In the measurements made in this thesis we assume that the sender is located in the same network with the last-hop router.

Because the fixed network provides higher transfer rates than the wireless link, the last-hop router between the fixed and the wireless parts of the connection is a serious bottleneck in the connection path. We assume that the last-hop router has a limited, separate buffer space for each user. Hence when packets are received from the fixed network faster than the router can forward them, the router buffer will overflow before long, causing the packets to be dropped at the router. This is an extreme form of congestion, therefore resembling the traditional problem of the internetworks. However, because the congestion is very severe and it occurs in a known location quite predictably, mechanisms for preventing this phenomenon are worth inspecting.

In addition to the setup presented in Figure 1, there are different architectures yielding different properties for wireless communication. For example, communicating over *satellite links* can be done in various different ways [ADG$^+$00]. Usually satellite communication involves high, asymmetric bandwidth and large round-trip times over an unreliable link [AGS99], for which the problems are different than in the wireless environment described in this section. Thus, this thesis is restricted to slow terrestial wireless links having a high-level architecture as presented in Figure 1.

2.2 Properties of the wireless link

The wireless link between the mobile host and the base station has usually very different characteristics than a conventional wired link. The available bandwidth on the wireless link is usually much lower than on a wired link. For example, in W-WAN systems such as GSM the typical bandwidth is 9600 bps. With enhanced encoding and channel allocation techniques the bandwidth acquired by the mobile user is somewhat higher, being around 20 - 40 kbps. It is also possible for the bandwidth to be asymmetric, giving more capacity downlink than uplink. Additionally, transmission through a wireless link takes more time to propagate to the other end of the link than with a traditional fixed link. This causes more delay (100-300 ms in W-WAN systems) in the transmission of the packets. The packets are usually fragmented into smaller frames at the link layer (e.g. the RLP frame size in GSM is 240 bits [MP92, Section 3.3.3]).

Wireless communication is usually prone to transmission errors because of highly variable conditions in the natural environment. The transmission errors are bursts of distorted bits causing the transmitted data frame to be useless. In W-WAN systems the average bit error rate can be as high as 10^{-3} after forward error correction. However, by using link layer ARQ retransmissions the bit error rate perceived by the higher protocol layers can be improved to a magnitude of 10^{-8} [KRL+97]. The transmission errors are usually perceived as packet losses by the upper protocol layers, hence the sender is required to retransmit the lost packets.

One of the most important, although optional, tasks of the link layer protocol is to provide reliable transfer over the link to the upper protocol layers. Therefore a link layer protocol might provide some retransmission mechanisms (e.g. *Radio Link Protocol (RLP)* in GSM [MP92, Section 3.3.3]), possibly maintaining the ordering of packets when delivering them to the upper protocol layers. In such a case the errors at the airway link do not necessarily cause packet drops as perceived by the higher protocol layers, but appear as noticeably long delays, because none of the packets received at the other end of the link is delivered to the upper layers before the missing packet is succesfully transmitted. Some higher layer protocols may not perform well, if the ordering of packets is not maintained. Out-of-order packets may cause unnecessary retransmissions with some protocols, which needlessly wastes the scarce link capacity.

One of the design details in a link layer protocol is the *persistency* of the link layer retransmissions, i.e. how long the link layer sender should try to retransmit a frame before it gives up and reports an error to the higher protocol layers. The selected persistency has effect when the mobile host moves to a location with bad radio link coverage. In this case several successively transmitted frames are lost and the wireless connection between the mobile host and the base station is completely unusable for some period of time. In this thesis we assume that the link layer protocol provides peristent retransmissions, maintaining the order of the packets.

The new packet-radio technologies such as GPRS are likely to cause the perceived delays to be highly variable for other reasons than because of link layer retransmissions [LK00]. For example, if the lower level protocols have support for *Quality of Service* by providing a priority-based scheduling between the different data flows, the flows with lower priority are likely to experience highly variable delays, when flows of higher priority have reserved the link.

3 TCP over Slow Wireless Links

Although the *Transmission Control Protocol (TCP)* [Pos81] has been widely used and tested on the Internet for several years, its behaviour on connection paths with slow and unreliable links is not ideal. Slow link and transmission errors are likely to cause notable implications on TCP performance. In this section we briefly describe some of the known phenomena of TCP when it is used over slow wireless links and briefly present some suggestions offered for improving the performance of TCP in these environments.

3.1 TCP basics

The reader should be familiar with the basics of the TCP behaviour [Com95, Ste95] in order to review the performance results presented in this thesis. In particular, the behaviour of the present state-of-the-art congestion control algorithms [APS99] need to be understood in detail. In this section we briefly describe the important TCP features related to loss recovery and congestion control. Appendix B gives a more detailed description of the algorithms.

A TCP sender must be conservative in the amount of data transmitted to the network to prevent the network getting congested. Therefore, a *congestion window* was introduced for the TCP sender to control the number of TCP segments outstanding in the network [Jac88]. The TCP sender starts with initial congestion window size of one or two TCP segments and increases its transmission rate exponentially until the congestion window size reaches a *slow start threshold (ssthresh)* value. In other words, during the slow start the sender increases the congestion window size by one segment and transmits two new segments each time a new acknowledgement arrives. This is called *slow start* algorithm. The slow start threshold is initialised to have a arbitrary high value, and it is set to half of the number of segments outstanding in the network if the TCP sender notices a packet loss. After the congestion window size reaches the slow start threshold, the sender continues with the *congestion avoidance* algorithm during which it increases the congestion window size by one segment only once per round-trip time.

Traditionally the TCP receiver only acknowledges the highest segment it has successfully received in order, and if a segment is lost, there is no way to indicate it to the sender. In the early versions of TCP the only way to recover from a segment loss was to wait for *retransmission timeout (RTO)* to expire. The length of the RTO is determined from the measured *round-trip time (RTT)* and the variance of the recent RTT measurements [Jac88, PA00]. The receiver cannot acknowledge new data in case of segment loss and hence the sender is not allowed to send new data. Therefore, before the retransmission timeout expires, there is usually an idle period during which the sender does not transmit any data, possibly causing the communication path to be underutilised.

A mechanism for passing negative acknowledgements was introduced with the *Reno* congestion control algorithm, which later became basis for the standards track RFC[2] [APS99]. Reno congestion control allows the sender to retransmit a segment when it receives three successive *duplicate acknowledgements (ACKs)*. Duplicate ACK is an acknowledgement which acknowledges exactly the same octet as the previous segment. The receiver generates a duplicate ACK when it receives an out-of-order segment. The algorithm for sending a retransmission on the third duplicate ACK is called *fast retransmit*. Additionally, the congestion control specification also improves the TCP performance by allowing the sender to transmit a new segment each time an additional duplicate ACK arrives after the fast retransmit, if the TCP congestion window and receive window allow that. This is based on the judgement that each duplicate ACK is an indication of a packet that has arrived at the receiver, meaning that the network load was reduced by one packet. This is called the *packet conservation rule* [Jac88], which requires that the number of outstanding segments in the network is maintained during the fast recovery. However, as the packet loss is possibly a notification of congestion, the sender has to wait for the number of outstanding segments to be halved before new segments can be transmitted. The algorithm following the fast retransmit is called *fast recovery*, and it is finished after the first acknowledgement for new data arrives to the sender.

It was observed, that Reno does not work well if there are several packets dropped during a single round-trip [Hoe96], and therefore *NewReno* [FH99] was introduced as an experimental RFC. In contrast to Reno, NewReno does not exit the fast recovery algorithm if a *partial acknowledgement* is received. Partial acknowledgement is an ACK which acknowledges new data, but not all of the data that was sent before the sender received the third duplicate ACK. When the sender receives a partial ACK, it retransmits the first unacknowledged segment and then continues the fast recovery as defined for the Reno algorithm. It is not clearly specified whether the sender should transmit new data with the retransmitted segment when it receives the partial ACK. Literally the specification would allow it, but on the other hand, it would then violate the packet conservation rule described above and gradually increase the amount of data in the network. The TCP implementation we use in the performance tests transmits a new segment with the retransmitted segment.

[2] *Request For Comments (RFC)* are specifications and recommendations used by *Internet Engineering Task Force (IETF)* to define the functionality of Internet protocols and applications.

3.2 Effect of variable delays

If the link layer provides persistent retransmissions to recover from corrupted frames, the TCP receiver perceives only an additional delay when there are transmission errors on the link. In this thesis we assume that no TCP segments are lost because of data corruption, and the only anomaly caused by the link layer are the additional delays because of link layer retransmissions.

The variable delays may trigger unnecessary retransmission timeouts, because the TCP round-trip time calculation algorithm determining the RTO value has no way to predict such delays. We call a retransmission timeout triggered by an excessive delay a *spurious retransmission timeout.* Because the purpose of RTOs is to help the sender to recover from segment losses which are not recovered by fast retransmit and fast recovery algorithms, the sender reacts to the timeout by retransmitting the first unacknowledged segment. Additionally, the TCP sender resets the congestion window to one segment and restarts the transmission from the retransmitted segment. From this point on the TCP sender proceeds with slow start, increasing the congestion window and retransmitting two packets for each incoming new ACK until the congestion window size reaches the slow start threshold. The slow start threshold was set to half of the number of outstanding packets when the retransmission timeout occured. This behaviour is also followed by the TCP implementation of widely used 4.4BSD Unix [WS95] and it is called *go-back-N* behaviour.

If the TCP retransmission timeout expired because of the excessive delay caused by link layer retransmissions, several segments may get unnecessarily retransmitted, although no segments were dropped. The unnecessary retransmissions are shown in Figure 2. The TCP sender retransmits the segments, because it is not able to distinguish the acknowledgements from the original transmissions from the acknowledgements of the retransmissions. This problem is called *retransmission ambiguity* [KP87, LK00]. If no packet losses occured before the retransmission timeout, the number of unnecessary retransmitted segments is equal to the congestion window size at the moment when the retransmission timeout occured. During the unnecessary retransmissions no useful data is transferred over the link.

The unnecessary retransmissions may cause further problems for the communicating TCP endpoints, because the unnecessary retransmissions made by the TCP sender are considered to be out-of-order segments at the receiver. Some suggestions have been given for the TCP implementation to minimize the bad effects caused by a spurious retransmission timeout [FH99], but the exact actions to be taken after a spurious retransmission timeout depends on the TCP implementation. We discuss these features in more detail in

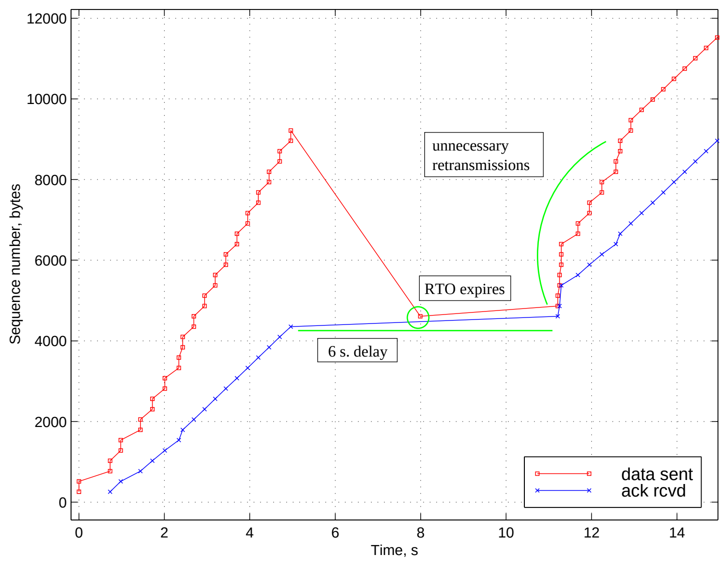

Figure 2: A spurious timeout triggered by an additional delay of 6 seconds.

the baseline TCP analysis in Section 5.

3.3 Effect of congestion at the last-hop router

Because the packets transmitted from the fixed network often arrive to the last-hop router at a higher rate than the router can forward them to the wireless link, the router must queue the packets until it can forward them. Queueing causes additional delay on packets, affecting the measured round-trip times. The problem of queueing delays is also noticed in the studies on using TCP over a GSM link [KRL$^+$97, LRK$^+$99]. The queueing delay is problematic because it inflates the calculated RTO based on round-trip time estimations [LRK$^+$99]. This is harmful if a lost segment should be retransmitted and fast retransmit is unusable for some reason.

The last-hop router has a limited amount of buffer space, which will eventually overflow, because the TCP congestion control algorithms allow the number of packets in the network to increase gradually (during congestion avoidance) or rapidly (during slow start). This causes some of the packets to be dropped at the router, and the TCP sender will eventually notice this from the duplicate ACKs generated by the receiver. However, the duplicate ACKs arrive to the sender roughly one round-trip time later from the buffer overflow, that is, after the segments transmitted before the buffer overflow are acknowledged. The larger the TCP congestion window, the more new acknowledgements are still arriving at the sender after the buffer overflow. If the sender is in slow start, it transmits two new segments for each acknowledgement arriving before the duplicate ACKs. Some of these segments are likely to be dropped also, because they arrive to a full queue. This behaviour is called *slow start overshooting* [MM96] and it is illustrated in Figure 3.

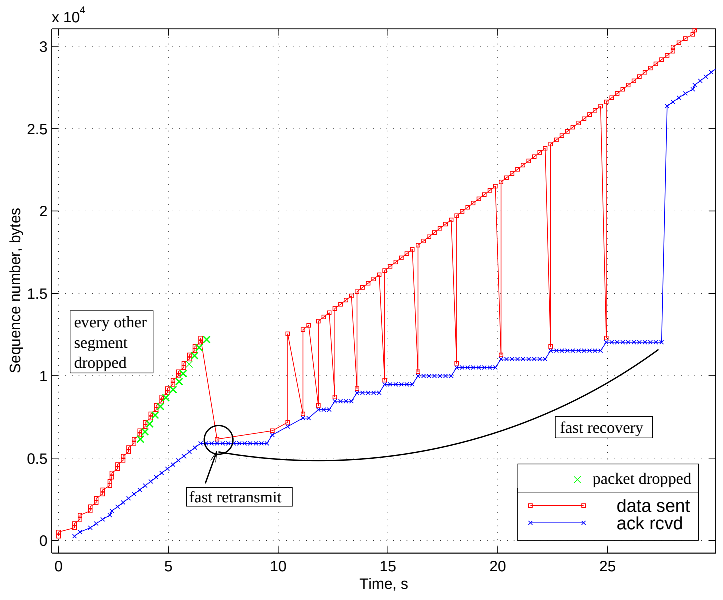

Figure 3: Packet losses caused by slow start overshooting. A router buffer size of 7 packets is used.

Slow start overshooting causes the most serious problems at the beginning of a TCP connection. This is because the slow start threshold is initialised to an arbitrary high value and the sender stays in slow start until the first packet drop is signalled back to the sender. When the duplicate ACKs caused by the first packet drop arrive to the sender, the congestion window size has increased to be notably larger than what the actual network capacity is, hence several packets have been dropped at the last-hop router. The larger the last-hop router buffer, the more packets are in flight at the moment the first router buffer overflow occurs and the more severe the first slow start overshooting is. After the initial buffer overflow, slow start threshold gets assigned a proper value, causing the sending rate to be slowed down by the congestion avoidance algorithm. Therefore the following buffer overflows are less harmful and usually cause only a single retransmission.

Choosing the optimal buffer size in the last-hop router (or in the mobile host) is one of the main problems in our work. The problem of having too large a link interface buffer is also identified in related research [LRK+99], in which a link interface buffer size of 8 kilobytes (approx. 16 packets, as they use MTU of 512 bytes) was selected. It has been suggested that the buffer size should be large enough to hold link *bandwidth * delay*[3] product's worth of data for each TCP connection on the link [KFT+00]. In our performance tests we have a link bandwidth of 9600 bps and the round-trip delay on the link is approximately 700 ms, for which at least 840 bytes (i.e. 2 full-sized packets of 296 bytes) of buffer space would need to be allocated according to the suggestion. However, this buffer size would not be large enough for the fast retransmit algorithm to work, as it requires three duplicate ACKs for triggering a retransmission.

3.4 Effect of parallel TCP connections on the bottleneck link

When multiple parallel TCP connections are involved, the queueing delay is expected to affect the behaviour of TCP even more than what it does for a single connection. It has been reported that multiple parallel connections start to interfere with each other, because the queueing delay increases and becomes more unpredictable [KRL+97]. If one connection gets its packets in the router, the other connections suffer from an unexpectedly long queueing delay.

It has been reported that the packets of multiple TCP connections running in parallel over a bottleneck link are separated into groups of segments belonging to a particular TCP connection [SZC90]. Each of the TCP senders transmits one window's worth of packets to the network, which arrive at the receiving host successively. No packets belonging to the other TCP connections are mixed between the burst of packets sent by one of the TCP senders. Each of the senders transmits a burst of packets, because the acknowledgements for the previous window transmitted arrive successively to the sender. After sending a window of data, the sender will have to wait for a longer time, because the packets sent by the other TCP senders are in the router queue, delaying the new packets transmitted. This behaviour is a result of making the new packet transmissions clocked by the incoming acknowledgements. The incoming acknowledgements for a particular TCP connection arrive successively, because every time the TCP sender increases the congestion window, it transmits two or more segments successively on the link. This behaviour is called *separation of packets*.

The number of dropped packets due to buffer overflows is expected to be higher when several parallel TCP connections are in transit, because the last-hop router is likely to be more congested. It has been observed that when the router buffer is full, packets

[3] *bandwidth * delay* product determines the amount of data that can be outstanding on the link at once. Bandwidth is the link bandwidth and delay is the round-trip delay for the packets on the link.

from several parallel connections are usually dropped [FJ93, p. 4]. This causes all the TCP connections that suffered from packet loss to decrease their congestion window at the same time, which causes a sudden decrease in the amount of traffic outstanding in the network until the congestion window for different TCP connections is increased again. This phenomenon is called *global synchronization*.

If one of the parallel TCP connections ceases transmitting, the other TCP connections will use the available bandwidth on the link and fill the router buffer with their packets. When the idle TCP sender starts transmitting again, it is possible that there is not enough space in the router buffer, and the router drops the packet transmitted by the TCP sender. If the TCP sender did not transmit enough data to trigger fast retransmit, it will have to wait for the retransmission timeout, after which it may retransmit the dropped segment. By doing this, the TCP sender allows the other connections to continue using the available capacity of the link and the router buffer. When the retransmission timer expires, the router buffer may still be occupied by the packets of other the TCP connections and the retransmission triggered by the retransmission timeout is dropped at the router. Therefore, the TCP sender will have to wait for another retransmission timeout in order to retransmit the segment again. If the TCP sender is unfortunate, the above presented events may be repeated several times. This is called *lock-out* behaviour [BCC$^+$98], which causes substantial unfairness among the parallel TCP connections.

Another possible reason for unfairness is the nature of TCP congestion control. When a packet is dropped in one of the two parallel TCP connections, the sender deflates its congestion window, i.e. reduces its transmission rate. The other connection which did not have its packet dropped will benefit of this and get a larger share of the available bandwidth. If the packet loss can not be recovered using fast retransmit, the sender has to wait for the retransmission timeout. This allows the link capacity to be used by the competing TCP connections and might lead to the lock-out behaviour described above.

3.5 Suggested improvements

Balakrishnan makes a distinction between three different approaches on improving the TCP performance, when there are corruption-related losses in addition to congestion-related losses [BPSK96]. Implications of link-layer protocols, split-connection approaches and end-to-end solutions for improving the performance are compared in the study. The best performance was gained by using a reliable link layer protocol. The study suggested that end-to-end solutions should be favored, because by using proper TCP enhancements a relatively good performance could be achieved without having support from the intermediate network nodes.

A reliable link layer protocol suggested above is used in the wireless networking environment assumed in our performance analysis. We consider the link layer to do retransmissions persistently, which is also recommended in a recent study [Lud00]. If the link layer does a small number of retransmissions and then gives up as a result of link outage, the TCP sender will have to retransmit the segment in the same way it would have done without any link layer error recovery method. Additionally, when the maximum number of link layer retransmissions has been made, the link layer protocol may reset its state and flush the transmission and receive buffers, causing several packets to be dropped. This is done, for example, in the GSM RLP protocol. Therefore, use of persistent link layer retransmissions may result in better performance at the transport layer. However, high persistency on link layer retransmissions may cause competing error recovery between the transport layer and link layer, i.e. spurious TCP retransmission timeouts because of the additional delays caused by the link layer retransmissions. Moreover, when a packet is retransmitted at the link layer, the following packets of all TCP connections using the wireless link are blocked, thus the additional delay is experienced in all TCP connections using the wireless link.

The Eifel algorithm was introduced to avoid the retransmission ambiguity caused by spurious timeouts [LK00]. It uses the TCP timestamp option [BBJ92] to distinguish the retransmissions from the original transmissions. If the TCP sender notices from the timestamp information that a spurious retransmission has been made, it starts sending new data instead of retransmissions. While this is a promising improvement, the use of TCP timestamps inflicts a disadvantage: TCP timestamp adds 10 bytes of overhead in every TCP segment, timestamps being not supported by the traditional header compression methods [DMKM00].

Improving the retransmission timer in order to reduce the number of spurious timeouts has also been a subject of research. Eifel Retransmission Timer [LS00] tries to achieve more accurate RTO calculations in occasions where the traditional RTO calculation algorithm fails. Traditional RTO calculation depends heavily on the variance of RTT measurements, and the RTO value increases significantly if round-trip times are suddenly decreased for some reason. The Eifel retransmission timer avoids inflation of RTO in cases when RTT drops. Additionally, the Eifel timer uses different weight constants from the traditional algorithm for RTT and RTO calculation in order to balance the effects of sudden changes in RTT. The Eifel retransmission timer collaborates with the Eifel algorithm mentioned above and adapts the timer to be less aggressive if the sender has done a spurious retransmission.

The effect of link layer error recovery has also been inspected by generating an analytical model of the environment in which there are link layer retransmission and bounded buffer size [CLM99]. Although using an analytical model to analyse TCP performance is difficult due to various approximations that have to be made, it was concluded after the analysis that link layer retransmissions do improve performance. The same result has also been achieved by the experimental research described above.

There have been a number of suggestions to use a *proxy* in an intermediate node between the wireless link and the fixed network [BKG+00]. A proxy can be transparent to the TCP endpoints [BSK95], in which case the retransmissions made by the proxy do not break the end-to-end semantics of TCP. Some proxies may break the end-to-end semantics of TCP, which causes problems concerning the TCP reliability. For example, if a proxy maintains state for different TCP connections, a sudden failure on the proxy can make the TCP connections maintained by it unusable. If end-to-end IP level security (IPsec) [KA98] is used, the proxy at intermediate node may not be able to parse the TCP headers and operate correctly. In the most extreme case, a proxy may totally split the TCP connection in two separate TCP connections, as done in the I-TCP protocol [BB95]. However, use of the proxies have been reported to improve the throughput between connection endpoints [KRL+97]. Because of the various disadvantages of using a proxy, an end-to-end solution for improving the TCP performance is usually favored over using a proxy.

Authors of NetReno identified too small a congestion window to be a potential reason for retransmission timeouts when a packet drop occurs, for example, due to congestion [LK98]. They suggested that the sender should send new packets for the first two duplicate ACKs in order to avoid the problem. IETF has also suggested this approach [ABF01].

Alternative queue management algorithms for the bottleneck routers have been suggested to avoid the congestion-related problems, such as the lock-out behaviour and the resulting unfairness. Authors of the *Random Early Detection (RED)* [FJ93] algorithm suggest that by dropping the packet randomly at the router before the router queue is exhausted instead of the traditional tail-drop algorithm would alleviate the congestion-related problems. We test the effect of RED on the TCP performance, and describe the algorithm in more detail in Appendix D. *Explicit Congestion Notification (ECN)* [FR99] has been suggested to be used with RED. If congested, an ECN capable router makes a congestion mark in the IP header instead of dropping the packet. When the receiver gets the marked packet, it provides feedback to sender in the TCP header, which then slows down its transmission rate. As a result, the number of packet losses at the router should be reduced.

4 Test Arrangements

In this section we describe the test methodology and objectives we are trying to attain with our performance measurements. We describe the environment and arrangements used in the performance measurements, such as the exact test parameters, workload models and the TCP features to be used. Finally, we discuss the validity and possible inaccuracies in the selected test model.

4.1 Methodology

The purpose of this study is to evaluate the performance of the current state-of-the-art TCP in a wireless environment with properties described in Section 2 and compare the effect of selected TCP enhancements on the TCP performance. We do the performance tests for the test cases defined in this section and in case we observe suboptimal performance, we identify the reason for the unwanted behaviour and try to estimate alternatives which could be used to avoid the problematic behaviour.

We use a real-time software emulator for modelling the target environment. Using software to emulate the environment has some benefits over measuring the performance in the natural environment. With the emulator we can control the behaviour and properties of the link and the emulated network architecture under TCP. For example, if we observe an interesting TCP behaviour with a certain pattern of events on the link, exactly the same pattern can be repeated with an alternative TCP implementation, which makes it easier to compare the different TCP implementations. With the software emulator we may also run predefined test configurations automatically, which makes the execution of the performance measurements more convenient.

The weakness of using emulation for the performance tests is that the method is vulnerable to the errors caused by inaccurate approximations made in the emulation model. Attaining the exact model of the target environment is usually very difficult and therefore some approximations have to be made in the emulator. Before deciding on which details are irrelevant to be modelled in the emulation, we must identify the objectives we want to achieve with the measurements, and evaluate whether the approximations have significant effect on the test results. We describe the possible inaccuracies in our software emulator in Section 4.5, and discuss their significance on the measurement results.

We have chosen a relatively modern TCP implementation used in Linux kernel version 2.3.99-pre9 as the baseline to be used in our performance measurements, after modifying it to conform the TCP specifications. Our baseline TCP implements NewReno congestion control [FH99] which is an experimental TCP fast recovery algorithm suggested by IETF, but which is likely to become (if not already) widely deployed in the TCP implementations of various operating systems. It has been suggested that the implementation selected as the baseline should be likely to be the state-of-the-art implementation of the near future [AF99], and we believe this is the case with our baseline. The details of our baseline TCP implementation and the bug fixes we made to it are described in Appendix C.

Certain topics are of special interest in our performance tests. We are interested in the performance implications of the excessive delays at the link layer. For example, by studying delays which are long enough to cause a spurious retransmission timeout, we can see what happens at the TCP level when the link layer error recovery and the transport layer error recovery interact. By observing the effect of short delay which does not cause spurious TCP retransmission timeout, we can see the effect of successful link layer error recovery on the TCP performance.

There are various reasons (see Section 2) for the excessive delays to occur on the link. Therefore we want to inspect the effect of excessive delays in general, although the delay behaviour we are modelling resembles the one caused by the link layer retransmissions. Throughout the analysis we use term *additional delay* when referring to the excessive erratic link layer delay, which might have occured, for example, because of the link layer retransmissions.

Another topic of interest is the effect of congestion at the last-hop router when the bandwidth of the outgoing link is low and there are highly variable delays on the link. Although the congestion effects can be observed already with one connection transferring bulk data, we also study the behaviour of several parallel TCP connections to see whether the interactions between the connections described in Section 3.4 are present when the outgoing link from the last-hop router is significantly slower than the incoming link. We analyse the congestion effects in the presence of excessive delays with small, medium and large router buffer sizes to see how the selected router buffer size affects on the TCP behaviour.

We evaluate the significance of certain implementation details which are left unspecified in the TCP specifications. In particular, the NewReno fast recovery algorithm has different variants which may have effect on the TCP performance when packet losses and excessive delays are present. Finally, we evaluate the effect of various TCP enhancements on the TCP behaviour when there are additional delays present on the slow link.

4.2 Modelling the target environment

The target environment consists of a mobile host which communicates with a fixed host via a point-to-point wireless link and a last-hop router. We ignore the problems of the global Internet in our tests and concentrate on the behaviour of the TCP on the wireless link combined with a LAN. The bandwidth of the wireless link is 9600 bps. The sender attached to the LAN is assumed to be able to provide traffic significantly faster than what the wireless link can carry. We do not specify the exact rate at which the data arrives from the LAN to the last-hop router, but consider it to be several magnitudes faster than the bandwidth of the wireless link. The bandwidth of the LAN is 10 Megabits, of which a small fraction is used for traffic not destined to the mobile host.

The upper section of Figure 4 illustrates the elements which are emulated from the target environment. The last-hop router has a limited buffer size to hold 3 - 20 packets transmitted downlink. The router buffer is dedicated to the wireless link, hence the traffic destined to other mobile hosts does not have any effect on the performance of the TCP flows under study. This is how ISPs usually configure their buffers in their last-hop routers for the dial-up connections [LRK$^+$99]. The uplink router buffer is not subject to congestion, because it is not likely to be filled as the packets arriving in it from the slow wireless link are immediately transmitted to the LAN having a significantly higher bandwidth than the wireless link. Uplink and downlink packet flows are handled independently and they do not interfere with each other in any way.

In addition to the router buffer, there are *link send buffer (LSB)* and *link receive buffer (LRB)* at the both sides of the wireless link. The link buffers are needed to provide a reliable service to the higher protocol layers using link layer retransmissions over the unreliable link. When a link layer frame is lost because of the data corruption, the receiving end of the link has to hold the following frames in the link receive buffer to deliver them in order to the higher protocol layers. After the missing frame has been received, all data in the link receive buffer is delivered to the higher protocol layers at once. After the sending end of the link has received an acknowledgement of the successful transfer, it can release the data from the link send buffer.

The bottom section of Figure 4 illustrates the emulation environment and its association to the target environment. We use Seawind software [AGKM98] to emulate the wireless link, link buffers and the router buffer shown in Figure 4. Seawind is a soft real-time network emulator and it is described in more detail in Appendix E. The performance tests were made using three Celeron class Pentium machines located on a private LAN with a bandwidth of 10 Mbps. The network hosts were running the Linux operating system.

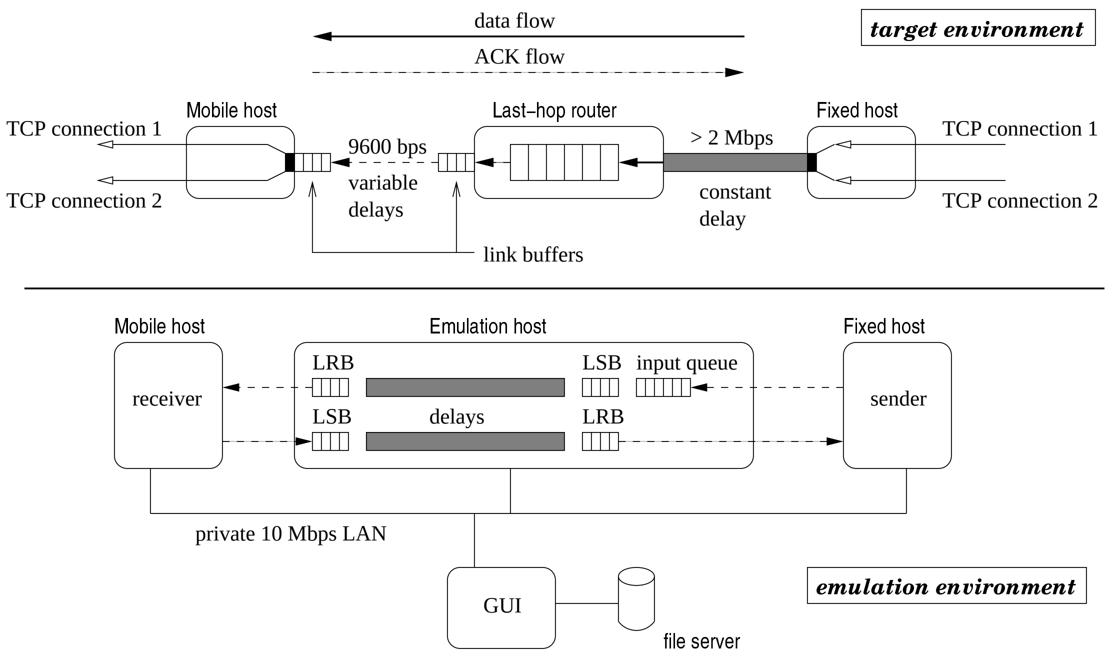

Figure 4: Network elements to be emulated and the arrangement of the emulation.

One of the hosts is the *emulation host*, taking care of the emulation of the wireless link and the last-hop router. The *Mobile host* and the *Fixed host* are the endpoints communicating with each other using one or several TCP connections. The TCP enhancements that are used in the performance tests need to be implemented only in the endpoint hosts, and the host running the emulator can run any operating system supported by Seawind. Finally, the user controls the emulation by using a *Graphical User Interface (GUI)* located in one of the hosts in the private LAN. This host gathers the log information generated by Seawind.

The slow link is emulated by issuing appropriate *transmission delay* and *propagation delay* for each packet. Link layer retransmissions are emulated by causing an *error delay* for a packet. During the error delay no packets are released from the link receive buffer. After the error delay all packets in the link receive buffer are forwarded to the receiving host and they are removed from the link receive buffer and the link send buffer maintained by Seawind.

The last-hop router queue is emulated using the *input queue* maintained by Seawind. The traditional first-in-first-out scheduling is used on the input queue. If there is no room for incoming packet in the input queue, Seawind discards the packet, similarly to what the emulated last-hop router would do. Additionally, RED active queue management algorithm described in Section 3.5 is available for handling the input queue. If there is room in the link send buffer, packets are moved to it from the head of the input queue.

Packets with a specified IP address are transparently forwarded to Seawind at the mobile and the fixed host. The workload packets are encapsulated within another TCP connection between the corresponding endpoint and the emulation host. The endpoint hosts can be used as if the connection was really made over a link with the emulated parameters. In this way we can see the effects of the slow link in the entire communications protocol stack all the way from the drivers level to the TCP level.

Because Seawind is designed to operate in real-time, the underlying hosts and network need to be free from other CPU-intensive applications and heavy network traffic to ensure as accurate results as possible. Therefore, during the performance tests there were as few applications as possible on the hosts attached to the private LAN to ensure that most of the CPU and network capacity were available for Seawind. Seawind itself uses CPU and I/O requests sparingly (e.g. the 1-minute average CPU load level was below 5 % during the performance tests). The extra transmission delay of no more than 1 - 2 ms caused by the 10 Mbps LAN did not have a meaningful effect on the results, because Seawind slows the TCP transmission rate down to 9600 bps. Additionally, there was a small amount of Seawind control traffic in the network, but we estimated the network to be very lightly loaded, allowing the Seawind packets to be transmitted without interference.

The core of Seawind is the *Simulation Process (SP)* located in the emulation host, which handles every packet generated by the connection endpoints, delaying and possibly dropping the packets. SP generates a log of its own in which it reports any delay caused for a packet, queue status, packet drops and other meaningful events with an exact timestamp. Analysing these logs together with the trace of packets from both connection endpoints we can see exactly how each packet was affected during the test.

The accuracy of the different delays used to model the link characteristics is important in order to obtain valid results. Because the Linux operating system does not guarantee the accuracy of the timer events, the actual delay length is of special interest. SP prints a warning message in the log if the requested delay was exceeded by more than 10 ms. In such a case we evaluated the effect of the excessive delay on the TCP performance separately and ignored the test result if the delay inaccuracy had effect on the TCP behaviour. Additionally, we use median in our statistic reports to hide the effects of outliers caused by the rarely occuring emulation errors. On an average the actual delay length differed from the requested length by less than 2 ms.

We use *Point-to-Point Protocol (PPP)* [Sim94] as the link layer protocol between the mobile and remote host. The IP packets carried in PPP frames have *maximum transfer unit (MTU)* of 296 bytes, which yields the TCP *Maximum Segment Size (MSS)*[4] of 256 bytes. The overhead caused by PPP is 7 bytes for each packet, thus the size of the packets transmitted through Seawind is 303 bytes. We selected a small packet size, because small packets are recommended with slow links in order to guarantee reasonable response times for interactive applications [MDK+00]. PPP was configured to not compress the protocol headers or the payload in any way. The compression mechanisms supported by PPP are disabled primarily because evaluating the link layer compression mechanisms is out of the scope of this study, although compression is an important topic when a slow link is used. Furthermore, it is known that some compression mechanisms (e.g. VJ header compression [Jac90]) are counterproductive if there are errors on the link [KFT+00].

Table 1 summarises the default Seawind parameters used for our performance tests. A detailed list with a description of the Seawind parameters is given in Appendix E (page 123). Summarising the table, SP emulates a static link bandwidth of 9600 bps with a link propagation delay of 200 ms. A limited input queue is given for downlink traffic at the SP, emulating the limited buffers at the last-hop router. The queue size limit is defined as number of packets, but the amount of bytes in the input queue is not limited in any way. The default policy for handling the packets which do not fit into the queue is '*tail-drop*', meaning that if the queue is full when a packet arrives, the packet is dropped, otherwise it is appended to the queue. For some tests the RED queue management algorithm is also used (see Section 4.4). In addition to the input queue, SP emulates the link send buffer and the link receive buffer for holding four full-sized packets. Although the link buffer size limit is given in bytes, partial packets are not inserted in the link buffers. We selected the link buffer size appropriately to ensure that exactly four 303-byte packets always fit into the link buffers.

We do not emulate link layer retransmissions for the uplink traffic. We decided to do this simplification in order to limit the number of needed test cases and to make the analysis of random tests easier. We believe that the effect of an additional delay for an ACK packet does not differ meaningfully from the effect of delaying the corresponding data packet, when inspecting the TCP behaviour. Some preliminary tests were made to verify this.

[4] *Maximum Segment Size (MSS)* is the maximum payload size transfered in a single TCP segment.

Table 1: Seawind parameters used in the emulation by default.

Parameter Name	Downlink value	Uplink value
input queue length	3, 7 or 20	-
queue overflow handling	drop	-
queue drop policy	tail-drop	-
link send buffer size	1220 bytes (4 pkts)	1220 bytes (4 pkts)
link receive buffer size	1220 bytes (4 pkts)	1220 bytes (4 pkts)
transmission rate	9600 bps	9600 bps
propagation delay	200 ms	200 ms
error handling	delay	-
error rate type	per packet	-
error probability	*varying*	-
error delay function	*varying*	-
NPA: mtu	296 bytes	296 bytes
NPA: mru	296 bytes	296 bytes

4.3 Metrics

As “performance” can be considered a relatively subjective concept, we need to specify which metrics we want to measure. In this subsection we introduce the metrics we are interested in our performance evaluation and justify why the particular metric is interesting to us.

- **Throughput** is probably the most important metric when measuring the performance of bulk data communication, because it is usually the most significant factor affecting the end-user satisfaction. For the purposes of throughput, we measure the *elapsed time* for transferring the data at the sender. When we are measuring the elapsed time for entire connection, it is the time between sending the first TCP *SYN* segment and receiving the acknowledgement for the *FIN* segment[5]. When measuring partial connections, the elapsed time is the time elapsed from the first packet transmitted to the reception of the acknowledgement for the specified amount of data. To define the throughput, we divide the amount of data transferred by the elapsed time. We also use the term *acknowledged throughput* interchangeably to emphasize that the throughput measurements depend on the arrival rate of the acknowledgements.

[5]TCP segments with a *SYN* flag are used in the three-way handshake that opens the TCP connection. TCP segments with a *FIN* flag are used to indicate that the end-point is willing to close the connection. Both end-points are required to transmit and acknowledge a *FIN* segment.

- **Fairness** between the connections can only be measured for tests with multiple connections. To measure the level of fairness we use Jain's fairness index [Jai91, p. 36], which is directly derived from throughputs of the parallel connections. Fairness is not very well defined, and its importance to the end user or the network administrator is not very obvious. Nevertheless, we use this metric for supporting our analysis.

- **Number of retransmissions** strongly affects the throughput achieved with TCP and is therefore an important metric. If we can reduce the number of retransmissions, we will most likely improve the throughput. In our environment retransmission can be triggered either by packet drop at the last-hop router or by a retransmission timeout caused by excessive delay, in which case there are usually unnecessary retransmissions made. Therefore, an additional analysis is needed to support this metric.

- **Number of dropped packets** directly affects the number of retransmissions, but this metric gives the additional information needed to derive the number of unnecessary retransmissions. Because packet drops occur only because of last-hop router buffer overflow in our environment, this metric measures the severity of the congestion at the last-hop router.

In addition to the metrics described above there are some interesting metrics we are not giving as much attention, but refer to them during the analysis. These metrics are described below.

- **Number of RTOs** is difficult to automatically measure with the current tools we have. However, the retransmissions triggered by retransmission timeout are relatively easy to observe from the TCP traces manually. Before RTO is triggered, there is usually an idle period substantially decreasing the link usage, hence the number of RTOs is a significant factor affecting the throughput. Moreover, retransmission timeouts are usually followed by unnecessary retransmissions.

- **Number of unnecessary retransmissions** is the difference between the number of dropped packets and the number of retransmissions. The number of unnecessary retransmissions is the measure of the inefficiency of the used protocol. If we would have an optimal transport layer protocol, there would be no unnecessary retransmissions.

- **Router queue length** can be used to measure the congestion at the last-hop router. The larger the queue length, the more there is queueing delay for the packets at the router. Therefore it is desirable to keep the queue length as short as possible.

> Because the queue length varies depending on the decisions made by the TCP sender, we follow the progression of the queue length over time to see the effects of the TCP sender behaviour on the queue length.

In the basic case with a single TCP connection we measure the results for entire connections transmitting 100 KB, starting from the first SYN segment transmitted to the last acknowledgement received. However, when evaluating the results of tests with multiple parallel TCP connections, choosing the measurement point is not as obvious as with a single TCP connection. If one of the parallel TCP connections has acquired a noticeably larger share of the link bandwidth and hence finished earlier than the other connections, the other connections can have the bandwidth reserved by the fastest connection and achieve a considerably improved throughput for the rest of the test run. Although this is the realistic behaviour, for example when making multiple `ftp` transfers in parallel, it is usually difficult to detect the unfairness at the beginning of the parallel connections.

For the reason described above, we also measure the results for the first 10 kilobytes of the parallel TCP connections. At that point all the TCP connections are still running and the unfairness can be observed in the difference of the throughput measurements. This can also be justified by the observation that 80 % of the transfers from a WWW server tend to be smaller than 10 KB [All00]. In a WWW-type transmission the throughput of partial connections is interesting, because the user can see the progress of the transfer on his desktop.

Finally, we have to decide on which of the parallel TCP connections we are making the comparisons and conclusions made, i.e. which of the connections is the most interesting one. Presenting the results from all of the parallel connections is not interesting nor reasonable in most of the test cases. Furthermore, the throughputs of the parallel TCP connections are often strongly interacting with each other. Usually, a significantly improved throughput on one of the connections is only an indication of problems (e.g. retransmission timeout) on the other connections. Although we inspect all of the connections run in parallel, we use the throughput of the connection finished last as the basis of the conclusions made, because that is the point at which the test run is actually finished[6].

The test results are obtained by inspecting the transmitted and received IP packets using the `tcpdump` tool [JLM97] in both endpoint hosts and then combined with Seawind

[6] *Total time* was also considered as the basis of the analysis. It is the time difference between the first segment of the first connection sent and the arrival of the last segment of the last connection. However, because the SYN packets of the connections are transmitted successively by the parallel senders, the elapsed time of the connection finished last differs only by tens of milliseconds from the actual total time, and can therefore be used as well.

logs. We primarily use the measurements made in the sending (i.e. fixed) host in our analysis. The `tcpdump` output is collected in log files and analysed using tools such as `tracelook` [PS98], `matlab` [Mat97] and `tcptrace` [Ost] in addition to our own scripts.

4.4 Selected test cases

We perform two kinds of analysis of TCP behaviour. Firstly, we measure the above-mentioned metrics of a selected set of test cases and summarise the observed performance based on the measured metrics. Secondly, we select the most interesting test cases and analyse them in detail on the packet level to explain the reasons for the observed performance. We use the baseline TCP implementation described in Appendix C as the basis of our analysis, identifying the problems causing bad performance. Then we compare the effects of the TCP enhancements listed below to see how they affect on the performance, and whether the enhancements help in avoiding the problems observed with the baseline TCP.

The different test cases are defined by considering three different factors: the workload model to be used in the test, the emulated network and link model, and the TCP implementation we are using.

Workload

For the purposes of this study we have selected two workload models to be used in the performance tests. The workload models are described below.

Single unidirectional bulk transfer The simplest workload model is to transfer bulk data only in one direction using a single TCP connection. In this workload model we transmit 100 KB of bulk data from the fixed end to the mobile end. We believe this kind of unidirectional transfers to be rather usual in normal mobile networking, because typically a mobile host is used as a client machine which downloads web pages or other files from a fixed end server. The bulk data is generated using the `ttcp` tool. Bulk data transfers are widely used in performance measurements, because they are simple and easy to analyse. For the same reason we primarily use bulk data transfers in our performance tests.

Parallel unidirectional bulk transfers Having several parallel TCP connections is another common case in normal Internet traffic. For example, user may have active ftp transfer while he is browsing web pages. We run a number of tests using this workload model, because having multiple parallel TCP connections transmitting over a bottleneck link has remarkable implications on the TCP behaviour, as described in Section 3.4. We use 2 and 4 parallel TCP connections transmitting bulk data from the fixed host to the mobile host using multiple `ttcp` processes running in parallel. Each of the connections are used to transmit 50 KB of data.

SP configuration

As described above, our objective is to inspect the performance implications of the additional delays on a network path containing a slow link and a last-hop router with a limited buffer size. Additional delays are defined in two different ways: we determine a simple random distribution for the delay frequency to see how TCP behaves in a natural, randomly behaving environment. For the tests with random additional delays we use a common random distribution generated in advance for all test cases with all TCP implementations to make the results more comparable.

In addition to the random delay tests we make tests with selected delay scenarios to inspect certain details of the TCP behaviour more closely. In these tests we explicitly define the location and the length of the additional delay based on the preliminary test runs we have made earlier. We place an additional delay to occur just before the buffer overflows the first time, during the fast recovery algorithm following the first packet drop, and after roughly three quarters of the workload have been transmitted, when the TCP behaviour has reached a steady state. Additionally, we test different delay lengths. The length of the additional delay is related to the length of the retransmission timer, and is chosen to be either shorter than what is required for the RTO to expire, long enough to trigger an RTO and long enough to trigger an additional backed off RTO in addition to the first RTO.

The different delay scenarios are combined with three different last-hop router buffer sizes: a *small* buffer has space for 3 packets, a *medium* buffer has space for 7 packets and a *large* buffer has space for 20 packets. One of the objectives in our performance tests is to evaluate the effect of the buffer size on the performance of wireless communication with excessive delays and show the problems observed with the different buffer sizes. Our primary interest is not to find the exactly the optimal buffer size for different delay scenarios, hence we have selected only three different buffer sizes to limit the number of test runs.

Table 2 summarises the delay and buffer configurations we are using in our performance tests. The table shows the combinations of buffer sizes and the different delay scenarios to be tested. For each delay scenario the delay location as a packet count from the beginning of the connection and the length of the delay in seconds are shown. Additionally, a brief description of the test case is given. Each test case is repeated 20 times and the *median* of the measurements is used in analysis to eliminate the effects caused by the possible erratic behaviour in the test environment. When testing the different TCP enhancements we select only the most interesting cases, based on the results of tests made with the baseline TCP.

Table 2: Scenarios used in the performance tests.

Buffer (pkts)	**Delay**	**Description**
3	-	Small buffers, no additional delays
	12th, 1 s.	before the 1st overflow, less than RTO
	12th, 6 s.	before the 1st overflow, more than RTO
	30th, 6 s.	during the 1st fast recovery, more than RTO
	30th & 50th, 6 s.	during the first two fast recoveries, more than RTO
	rand. 0.01, 6 s.	random delays of 6 seconds, packet delay probability 0.01
7	-	Medium buffers, no additional delays
	20th, 1.5 s.	before the 1st overflow, less than RTO
	20th, 6 s.	before the 1st overflow, more than RTO
	40th, 10 s.	during the 1st fast recovery, more than RTO
	300th, 6 s.	after 3/4 of the data has been transmitted, more than RTO
	20th, 15 s.	before the 1st overflow, two successive RTOs
	40th & 80th, 15 s.	during first two fast recoveries, more than RTO
	rand. 0.01, 6 s.	random delays of 6 seconds, packet delay probability 0.01
20	-	Large buffers, no additional delays
	40th, 4 s.	before the 1st overflow, less than RTO
	40th, 6 s.	before the 1st overflow, more than RTO
	100th, 12 s.	during the 1st fast recovery, more than RTO
	300th, 6 s.	after 3/4 of the data has been transmitted, more than RTO
	40th, 30 s.	before the 1st overflow, two successive RTOs
	rand. 0.01, 6 s.	random delays of 6 seconds, packet delay probability 0.01

Selected TCP enhancements

Based on the assumed problems TCP has in the environment we are modelling, we test a number of enhancements to see whether they improve the performance when compared to the baseline TCP. Some of the enhancements are a modification of the TCP protocol behaviour, whereas others require only adjusting a TCP configuration parameter. Additionally, we measure the effect of using an active queue management algorithm at the last-hop router instead of the traditional drop-tail queueing. The modifications are briefly described below, and a more complete description of the modifications can be found in Appendix D.

- **Selective acknowledgements (SACK)** [MMFR96] TCP option makes it possible for the receiver to give more information in the acknowledgements about which segments have been received when there are more than one segment missing between successfully received ones. With SACK it is possible to perform retransmissions more efficiently than with the baseline TCP, because SACK allows several retransmissions to be made within a round-trip time. Because the slow start overshooting at the beginning of the connection potentially causes several packets to be dropped during one round-trip time (see Section 3.3), this TCP enhancement is expected to improve the performance in our target environment. We use the SACK implementation provided in the Linux kernel, which uses the *Forward acknowledgment (FACK)* algorithm [MM96] on deciding when to transmit segments.

  Because SACK is currently widely deployed on the Internet [AF99], including it in the baseline TCP could be well justified. However, use of the FACK algorithm and the other details of the Linux implementation of SACK may cause performance implications which we need to inspect separately.

- **Initial congestion window of four segments** has been suggested for enhancing the TCP performance especially with short connections over high-delay links [AFP98]. We measure the effects of this enhancement with short connections, but no meaningful improvement is expected for the throughput in longer bulk transfers. It has been reported that this improvement reduces the response times experienced with interactive protocols, for example in web browsing [PN98]. Furthermore, a related measurement report states that when a single TCP connection is used, initial congestion window of 4 * MSS does not cause notable negative effects even when the last-hop buffer can hold only 3 packets [SP98]. Although the environment assumed in the report was similar to what we are modelling, we believe that having multiple parallel TCP connections will result in different behaviour because of increased load at the last-hop router.

- **Shared advertised window limitation** is assumed to reduce the number of packet losses caused by overflow of the last-hop router buffer. We define an upper limit for the size of the TCP window advertisement transmitted with acknowledgements in order to reduce the number of outstanding TCP segments the sender is allowed to inject in the network [DMKM00]. By default Linux uses a socket buffer size of 32 KB, which evidently is too large, as the *bandwidth * delay* product over the connection path we are using is less than one kilobyte. The *bandwidth * delay* calculation is based on the bandwidth of 1200 bytes per second and the estimated round-trip delay, which consists of data packet delay of 450 ms, added with ACK delay of 250 ms. The delay estimations include the propagation delay and the packet transmission delay. If the number of outstanding segments is limited to be smaller than the bottleneck link capacity augmented with the last-hop router buffer size[7], we believe the packet losses due to an overflowing router buffer can be avoided. Additionally, limiting the number of outstanding packets results in smaller queueing delays. This is appealing optimisation, because it requires changes only at the receiving end. Usually it is easier to modify the mobile system, which we assume to be the receiving end, than a legacy server from which the data is sent.

This enhancement is expected to improve the performance of multiple parallel transfers, because the available advertised window space is shared at the receiving host equally among the TCP connections transferred over the wireless link. This means that even if the number of parallel TCP connections increases, the total number of outstanding segments is maintained the same as if there was only one connection open. We expect to avoid the lock-out behaviour and the resulting unfairness with this enhancement. We are not aware of other similar suggestions in the related studies[8], although automatic adjustment of the socket buffer size for high-speed links has been suggested [SMM98]. Sharing the receiver window can be considered an application of the *Control Block Interdependence* [Tou97], which has been previously used for sharing the RTT estimates and the slow start threshold between the different TCP connections.

- **Random Early Detection (RED)** [FJ93, BCC$^+$98] does not require modifications at the communicating endpoints, but is a stochastic active queue management algorithm employed at the router. When using a drop-tail router with multiple parallel TCP connections, the lock-out problem is likely to occur when the router becomes congested. By using the RED algorithm at the router the lock-out problem should be avoided, because RED drops the packets randomly before the router buffer becomes full. This should improve fairness among the parallel TCP connections. RED also reduces the average queue length at the router and makes the queueing delays shorter.

[7]We use the term *pipe buffering capacity* to refer to the maximum number of packets that can be outstanding in the network at once.

[8]The idea of advertised window sharing was suggested by Markku Kojo.

We call the techniques presented above *TCP enhancements* in the analysis, regardless of the fact, that the RED algorithm could also be used with other transport protocols than TCP.

4.5 Discussion of the model

Before proceeding to the test analysis, we briefly discuss the implications of the different approximations we have made in our network model. Additionally, we discuss some of the common pitfalls made in performance analysis and give a justification of why we believe that our test results are relevant for analysing TCP behaviour over the target environment we described.

A common mistake is to draw the wrong conclusions about the TCP performance based on tests made using an irrelevant workload [AF99]. By using a single TCP connection we may easily inspect the theoretical TCP behaviour, but the TCP behaviour in the actual network with competing traffic is likely to be significantly different. Therefore it is important to perform tests with competing traffic in addition to the tests with a single TCP connection. Although an accurate model of the competing traffic in the Internet is very difficult to come up with [PF97], our environment is easier to model, because we only inspect traffic originating from the local network. Moreover, because the bottleneck link from the last-hop router is isolated from the traffic of other mobile hosts, the type of competing traffic is relatively limited. We use several parallel bulk transfers to inspect the effect of competition, which is likely to cause synchronization effects that might not be present in an open network. However, because we assume the bottleneck link to be dedicated for the user, we believe that the workload model used is good enough to be used in the performance evaluation, even when using only a single TCP connection as a workload.

A specific TCP implementation may contain features and bugs which have an unwanted effect on the performance. In particular, older versions of Linux are known to contain several features which cause incorrect behaviour [Pax97a]. Eliminating the known bugs in the Linux TCP implementation has been one of our special interests and an extensive set of preliminary tests were run before the actual performance tests in order to verify that the TCP implementation works correctly (the bug fixes that were made are listed in Appendix C). It is also important to be familiar with the TCP implementation and the corresponding specifications in detail to draw the correct conclusions from the test results and not to make too strong statements because of the implementation-specific features. We believe this is the case with our performance analysis. Furthermore, we analyse most of the test cases in detail using the tracing and the visualisation tools mentioned above in order to verify that the TCP endpoints have behaved according to the specifications.

Using additional delays in general to model the link-layer retransmissions, pre-emptive QoS scheduling or other behaviour of the underlying network architecture is an approximation which deserves to be discussed. Our model of delays resembles the behaviour of link layer retransmissions, which differs from the implications caused by excessive queueing delay, for example. However, because we are inspecting TCP performance, we are not interested in the reasons behind the modelled delay in detail, but the consequences the excessive delays have at the TCP layer, which are distorted RTO estimates and spurious timeouts. Furthermore, an important question is whether the occurances and the lengths of the additional delays resemble any real world case. As we do not have any measured data of the actual delay behaviour in the natural environment with actual link-layer protocols, we avoid making definite conclusions concerning any specific wireless architecture and discuss the effect of excessive delays on the TCP performance in general.

The accuracy of the delay lengths affecting the packets is an important issue, because the emulation is performed on a real-time basis. As described in Section 4.2, we tolerate inaccuracies up to 10 ms in the actual length of the delays. This kind of inaccuracy was very rare during our tests. Furthermore, a similar kind of variance in transmission delay could occur in the target environment, for example because of *byte stuffing* done by PPP[9]. However, we have been cautious to avoid byte stuffing from occuring in our tests by assuring that the TCP payload does not contain characters that would be stuffed.

Another factor which could cause unwanted inaccuracy is that the workload data is encapsulated within another TCP connection to be forwarded to Seawind. The congestion control mechanisms used for the outer TCP connection between a connection endpoint and Seawind could lead to notable inaccuracy. This is not the case with our tests, because the TCP MSS on the LAN is 1460 bytes, which can easily hold at least four packets from the actual workload. Secondly, we have disabled the *Nagle's algorithm*[10] from the outer connections to enforce the segments to be transmitted timely. By reviewing the TCP traces we verified that the behaviour of the outer TCP connection does not have any notable effect on the emulation accuracy.

Seawind handles the data in the granularity of IP packets, which causes slight inaccuracy, for example when defining buffer sizes in bytes. A real link layer protocol usually

[9]*byte stuffing* occurs if the PPP payload contains bytes that are used as terminal control characters. All such bytes are replaced by a two-byte sequence, which causes variance in the length of PPP frames.

[10]*Nagle's algorithm* [Nag84] makes the TCP sender delay the transmission of an under-sized segment until the next acknowledgement arrives in order to make it possible to collect more data in the same segment before transmitting it. For the workload segments Nagle's algorithm does not have an effect, because we always transmit full-sized segments in out workload.

fragments the IP packets into smaller units (e.g. GSM RLP uses frames of 240 bits), but the delay probabilities and lengths are defined on the granularity of IP packets in our emulation. However, we inspect the results at the TCP layer which handles the data as whole segments, thus the inaccuracy is not meaningful in our tests as long as the final result, an additional delay observed by the TCP receiver, is correct.

5 Analysis of the Baseline TCP

We analyse the baseline TCP implementation with various delay scenarios and three different router buffer sizes described in Table 2 on page 27. Our goal was to evaluate whether the problems described in Section 3 can be observed experimentally and whether there are additional problems caused by the implementation we are using. We take a detailed look at the problems to understand how the behaviour should be changed in order to improve the TCP performance. Another motivation for the detailed analysis is to validate the experimental results.

5.1 Single unidirectional connection

In this subsection we analyse the results of tests with single connection workload. 100 kilobytes of bulk data were transmitted from the remote host to the mobile host and only acknowledgements were transferred upstream. Results of single connection tests are summarised in Table 14 in Appendix A.

Effect of the router buffer size

The summary of test results in Table 14 in Appendix A shows that the smaller the router buffer, better the resulting performance, if there are additional delays involved. If there are no additional delays, the medium router buffer size gives the best throughput and the least packet drops. The large router buffer size yields slightly worse performance than small or medium buffers.

There are two distinct patterns of how the packets are dropped at the last-hop router. During the *slow start* there are usually several packets dropped in a single round-trip time because of *slow start overshooting.* After the slow start the sender enters *congestion avoidance*, during which one packet is dropped occasionally at constant time intervals.

The number of packets dropped because of slow start overshooting can be estimated analytically when there are no additional delays on the link. We observed that the number of packets dropped during slow start is exactly one more than what the pipe capacity is, with an exception of the extremely small router buffer size of one packet. The reason for the exceptional result with the router buffer size of one packet is not verified, but we believe it to be due to possible phase effects[11] caused by the small pipe. We describe the events

[11]Phase effects related to TCP communication are analysed by Floyd and Jacobson [FJ92], and we do not discuss them any further in this study.

causing the packets to be dropped in detail to justify the above presented statement.

The events following the slow start overshooting are illustrated in Figure 5. The figure illustrates an environment with pipe buffering capacity of 8 packets; the router buffer size is 3 packets, the link send buffer can hold 4 packets and one packet is outstanding elsewhere in the network (e.g. at the receiver) before the acknowledgement for it arrives at the sender. Similarly, by adding 5 packets to the router buffer size used we can determine the available buffering capacity for all the test cases analysed in this study.

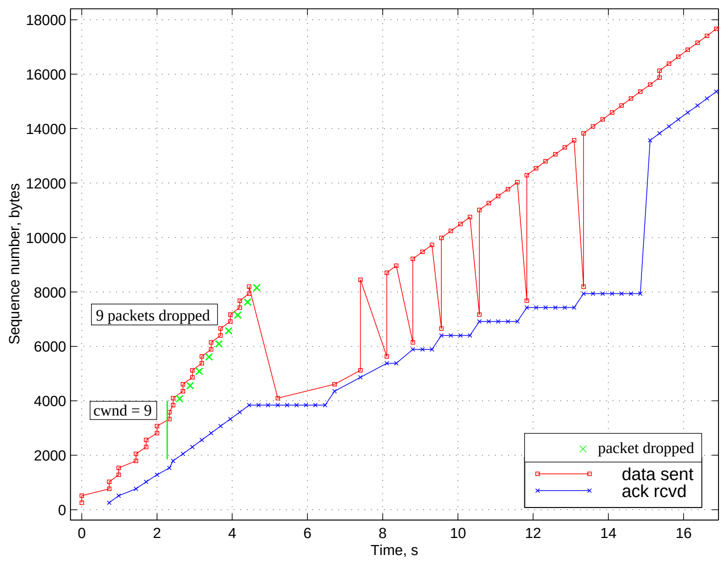

Figure 5: Implications of slow start overshooting with router buffer size of 3 packets.

The first packet is dropped because of router buffer overflow when the *congestion window* size exceeds the pipe capacity. For example, with a router buffer size of 3 packets the congestion window size is 9 * MSS at the time of the first packet loss. At this point there are 8 packets in the pipe between the sender and the receiver, for which the acknowledgements will eventually arrive to the sender. Each of these acknowledgements indicates that one packet has successfully arrived to the receiver. Every packet is acknowledged separately, because the *delayed acknowledgements* algorithm[12] does not have effect with the slow link we are using. Because the sender is in slow start, it will transmit two new

[12]*Delayed acknowledgements* algorithm causes the acknowledgement to be held back at the receiver for short amount of time (usually 200 ms) in order to make it possible to acknowledge more than one segments at a time. Because the transmission delay of a 303-byte packet over 9600 bps link is larger than 200 ms and Linux refrains from delaying acknowledgements when packets arrive at a low rate, delayed acknowledgements do not usually have effect in our environment.

segments for each incoming acknowledgement into the network, of which one does not fit into the router buffer and is therefore dropped. Hence, the number of packet drops following the first packet loss is equal to the pipe capacity.

Another pattern in which the packets are dropped can be observed during the congestion avoidance phase after the first buffer overflow. The packet drops occur in constant time intervals, the frequency of the packet drops being determined by the router buffer size. The packets are dropped because the congestion window size increases by one MSS for each round-trip time and eventually exceeds the pipe capacity, causing the router buffer to overflow. The smaller the router buffer, the more frequently the packets are dropped during the congestion avoidance.

Table 3 shows the number of packets dropped separated into packet losses during the first slow start and packet losses during the congestion avoidance, which the sender uses for the rest of the test run after the slow start overshooting. It can be noticed from the table that there are least packet losses with a router buffer size of 7 packets, which explains why the throughput is best with that router buffer size. However, the optimal buffer size depends on the amount of transferred data, because the number of packet losses during the congestion avoidance increases with the amount of transferred data. On the other hand, the number of packets dropped during the first slow start is usually the same regardless of the amount of data transferred, as long as enough packets are transmitted to trigger the fast retransmit.

Table 3: Number of dropped packets during slow start and during congestion avoidance

Buffer size (pkts)	**Pipe capacity (pkts)**	**Drops / slow start**	**Drops / congestion avoidance**	**Drops total**
1	6	9	13	22
3	8	9	9	18
7	12	13	4	17
20	25	26	1	27

Based on the justification given above we constructed Equation 1 to estimate the number of packet losses caused by a router buffer overflow with a single TCP connection, when there are no packet corruption on the link and no retransmission timeouts occur. Due to reasons described in the footnote above, it is assumed that the delayed acknowledgements do not have effect on the TCP behaviour, and that the fast recovery algorithm is equally aggressive to the congestion avoidance algorithm, as it is in our baseline TCP implementation. Moreover, the equation assumes that the data transmission is not application limited, i.e. it is bulk transfer. We tested the equation against the experimental results in Table 3, and got matching results excluding the test case with a router buffer size of 1 packet, which was briefly discussed above. Although we skip the formal proof of the equation at this time, we believe the equation is useful when estimating how the router buffer size and the amount of data to be transferred affect the number of dropped packets.

Let m be the number of non-retransmitted segments to be transmitted, i.e. the amount of the workload data divided by the MSS, which is 400 with our test cases. Let n be the pipe buffering capacity plus one, i.e. the congestion window size at the moment when packet loss occurs. The result estimates the number of packet losses during the test run.

$$x = \lfloor \frac{2(m-2n)}{n(n+1) - (\lfloor n/2 \rfloor - 1)(\lfloor n/2 \rfloor)} \rfloor + n \tag{1}$$

The packet losses due to slow start overshooting cause the acknowledged throughput to be significantly lower for the duration of the *fast recovery* algorithm following the packet losses. As described in Section 3, the NewReno TCP sender can do only one retransmission in a round-trip time during fast recovery. Hence, the larger the router buffer, the more round-trip times are needed before the sender has retransmitted the segments dropped due to slow start overshooting. Moreover, the round-trip time gets higher for each retransmission, because our TCP implementation increases the number of outstanding segments with each retransmission during the fast recovery, causing the queueing delay at the last-hop router to gradually increase. The increasing round-trip time can also be observed in Figure 5 on page 34.

Table 4 shows the the measured acknowledged throughput for transmitting given amount of data with different router buffer sizes. The table shows that although the throughput is very low at the time when the sender is in fast recovery (marked as *(FR)* in the table), the final throughput is not significantly lower than the optimal value[13] after 100 kilobytes have been transferred. This is because the fast recovery algorithm allows the sender to transmit new data when new duplicate acknowledgements arrive, hence keeping the communication path utilised.

Effect of the additional delay length

We made tests with additional delays of selected lengths to see the effect of the delays on the TCP performance and to find possible problems in the behaviour of the baseline TCP. We were interested to see how an additional delay affects the occurance of buffer

[13]Tests with unlimited router buffer and without additional delays yield throughput of 1003 Bps [Kuh]

Table 4: Achieved throughput in bytes per second after the given amount of data has been transmitted and acknowledged with the last hop router buffer size shown.

Buffer size (pkts)	**3 KB**	**5 KB**	**10 KB**	**25 KB**	**50 KB**	**100 KB**
1	830.4	602.2 (FR)	785.1	933.8	973.0	985.4
3	831.1	631.1 (FR)	677.7	949.8	981.7	978.9
7	831.2	896.6	564.6 (FR)	923.6	981.4	998.3
20	831.5	896.5	952.5	500.3 (FR)	923.3	967.3

overflows in cases when the delay did not cause the retransmission timeout to expire, and on the other hand, how does a spurious retransmission caused by the additional delay and the resulting *go-back-N* behaviour affect the performance. Furthermore, we tested an additional delay which was long enough to cause a *backed off*[14] retransmission for a segment in addition to the first retransmission of the same segment triggered by the retransmission timer.

An additional delay shorter than what is required for the RTO to expire does not cause significant effect on the throughput. Because the additional delay primarily emulates link layer retransmissions maintaining the ordering of the packets, the packets that are in the link buffers are considered to be transmitted to the receiving end during the additional delay[15]. All the packets which have arrived to the link receive buffer during the additional delay are delivered to the higher protocol layers at once. After the packets are delivered to higher protocol layers, they are removed from the sending and receiving link buffers. Therefore, the next four packets from the router queue can be moved to the link send buffer, making room for four new packets in the router queue.

Because all the packets in the link receive buffer are delivered to the receiving host at once after an additional delay, the receiver generates a burst of acknowledgements. These acknowledgements arrive at the sender in a higher rate than normally, when the ACK segments were clocked by the steadily incoming data segments.[16] If the sender is in slow start, it increases the congestion window size by one segment for each incoming acknowledgement and transmits a short burst of data into the network. Similar behaviour

[14]Due to *retransmission ambiguity* the RTO estimate and RTT measurements are not made for the retransmitted segments. Instead, the RTO value is *backed off* to be twice its previous value [KP87].

[15]This assumption does not hold, if the delay is not caused by link level retransmissions.

[16]The transmission delay of an ACK segment without data is less than 40 ms with a link bandwidth of 9600 bps, whereas the transmission delay of a full-sized segment is approximately 250 ms.

has been noticed also in the related work with similar link characteristics [CLM99, LK00].

The behaviour caused by a small additional delay is illustrated in Figure 6. The number of packets dropped in test case with short delay is reduced by 1-2 packets when compared to the corresponding test cases without additional delays. The number of packet losses is smaller, because the receiver uses the delayed acknowledgements algorithm when sending the burst of acknowledgements. The delayed acknowledgements algorithm causes the receiver to generate less acknowledgements than normally, hence the sender increases its congestion window less aggressively and transmits less segments after the small additional delay than when there was no additional delay.

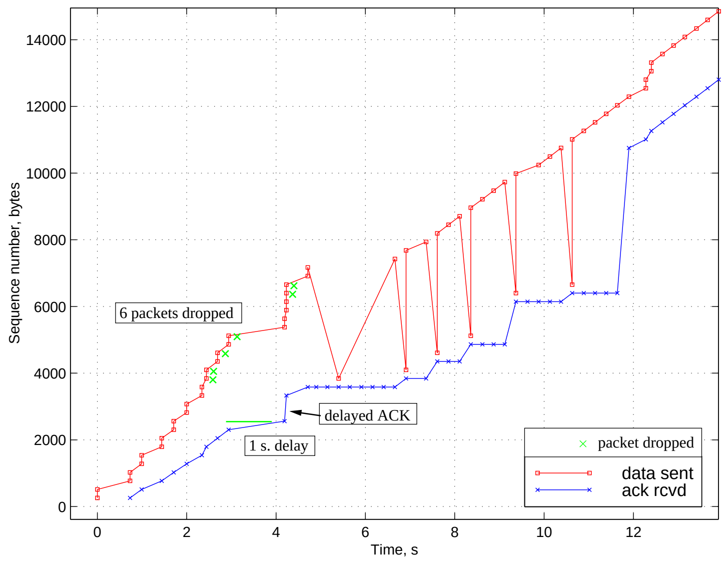

Figure 6: A short additional delay in the beginning of connection. The router buffer size is three packets.

When an additional delay long enough to trigger a *spurious retransmission timeout* occurs, the performance suffers significantly. Figure 7 shows the events following the spurious retransmission timeout. When the unnecessary retransmissions arrive at the receiver, it considers them out-of-order segments and generates a duplicate ACK for each retransmitted segment. The sender reacts to the duplicate ACKs by doing a fast retransmit and unnecessarily entering the fast recovery algorithm, causing yet more unnecessary retransmissions and unnecessary deflation of the congestion window.

The problem of the needless fast retransmit caused by duplicate ACKs after unnecessary retransmissions is also identified by the authors of the RFC describing the NewReno

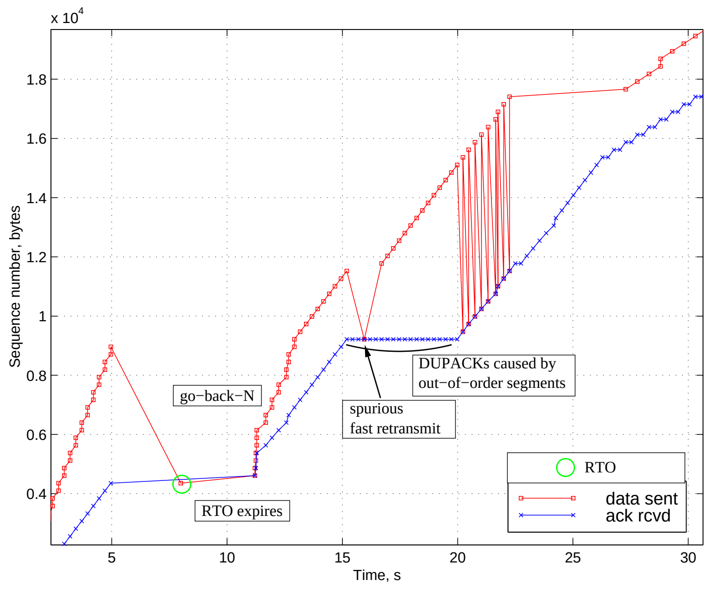

Figure 7: A spurious timeout triggered by an additional delay of 6 seconds.

algorithm [FH99]. Therefore it is recommended that fast retransmit and fast recovery should not be entered when retransmitting after a retransmission timeout. This is to avoid the unnecessary fast retransmit caused by the duplicate ACKs, which causes the sender to enter fast recovery algorithm and unnecessarily retransmit another window of segments in addition to the segments retransmitted after the spurious retransmission timeout. The above-mentioned rule is called "NewReno *bugfix*".

Two variations of the NewReno bugfix have been suggested. The *less careful* variant disables the fast retransmit and the fast recovery algorithm until the segments up to the last originally transmitted segment before the retransmission timeout have been acknowledged. This variant is implemented in our baseline TCP, but it does not help in avoiding the unnecessary fast retransmit caused by the out-of-order segments, as can be seen in Figure 7. The *careful* variant disables the fast retransmit and the fast recovery algorithm until more segments have been acknowledged than what was transmitted before the retransmission timeout. This variant avoids the unnecessary fast retransmit and fast recovery caused by unnecessary retransmissions, but it may cause an additional retransmission timeout in a scenario in which the first of the new data segments transmitted after the retransmission timeout is dropped.

The effects caused by the additional delay triggering a spurious retransmission timeout varies depending on the location in the data flow at which the delay occured. The TCP sender takes different actions depending on whether there are packets dropped before the additional delay or after the additional delay. These cases are discussed later in this section. Nevertheless, the reason for decreased throughput is the unnecessary retransmissions triggered by the spurious retransmission timeout.

An additional delay long enough to trigger two retransmission timeouts for the same segment is rare in the natural environment, but could occur, for example, as a result of a bad link outage. Two successive retransmission timeouts may significantly diminish the achieved throughput. This is not only because the delay is longer, but both of the retransmission timeouts back off the calculated RTO value by doubling its length twice. If a retransmission is lost after the retransmission timeouts, a long idle period follows, because the only way to recover from a dropped retransmission is to wait for the retransmission timeout to expire. Because the retransmission timer was backed off twice, the RTO length will be four times the original value measured from the RTT samples. Moreover, the larger the router queue length at the moment of the additional delay, the longer the RTO length will be, because the queueing delay increases the measured RTT values. This occured in our tests with the additional delays causing two successive retransmission timeouts, which resulted in collapsed throughput.

In Table 14 in Appendix A we can see that if the additional delay does not cause a spurious retransmission timeout, it does not have negative effect on the TCP performance. In fact, with small router buffer the throughput is slightly improved. On the other hand, an additional delay which causes a spurious retransmission timeout will lower the throughput, especially when it is followed by packet losses due to buffer overflow. In addition, the table shows that larger the router buffer, worse is the performance when an additional delay long enough to trigger a retransmission timeout occurs.

Effect of the additional delay location

We made tests with an additional delay long enough to cause a spurious retransmission timeout in a selected location of the test run. In particular, we were interested to see the effect of an additional delay which occured during the initial slow start, before there are packets dropped due to slow start overshooting. Further, we made tests with an additional delay that occured during the fast recovery following the first slow start overshooting and an additional delay that occured during the congestion avoidance later in the test run. We concentrated only on delays long enough to cause a spurious retransmission timeout, because preliminary test runs revealed that a shorter delay does not cause surprising effects which would deviate from what was described above, regardless of the location of the short additional delay.

Figure 8 illustrates a connection in which an additional delay of 6-seconds triggers a retransmission timeout. After the retransmission timeout the sender starts retransmitting segments unnecessarily using slow start, as explained in Section 3.2. Moreover, the receiver sends a burst of ACKs because the whole link receive buffer has been delivered at once after the additional delay, which results in a burst of segments transmitted by the sender using the slow start algorithm.

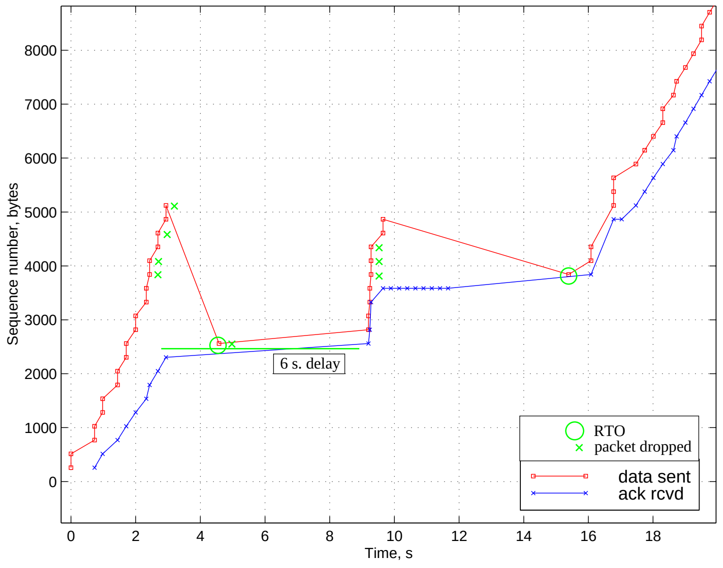

Figure 8: An additional delay of 6 seconds in the beginning of connection. Router buffer size is three packets.

Some of the originally transmitted segments are dropped at the router due to buffer overflow soon after the sender has transmitted the segment which suffered from the additional delay. Because of this, some of the retransmissions after the spurious timeout are not unnecessarily made. Unfortunately, two of the necessary retransmissions are dropped due to router buffer overflow. Because the NewReno bugfix is implemented in our TCP, the sender cannot do fast retransmit and enter the fast recovery, even though more than three duplicate ACKs arrive. Therefore, the sender must wait for the retransmission timer to expire. Because the sender is retransmitting, it must use the backed off retransmission timer, which has twice the value of the first RTO that occured due to the additional delay. Before the retransmission timer expires, the sender cannot transmit new data and the throughput deteriorates significantly. This scenario occurs every time a necessarily made retransmission is dropped when retransmitting after a retransmission timeout, and it indicates that the NewReno bugfix can also be a reason for bad performance.

An additional delay which occurs during the fast recovery after the initial slow start does not cause as notable decrease in throughput as the additional delay which occured just before the first packet loss. This scenario is illustrated in Figure 9, which shows that the fast recovery algorithm is interrupted by the retransmission timeout, which is followed by retransmissions using the slow start algorithm. Because there were packets dropped at the last hop router before the additional delay, all retransmissions following the spurious retransmission timeout are not unnecessarily made. Therefore the number of duplicate ACKs caused by out-of-order segments is smaller. Similarly, several segments are cumulatively acknowledged with a single acknowledgement during the go-back-N retransmission phase, which causes the sender to transmit small bursts of data segments during the retransmissions.

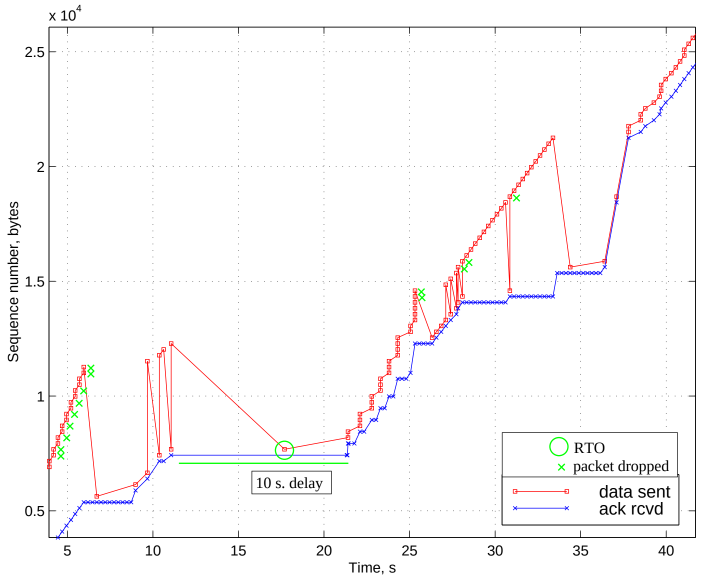

Figure 9: An additional delay during the fast recovery following the first router buffer overflow. The router buffer size is 7 packets.

The out-of-order segments caused by the unnecessary retransmissions made after the spurious retransmission timeout cause a spurious fast recovery because the careful variant of the NewReno bugfix is not implemented in our baseline TCP. During the unnecessary fast recovery the router buffer becomes full, because the partial ACKs cause the number of segments outstanding in the network to be increased gradually. After the router buffer has become full, every new segment sent with a partial ACK is dropped, which indicates that sending new data with the retransmissions on partial ACKs should not be allowed.

An additional delay which occurs during the congestion avoidance after roughly three quarters of the 100 KB transfer have been completed does not cause as severe problems in the TCP behaviour as the additional delay which occurs in the beginning of the test run. Figure 10 illustrates an additional delay during congestion avoidance. We can see that the number of unnecessary retransmissions depends on the congestion window size at the moment when the segment is delayed. The number of unnecessary retransmissions is usually higher with large router buffer, because a larger router buffer size allows the congestion window to be larger.

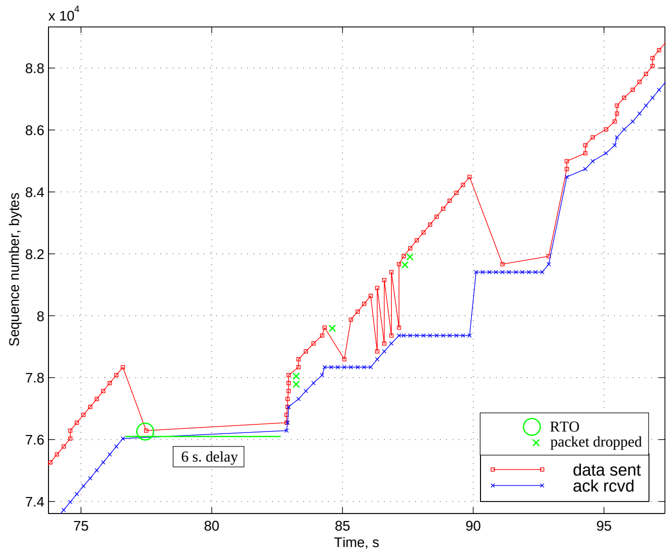

Figure 10: An additional delay during the congestion avoidance. The router buffer size is 7 packets.

In addition to the selected single-delay tests, we made a test with two additional delays of which the first occurs during the fast recovery following the first packet loss and the second occurs during the spurious fast recovery triggered by the out-of-order segments caused by the first delay. However, the second delay did not cause surprising performance implications, but similar spurious retransmission timeout followed by slow start than what single additional delay did when it occurred during fast recovery.

5.2 Multiple parallel unidirectional connections

We evaluated the performance of multiple parallel TCP connections sharing the wireless link. Each of the TCP connections were transmitting 50 kilobytes of unidirectional bulk data. We primarily measured throughput of the connection which was finished last to see the effect of competition over the bottleneck link. Additionally, we inspected fairness among the parallel TCP connections, as unfairness was identified as a problem in Section 3.4.

Two parallel connections without additional delays

We made a set of tests with two parallel unidirectional TCP connections when no additional delays occured on the link. A summary of the results of the full TCP connections can be seen in Table 15 in Appendix A. Additionally, we have extracted the results for the first 10 KB of the parallel TCP connections in Table 16 in Appendix A. When observing the results of the full TCP connections, one can notice that although the large router buffer results in unfair sharing of the bandwidth, the best throughput can be achieved with it. However, as explained in Section 4.3, we are not usually interested in the results of full connections, hence we primarily inspect the results from the first 10 KB of the parallel TCP connections, in which case using the small router buffer results in the highest throughput, when there are no additional delays on the link.

A detailed look at the test results reveal that the behaviour of the parallel TCP connections changes during the test because of the interaction between the connections. Figure 11 shows the two parallel TCP connections transmitted through a last-hop router with a buffer size of 20 packets. Sender A has transferred 10 kilobytes in 25 seconds, but it takes 41 seconds for sender B to transfer the same amount of data. However, sender B transfers all of the 50 kilobyte workload in 91 seconds, but connection A is not finished until 102 seconds have passed. The change in the TCP behaviour occurs because most the packet losses caused by the slow start overshooting are concentrated on connection A, because sender A transmitted more segments before the router buffer overflow.

When there are parallel TCP connections transmitted over the wireless link, each window of segments for a particular connection is transmitted separately over the link, as described in Section 3.4. Because the senders are in slow start, they double the congestion window size on each round-trip time. As a result, the queueing delay perceived by the competing sender doubles each round-trip time. Because increasing the congestion window for one sender results in increased amount of delay for the other sender, the unfairness between the connections increases as the congestion window gets larger. Therefore, for

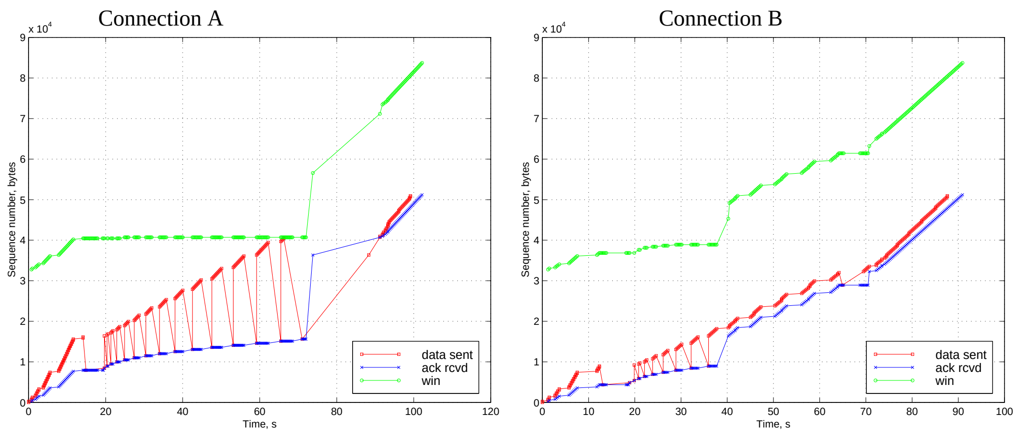

Figure 11: Two parallel connections through a router buffer of 20 packets.

most of the time during first slow start sender A has transmitted almost twice as much segments than sender B and the congestion window size for sender A is twice the congestion window size for sender B.

When the last-hop router buffer is filled up, several packets are dropped because of the slow start overshooting, as observed with the single connection tests. This causes both senders to enter the fast recovery algorithm at the same time, illustrating the problem of *global synchronization* discussed in Section 3.4. Because sender A has significantly larger congestion window size than sender B, most of the dropped packets are transmitted by sender A. Therefore, sender B has less retransmissions to be made, thus it can finish retransmitting and proceed with the congestion avoidance algorithm, while sender A is still retransmitting.

After the senders have made the fast retransmit, the pipe between the sending host and receiving host gets empty before the senders start transmitting again using the fast recovery algorithm[17]. Once the senders have started transmitting again, they increase the congestion window size by one segment only once in a round-trip time (due to partial ACKs or due to congestion avoidance), hence the senders are transmitting at equal rate from that point on.

The reason why connection A ends up with lower throughput than connection B is

[17]Reno and NewReno require that the number of outstanding segments is halved after the third duplicate ACK, but because most of the segments are dropped, the pipe is empty after the sender has received duplicate ACKs worth half of the congestion window. By using the rate-halving algorithm [SM99] the pipe could be utilised more efficiently.

that one of the segments sent by sender A during the fast recovery is dropped. Moreover, because several segments are dropped earlier, the advertised window limit of sender A is reached during fast recovery and after reaching the window limit, sender A cannot send new data when receiving partial ACKs. Therefore there are no duplicate ACKs arriving to trigger the retransmission and sender A has to wait for the retransmission timeout to expire, which causes an idle period of 13 seconds in the transfer. However, this occurs only in the test case with a router buffer size of 20 packets.

When a last-hop router with a small buffer is used, the first buffer overflow occurs earlier and the congestion window size has not inflated as much as in the above-presented example. Both of the parallel TCP connections suffer from a short fast recovery period after the first packets are dropped at the router. After the fast retransmit the connections get equal share of the link bandwidth in a similar way described above for the test case with large router buffer. As a result, the best fairness can be achieved when using the router with small buffer.

Two parallel connections with additional delays

When there are additional delays at the link shared by two parallel TCP connections, the results differ remarkably from the scenarios without the additional delay. The results in Table 16 in Appendix A show that with a small and medium router buffer, the presence of additional delays causes unfairness among the TCP connections when inspecting the first 10 KB of the parallel TCP connections.

When the a small router buffer is used, an additional delay causes the most substantial effects on the performance. Using the small router buffer increases the probability of having a retransmitted packet dropped when retransmitting after a retransmission timeout, which results in another retransmission timeout because of the NewReno bugfix. Additionally, the queueing delays are smaller with the small router buffer, which causes the calculated RTO value to be smaller and the probability of retransmission timeouts to be higher. When one of the TCP connections suffers from a retransmission timeout, another connection is likely to benefit from that, because it can use the whole bandwidth of the link while the sender of the other connection waits for the retransmission timer to expire.

Figure 12 gives an example of the effects of an additional delay causing a spurious retransmission timeout for both TCP senders when the last-hop router buffer size is 3 packets. Connection A transmits 10 kilobytes of data in 25 seconds whereas connection B uses 71 seconds for transmitting the same amount of data. The figure also illustrates the *lock-out* problem which was discussed in Section 3.4.

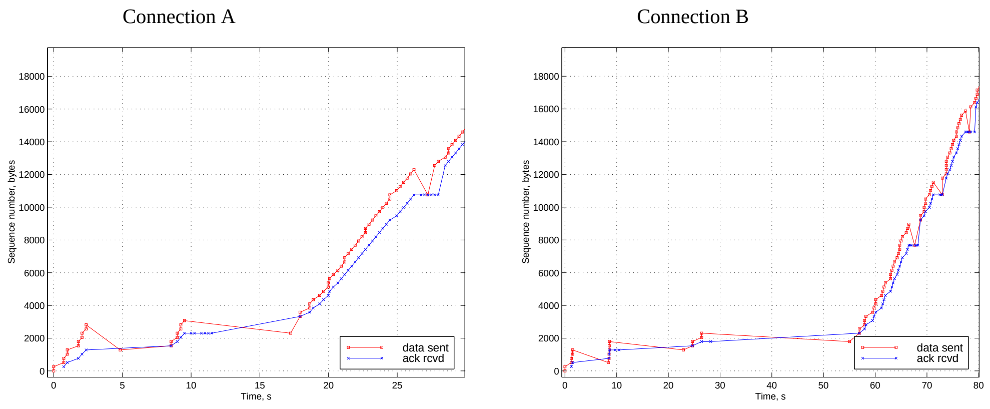

Figure 12: Two parallel connections through a router buffer of 3. An additional delay of 6 seconds occurs on 12th packet.

After the additional delay triggering the spurious retransmission timeout is finished, both senders start retransmitting using the slow start algorithm. However, because of the small router buffer size used, segments from both connections are dropped, which causes another retransmission timeout for both senders, because the NewReno bugfix does not allow the senders to retransmit after receiving three duplicate ACKs. Sender B has a higher RTO estimate than sender A, because it suffered from higher queueing delays at the beginning of the connection due to the packet separation effect described above. The difference in the throughput is further emphasized, because the RTO estimates for both senders are backed off once because of the earlier spurious retransmission timeout.

Because of the different RTO estimations between the two senders, sender A restarts the transmission first after the second RTO at 17 seconds, while sender B is still waiting for its retransmission timer to expire. When the retransmission timer expires for sender B at 22 seconds, sender A has been transmitting for 5 seconds after its RTO expired, and the pipe between the sender and the receiver is almost full of packets transmitted by sender A, causing another packet loss for connection B. Because sender B did not transmit enough segments to have three duplicate ACKs before the packet loss, it has to wait for the retransmission timer to expire again, leaving all of the link bandwidth to be used by sender A. This time the RTO length for sender B is four times the originally calculated value due to the timer back-off algorithm.

The lock-out behaviour described above results in substantial unfairness. After the third retransmission timeout expired, sender B can finally transmit enough of its segments to the network to avoid further timeouts. At this point sender B has successfully transmitted only 3 KB of its workload, whereas sender A has transmitted 40 KB of the 50 kilobytes it has to transmit.

Use of the medium and large last-hop router buffers does not cause as low throughput as use of the small router buffer, because the larger router buffers avoid the lock-out problem which was present when using the small router buffer. For the tests with random additional delays medium router buffer size is a good compromise; with a smaller router buffer the probability of retransmission timeouts and the lock-out behaviour is higher due to smaller RTO estimates, but with a larger router buffer size there are more unnecessary retransmissions after a retransmission timeout because of larger average congestion window size.

Four parallel connections

We made tests with four parallel TCP connections using small, medium and large last-hop router buffer sizes without additional delays, with an additional delay just before the first packet loss caused by router buffer overflow, and with random additional delays. Each of the TCP connections transmitted 50 KB of unidirectional bulk data. We mainly analyse the first 10 KB of the parallel TCP connections, of which the test results are shown in Table 18 in Appendix A. The test results for the full connections are shown in Table 17 in Appendix A.

Four parallel connections fill the router buffer very quickly and even with a medium-sized router buffer one of the TCP senders is forced to wait for the retransmission timeout to trigger a retransmission, because it could not transmit enough data to generate three duplicate ACKs before the first router buffer overflow. During the time when the connection is waiting for the retransmission timeout to expire, the other connections fill the available router buffer space. Therefore, one of the retransmissions triggered by the retransmission timeout is also dropped because the router buffer is full. The sender cannot do fast retransmit, hence it must wait for another retransmission timeout to do the retransmission. Because of the timer back-off algorithm, the RTO length for the second retransmission is twice as long as the original RTO. As a result of this typical lock-out behaviour, it takes 138 seconds for the slowest connection to transfer 10 KB, when the fastest connection can transfer the same amount of data in 24 seconds. The lock-out behaviour and the resulting substantial unfairness could be avoided only when the large router buffer was used. Similarly, the throughput of the connection finished last was highest when using the large router buffer.

The additional delays do not cause significant differences in the results. When using four parallel TCP connections, the main problem is the extensive congestion at the last-hop router, causing packet losses which cannot be recovered by using fast retransmit. Because there are a number of retransmission timeouts because of the packet losses at the router, the additional spurious retransmission timeouts caused by the additional delays do not have a great effect on the performance. The trend is the same with and without the delays, the best fairness can be achieved by using the large router buffer.

Limited transmit [ABF01] is expected to improve the performance with parallel connections, because it allows sending new data on the first two partial ACKs before the third duplicate ACK. In this way the receiver would get enough segments to trigger at least three duplicate ACKs, allowing the sender to do fast retransmit. By doing this most of the RTOs could be avoided.

5.3 Summary of the baseline tests

After analysing the behaviour of the baseline TCP in various delay scenarios we have encountered certain problems in the TCP behaviour. We now briefly summarise the noticed problems, and during the analysis of the TCP enhancements we will inspect whether the problems are avoided by the enhancements.

As described in Section 3, the behaviour of the fast recovery is not strictly specified in the related IETF specifications [APS99, FH99]. The alternatives are either to maintain the number of packets outstanding in the network, following the packet conservation rule, or to increase the number of outstanding packets for each round-trip time, which resembles the behaviour of congestion avoidance. Although neither of the alternatives cause serious problems, the variant which increases the amount of outstanding packets may cause packet losses during the fast recovery. The packet losses that occur during the fast recovery do not cause the congestion window to be deflated. Therefore we suggest using the less aggressive variant, which does not increase the congestion window during the fast recovery.

The less careful variant of the NewReno bugfix implemented in our baseline TCP does not prevent the spurious fast retransmit after a spurious retransmission timeout and causes more unnecessary retransmissions, which could be avoided with the careful variant of the bugfix. Both bugfix variants can also cause an extra retransmission timeout if a retransmission following the spurious timeout is lost, for example, due to router buffer overflow. We are not aware of any reports discussing this problem. An alternative implementation for the bugfix, which would allow the fast retransmit below the `send_high` threshold[18], is

[18]The `send_high` variable stores the highest TCP sequence number transmitted when a retransmission timeout occurs, and is used by the NewReno bugfix to decide whether to allow fast retransmit.

worth inspecting in the future.

Two factors affect the performance of the tests with random additional delays. A small router buffer results in smaller average queueing delay and smaller RTO values. This increases the probability of retransmission timeout when an additional delay occurs. A large router buffer, on the other hand, allows a larger average congestion window, which results in a larger number of unnecessary retransmissions when a spurious retransmission timeout occurs. Because of the negative impact caused by the the unnecessary retransmissions, using the small router buffer resulted in the best performance on tests with random additional delays.

When multiple parallel connections are transmitted through the bottleneck link with additional delays, one of the connections suffers significantly because of the lock-out behaviour, if too small a router buffer is used. Lock-out behaviour usually follows a retransmission timeout, which causes some of the connections to be idle for a short time period, during which the other connections can fill the pipe between the sender and receiver. Because the NewReno bugfix causes more retransmission timeouts when packet losses occur after a retransmission timeout, use of it will increase the number of lock-out occurances and the unfairness between the TCP connections. If there are no additional delays on the link, the parallel TCP connections are separated on the link, which can also cause unfairness because of unequal queueing delays.

6 Analysis of the TCP Enhancements

We listed a number of TCP enhancements in Section 4.4 which are expected to improve the TCP performance over what measured with the baseline TCP in Section 5. In this section we analyse the effects of the TCP enhancements for the test cases which were noticed to be problematic with the baseline TCP.

6.1 Results of the SACK tests

Selective Acknowledgements (SACK) [MMFR96] are reported to improve the TCP performance when there are several segments dropped within a single round-trip time [FF96]. Considering the extra information about missing segments provided by the SACK acknowledgements, this is easy to believe.

Tests with a single TCP connection

Results of the SACK tests are shown in Table 19 in Appendix A. The test results are compared with the corresponding baseline TCP test results in Table 5. The table shows the elapsed time and the number of retransmissions for the baseline TCP and the SACK TCP. It can be observed that when SACK is used, more retransmissions are done than with the baseline TCP, if no additional delays occur. On the other hand, if an short additional delay occurs, there are no significant differences in the number of retransmissions, but the elapsed times differ notably.

When SACK is used, retransmissions are likely to get dropped and unnecessary retransmissions are made when no additional delays occur. However, if a short additional delay occurs before the router buffer overflows first time, the router buffer is slightly less loaded after the delay and no retransmissions are dropped. There are no unnecessary retransmissions with a short additional delay. Figure 13 illustrates the beginning of the TCP connection in the test case with small router buffer and without an additional delay. Unnecessary retransmission occurs after 13 seconds have elapsed. This test case can be compared with the test case with an additional delay of one second occuring before the first router buffer overflow, which is shown in Figure 14.

We consider the unnecessary retransmissions to be caused by a Linux specific behaviour which should not be thought of as a problem of SACK algorithm in general. After a retransmission timeout the SACK sender restarts the retransmissions based on the information received from the SACK blocks contained in the following acknowledgements. Because of

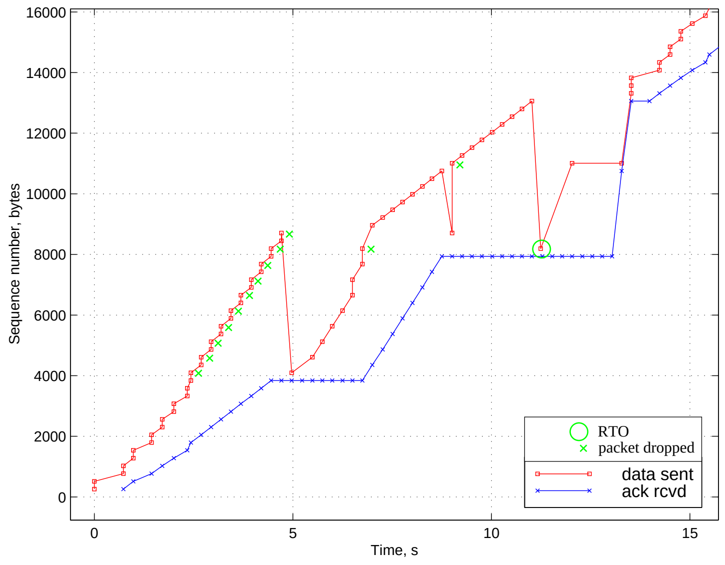

Figure 13: SACK TCP through router buffer size of three packets. No additional delays.

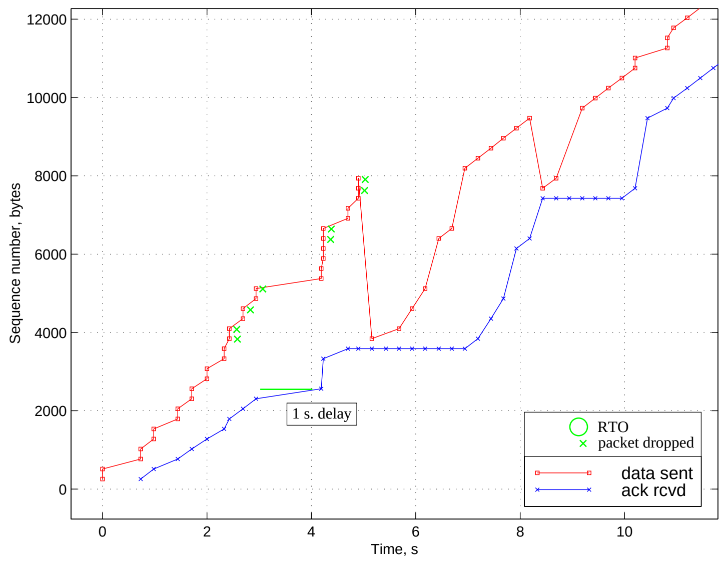

Figure 14: SACK TCP through router buffer size of three packets. An additional delay of one second occurs before the first buffer overflow.

Table 5: Comparison of the single connection test results of the baseline TCP and the SACK TCP.

		Baseline		**SACK**	
Buffer (pkts)	**Delay**	**Time (sec)**	**Rexmits**	**Time (sec)**	**Rexmits**
3	-	105.15	18.00	102.49	20.00
	12th, 1 s.	102.66	16.00	102.48	16.00
	12th, 6 s.	113.82	24.50	109.20	23.00
	30th, 6 s.	114.76	40.50	109.72	29.00
	rand. 0.01, 6 s.	130.63	42.00	129.62	43.00
7	-	103.11	17.00	103.04	26.00
	20th, 1.5 s.	105.11	16.00	104.92	15.00
	20th, 6 s.	120.31	24.00	111.80	24.00
	40th, 10 s.	116.86	34.00	113.70	34.00
	300th, 6 s.	112.52	32.00	122.77	31.00
	20th, 15 s.	140.30	24.50	118.44	19.00
	rand. 0.01, 6 s.	132.61	47.00	138.47	44.50
20	-	106.40	27.00	102.79	32.00
	40th, 4 s.	109.80	25.00	106.24	27.00
	40th, 6 s.	131.04	42.00	118.32	71.00
	100th, 12 s.	142.28	104.00	125.42	28.00
	300th, 6 s.	117.60	51.00	129.59	45.00
	rand. 0.01, 6 s.	137.42	52.50	142.74	64.00

the additional information provided in SACK block, the sender can usually transmit only the missing segments and avoid the unnecessary retransmissions, which were present with the baseline TCP after a retransmission timeout. The unnecessary retransmission occurs when the cumulative acknowledgement field is inflated after the retransmission timeout. This happens in Figure 13, because the sender does its next retransmission after the retransmission timeout based on the SACK information. However, the receiver sends the increased cumulative acknowledgement before receiving the next missing segment, but the acknowledgement arrives to the sender after it has retransmitted the missing segment. Because of this asynchrony, the sender retransmits the same segment again. The occasional unnecessary retransmissions caused by the Linux SACK occur only after a retransmission timeout and do not significantly affect on the throughput.

There are more packets dropped due to router buffer overflow with SACK than without it, if there are no additional delays. Reason for this is that the SACK sender starts retransmitting the dropped segments as soon as it gets feedback about the missing segments in the SACK blocks contained in the duplicate ACKs. Because the slow start overshooting causes several segments to be dropped, almost every acknowledgement arriving at the sender contains more information about the missing segments. Therefore, the sender makes a new retransmission for almost every incoming acknowledgement. This violates the principle of halving the number of outstanding segments after a packet loss is observed, which should be followed by every TCP sender. Thus, the number of packets outstanding in the network is constantly too high, when the load on the network should be decreased. This causes further router buffer overflows during the retransmissions, and in case a retransmitted segment is dropped, a retransmission timeout. This happens in Figure 13.

Despite the aggressiveness of FACK and the increased number of dropped packets, the SACK TCP slightly improves throughput in the presence of serious congestion. SACK allows several retransmissions to be made during one round-trip time, which makes the recovery after the slow start overshooting more efficient. With the baseline TCP a new retransmission is triggered only when a partial acknowledgement arrives to the sender, i.e. once per round-trip time.

Table 5 shows that if an additional delay long enough to cause a spurious retransmission timeout occurs during the first slow start, throughput with the SACK TCP is significantly better than with the baseline TCP. Nevertheless, there are more retransmissions made with SACK. The reason for the low throughput with the baseline TCP was that the packet losses after the spurious retransmission timeout caused another retransmission timeout to expire after an idle period in transfer, because the NewReno bugfix disabled the fast retransmit.

Figure 15 shows a spurious retransmission timeout occuring during the initial slow start. Because the SACK TCP does not follow the NewReno bugfix, the second retransmission timeout which was present with the baseline TCP (see Figure 8 on page 41) can be avoided. Instead, the SACK algorithm in Linux allows the sender to retransmit the missing segments as soon as the needed information from the SACK blocks is available at the sender. Because each SACK block arriving at the sender contains new information about the successfully received segments, the sender can keep transmitting segments, causing new acknowledgements to be generated for every segment arrived at the receiver. As a result, the pipe between the sender and the receiver is utilised efficiently.

The SACK TCP did not improve the performance in random delay tests, despite the notable improvement in cases where the NewReno bugfix and retransmission timeouts interact at the beginning of the connection. SACK does not help when additional delays occur during the congestion avoidance, as it is meant for recovering from multiple packet losses during one round-trip time. Instead, an enhancement of SACK called *D-*

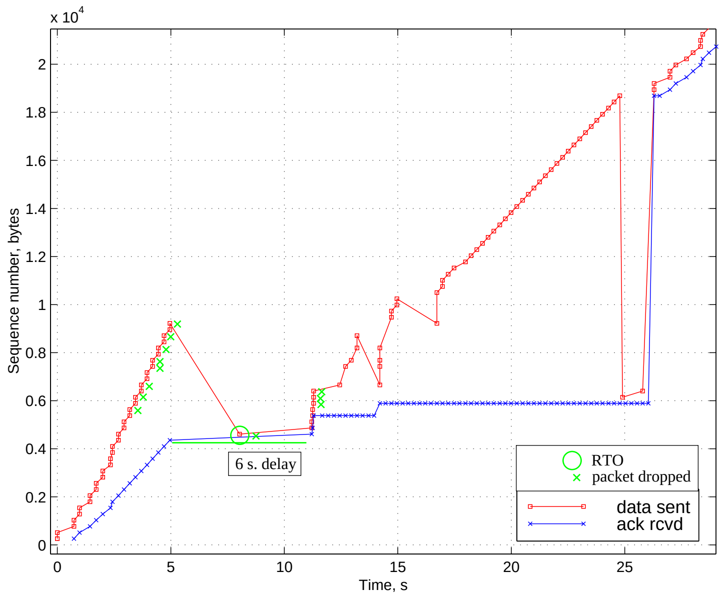

Figure 15: SACK TCP through router buffer size of 7 packets. Additional delay of 6 seconds occurs before the first buffer overflow.

SACK [FMMP00] should make it possible to detect the unnecessary retransmissions and hence improve the performance on test cases with spurious retransmission timeouts. Because the SACK blocks are an extra overhead in the transmitted segments, the throughput is lower with SACK than with the baseline TCP.

Tests with two parallel TCP connections

Table 20 in Appendix A shows that SACK slightly improves the throughput of one of the two parallel TCP connections, when compared to the baseline TCP without additional delays. However, SACK TCP increases the unequality between the parallel TCP connections, thus the other connection has usually slightly lower throughput than with the baseline TCP. Additionally, SACK TCP causes more retransmissions than the baseline TCP, because of the aggressiveness of the SACK algorithm. The same trend can be seen with the first 10 KB of the TCP connections shown in Table 22 in Appendix A.

Throughput achieved with the SACK TCP and with the baseline TCP with two parallel TCP connections are compared in Table 6. The table shows the throughput of the connection which was last to receive the acknowledgement for 10 KB. When SACK is used, the network load is higher, because the SACK TCP senders are more aggressive than the baseline TCP senders. A congested last-hop router is a reason for unfairness, as described in Section 5.2.

Table 6: Comparison of the throughputs of the baseline TCP and the SACK TCP with two parallel connections.

		Baseline		**SACK**	
Buffer (pkts)	**Delay**	**Throughput (bytes/sec)**	**Fairness (x 100)**	**Throughput (bytes/sec)**	**Fairness (x 100)**
3	-	491.55	100.00	394.96	98.16
	12th pkt, 6 s.	144.69	82.12	182.06	80.61
	rand. 0.01, 6 s.	229.15	88.12	291.57	95.55
7	-	411.08	100.00	373.19	97.73
	20th pkt, 6 s.	255.65	99.81	318.72	98.76
	rand. 0.01, 6 s.	309.52	99.42	285.52	98.33
20	-	254.48	94.75	247.84	91.46
	40th pkt, 6 s.	295.14	99.95	216.13	93.05
	rand.prob. 0.01, 6 s.	275.14	98.18	254.00	96.32

6.2 Results of the tests with larger initial congestion window

Using a larger initial congestion window than two segments has been suggested [AFP98, PN98], because it allows faster increase of congestion window and hence benefit when short TCP transfers are used, which is the case with most of the TCP transfers presently done [All00]. The disadvantage of using the larger initial congestion window is that it is likely to increase the level of congestion in the network, especially if several TCP connections are using the larger congestion window size. We expect the increased congestion to cause negative effects also in our tests with multiple parallel connections.

The link *bandwidth*delay* product limiting the amount of data that can be outstanding in the network is approximately 840 bytes (excluding the effect of queueing delay, the round-trip time for a packet is 700 ms), which corresponds to less than three packets. Because of the relatively small capacity of the pipe between the sender and the receiver, we do not expect to have any significant improvement in the TCP performance when increasing the initial congestion window over the original value of two segments. However, we believe that the available bandwidth and the round-trip delay will increase in future e.g. with the GPRS, which makes increasing the initial congestion window a more appealing enhancement. Thus, we are interested about the effects of the increased initial congestion window on the congestion at the last-hop router.

In the test cases with a single TCP connection there were no meaningful differences to the baseline TCP, as predicted above. The test results are presented in Table 43 in Appendix A. Table 7 compares the elapsed time and the number of retransmissions between the baseline TCP and the TCP with initial congestion window of four segments. In some of the test cases there were one more packet dropped at the router when initial congestion window of four segments was used. Presence of the randomly occuring delays did not cause differences to the baseline TCP, either.

Table 7: Comparison of the single connection test results of baseline TCP and the TCP with the initial congestion window of four segments.

		Baseline		**IW = 4**	
Buffer (pkts)	**Delay**	**Time (sec)**	**Rexmits**	**Time (sec)**	**Rexmits**
3	-	105.15	18.00	104.97	19.00
	12th, 6 s.	113.82	24.50	116.54	26.00
	rand. 0.01, 6 s.	130.63	42.00	131.22	39.50
7	-	103.11	17.00	102.92	17.00
	20th, 6 s.	120.31	24.00	109.80	26.00
	rand. 0.01, 6 s.	132.61	47.00	132.69	49.00
20	-	106.40	27.00	106.21	27.00
	40th, 6 s.	131.04	42.00	131.69	42.00
	rand. 0.01, 6 s.	137.42	52.50	136.97	49.00

A notably better throughput is achieved with an additional delay of 6 seconds on the 20th packet and the router buffer size of 7 packets. Reason for this is that the first packet is dropped earlier with the larger initial congestion window than with the baseline TCP, causing the sender to enter fast retransmit before the additional delay occurs, hence avoiding the problem with the NewReno bugfix described with the baseline TCP. Therefore, the improved throughput applies only to this particular test case, and not to the TCP behaviour in general.

When multiple parallel TCP connections are used as the workload, only the tests with the small router buffer introduce serious problems in the TCP performance (see Tables 44 - 47 in Appendix A). With the small router buffer some of the parallel TCP connections occupy the router buffer space, while segments belonging to the other connections are dropped in the beginning of the transmission. Some of the senders will have to wait for the retransmission timeout to expire, while other senders may fill the router buffer space with their own segments. This is the typical lock-out problem presented in Section 3.4.

If a large enough last-hop router buffer is used (a buffer size of 7 packets for two connections or a buffer size of 20 packets for four connections), using the initial congestion window of four segments does not have significant negative effect on the performance. In fact, when using a router buffer size of 7 packets with four parallel TCP connections, a notable improvement is achieved. This is explained by the fact that the corresponding baseline TCP test suffered from serious lock-out problem for one of the TCP connections. Occurances of the lock-out behaviour are very sensitive to small changes in the sender behaviour, thus no further conclusions can be drawn because of this test case.

Table 8 compares the throughput of the baseline TCP and the TCP with the initial congestion window of 4 * MSS. The throughput is measured for the first 10 KB of the transfer of the slowest connection. As expected, larger initial congestion window did not improve the performance in general. However, because the negative effects were not serious, as long as the router buffer was large enough considering the number of parallel TCP connections, this TCP enhancement is worth further inspecting on the connection paths with a higher *bandwidth * delay* product.

Table 8: Comparison of the throughputs of the baseline TCP and the TCP using the initial congestion window of four segments with two parallel connections.

		Baseline		**IW = 4**	
Buffer (pkts)	**Delay**	**Throughput (bytes/sec)**	**Fairness (x 100)**	**Throughput (bytes/sec)**	**Fairness (x 100)**
3	-	491.55	100.00	124.28	65.06
	12th pkt, 6 s.	144.69	82.12	174.78	78.85
	rand. 0.01, 6 s.	229.15	88.12	157.56	70.60
7	-	411.08	100.00	359.31	98.31
	20th pkt, 6 s.	255.65	99.81	147.56	78.04
	rand. 0.01, 6 s.	309.52	99.42	267.29	93.38
20	-	254.48	94.75	231.31	94.20
	40th pkt, 6 s.	295.14	99.95	170.32	89.06
	rand.prob. 0.01, 6 s.	275.14	98.18	254.12	97.42

6.3 Results of the tests with shared advertised window

We have pointed out that one of the main problems in our environment is the severe congestion caused by the slow start overshooting, which causes several segments to be dropped at the last-hop router. The larger the congestion window, the more packet losses there are because of the slow start overshooting. When a large router buffer is used, the number of packets outstanding in the network is large, which results in long queueing delays. Queueing delay inflates the RTT and RTO estimates, which is harmful if a packet loss causes a retransmission timeout. However, if there are additional delays on the link, smaller RTO estimate will increase the probability of spurious timeouts.

Limiting the advertised window should improve the performance, because it does not allow the sender to transmit more data to the network than what is limited by the receiver. If the advertised window is limited to be less than the pipe buffering capacity, the number of buffer overflows should be significantly decreased, if not totally avoided. Moreover, limiting the amount of outstanding data in the network reduces the queueing delays.

The size of the shared advertised window should be related to the pipe buffering capacity. Ideal size would be small enough to totally prevent the buffer from overflowing, but large enough to allow full utilisation of the bottleneck link. However, if the window size is smaller than four segments per connection, the fast retransmit algorithm becomes unusable. We chose shared buffer sizes of 2048 KB (8 * MSS), 3072 KB (12 * MSS) and 5120 KB (20 * MSS). When using a router buffer size of 3 packets, 8 packets are needed to fill the buffers between the sender and the receiver (see Section 5.1), which is equal to what allowed by the smallest advertised window limit given above. Likewise, with a router buffer size of 7 packets the network will be able to contain 12 packets. With the different combinations of the router buffer sizes and the advertised window sizes we can also test the effect of having an advertised window smaller than the pipe buffering capacity, as well as having an advertised window larger than the pipe buffering capacity. Results of the shared advertised window tests for single and parallel connections are shown in Tables 23 - 36 in Appendix A.

Tests with a single TCP connection

Table 9 shows the transfer times and the number of retransmissions for the selected test cases with the different sizes of the advertised window. The row and column headings of the table show the total pipe buffering capacity and the window limit as MSS-sized segments in parenthesis. The delay column shows which packet was delayed and the length of the additional delay. The table shows that least retransmissions are made when the advertised window is smaller than the pipe buffering capacity, because then most of the buffer overflows at the last-hop router can be avoided.

If the advertised window size is smaller than the pipe buffering capacity, the throughput is higher than with the baseline TCP. There are less retransmissions, because the router buffer does not overflow and packets are not dropped. However, there are still

Table 9: Comparison of throughput and number of retransmissions with the different advertised window sizes.

		Baseline		2 KB (8)		3 KB (12)		5 KB (20)	
Buffer	**Delay**	**Time**	**Rxmt**	**Time**	**Rxmt**	**Time**	**Rxmt**	**Time**	**Rxmt**
3 (8)	-	105.15	18	102.03	0	103.36	13	104.26	17
	12 (1)	102.66	16	109.03	10	103.55	15	103.75	16
	12 (6)	113.82	24.5	119.14	18	113.83	24	114.23	25
	30 (6)	114.76	40.5	118.26	18	109.55	17	111.73	27
	rand. (6)	130.63	42	134.47	31.5	132.62	38.5	-	-
7 (12)	-	103.11	17	102.02	0	102.03	0	102.92	12
	20 (1.5)	105.11	16	102.97	0	106.43	5	104.05	14
	20 (6)	120.31	24	111.13	14	111.46	12	120.40	25.5
	40 (10)	116.86	34	114.50	13	117.44	15	119.41	43
	rand. (6)	132.61	47	132.90	35.5	135.81	35	136.08	42.5
20 (25)	-	106.40	27	102.03	0	102.03	0	102.02	0
	40 (6)	131.04	42	110.75	13	112.26	19	111.25	22
	100 (12)	142.28	104	116.75	13	118.01	18	117.26	22
	rand. (6)	137.42	52.5	133.09	34	136.29	42.5	132.25	43

retransmissions which are caused by the spurious retransmission timeouts. Because of the limited advertised window there are less segments in the network, hence the queueing delay is smaller. Additionally, limiting the number of outstanding segments gives an upper bound to the *slow start threshold* value, which is set to half of the number of outstanding segments when the sender observes a packet loss. However, the effect of the congestion control algorithms on the throughput is not so notable as with the baseline TCP, because the advertised window controls the amount of outstanding segments for most of the time.

The buffer overflows cannot always be avoided by limiting the advertised window to be smaller than the pipe capacity. Using the limited advertised window does not help in limiting the number of segments sent as retransmissions. Figure 16 illustrates a situation in which a spurious retransmission timeout occurs because of an additional delay and the sender enters the slow start, starting the retransmissions from the segment which was delayed. Seven segments are retransmitted after the delay, even though the original transmissions are still in the router queue. One of the segments transmitted during the slow start following the retransmission timeout does not fit in the router queue and is therefore dropped. However, the fast retransmit would have been triggered because of the spurious retransmissions even if the segment was not dropped. Therefore, the segment loss does not cause the throughput to be any lower than what it would have been without the segment loss.

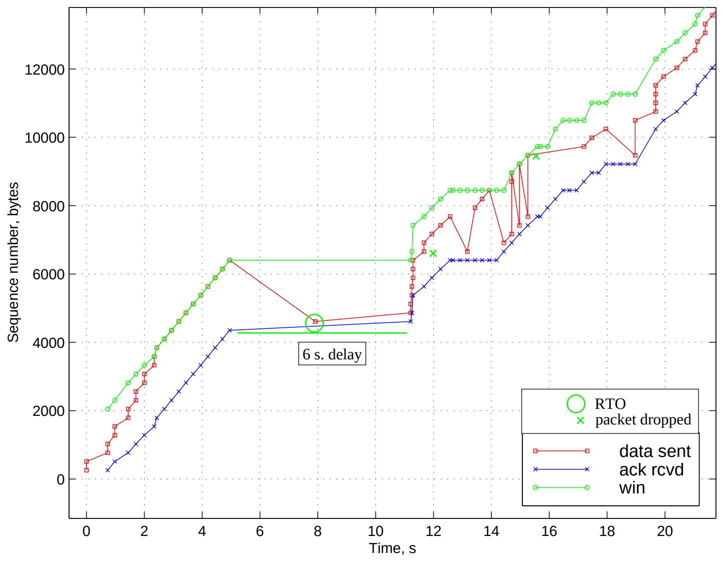

Figure 16: Shared advertised window of 2 KB (8 * MSS) with router buffer size of 7 packets. An additional delay of 6 seconds occurs before the first buffer overflow.

If there were random additional delays on the link, limiting the advertised window to 2 KB improved the performance. The performance was not meaningfully improved with the advertised window limitations of 3 KB and 5 KB. By using a small advertised window the packet losses caused by the router buffer overflow can be avoided, hence there are less retransmissions made. Limiting the advertised window does not usually help in avoiding the unnecessary retransmissions after a spurious retransmission timeout. However, by using a small advertised window the number of unnecessary retransmissions can be reduced.

Limiting the advertised window is a safe enhancement which usually improves the performance, provided that it is done on a path with known bottleneck link characteristics. It is not harmful for the network, but actually makes the sender to behave more conservatively. The highest throughput can be achieved with a relatively small advertised window size, but using too small window is harmful. In addition to the possibility of disabling the fast retransmit algorithm, small advertised window size can limit the amount of the new data transmitted during the fast recovery algorithm.

Tests with multiple parallel TCP connections

In tests with multiple parallel TCP connections the advertised window space is shared between the connections. Table 10 compares the throughput of the baseline TCP and the TCP with a shared advertised window of 2 KB. Throughput of the connection which was last to transfer 10 KB is shown in the table.

Table 10: Comparison of the throughputs of the baseline TCP and the TCP using the shared advertised window of 2 KB with two parallel connections.

		Baseline		**Adv. Wnd 2 KB**	
Buffer (pkts)	**Delay**	**Throughput (bytes/sec)**	**Fairness (x 100)**	**Throughput (bytes/sec)**	**Fairness (x 100)**
3	-	491.55	100.00	461.40	99.54
	12th pkt, 6 s.	144.69	82.12	241.31	87.39
	rand. 0.01, 6 s.	229.15	88.12	376.39	99.41
7	-	411.08	100.00	484.39	99.95
	20th pkt, 6 s.	255.65	99.81	339.96	99.36
	rand. 0.01, 6 s.	309.52	99.42	380.98	99.95
20	-	254.48	94.75	484.37	99.95
	40th pkt, 6 s.	295.14	99.95	345.81	99.97
	rand.prob. 0.01, 6 s.	275.14	98.18	380.89	99.95

When two parallel TCP connections were used, the throughput of the first 10 KB of the transfer was improved, if the shared advertised window size was smaller than the pipe buffering capacity. If an additional delay long enough to trigger a spurious retransmission timeout occured, the throughput with the shared advertised window was significantly better than with the baseline TCP. The fairness between the TCP connections was also better with the shared advertised window. If the shared advertised window was larger than the pipe buffering capacity, the throughput was not better than with the baseline TCP.

As described for the baseline TCP, a combination of an additional delay and a packet loss inflicted severe problems, because the retransmission timeout following the additional delay caused the fast retransmit to be unusable due to the NewReno bugfix. Because there were packets dropped just after the retransmission timeout, one of the senders suffered from another retransmission timeout, in worst case twice. The retransmission timeouts not only caused the throughput to be lower, but were a significant source of unfairness between the TCP connections.

Figure 17 compares an additional delay followed by a packet loss with the baseline TCP and with the shared advertised window. The buffer overflows can be avoided with the shared advertised window, thus the additional retransmission timeouts caused by the NewReno bugfix do not occur. This explains the significant improvement for the test cases with a delay causing the RTO to expire. Additionally, because the packet losses caused by a router buffer overflow can be avoided by using the shared advertised window, the lock-out problem is not present with this TCP enhancement.

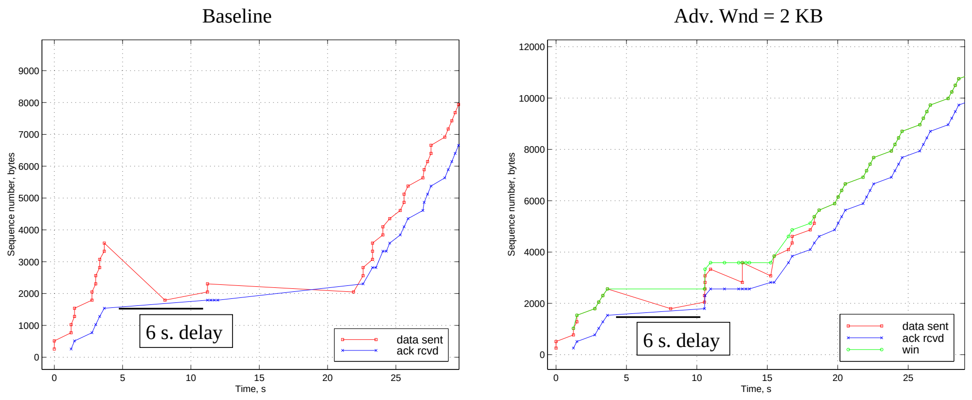

Figure 17: Comparison of the behaviour of baseline TCP and shared advertised window TCP when an additional delay of 6 seconds occurs. Slowest of two parallel TCP connections is shown.

Another factor improving the fairness between the connections is that the shared advertised window causes the number of packets in flight to be limited. Moreover, all the parallel TCP connections have equal amount of segments outstanding in the network. It was noticed in the baseline TCP analysis that the packet separation during the slow start increased the unfairness between the connections. With the shared advertised window, the parallel connections are limited by the same advertised window size, hence they transmit equal number of segments during one round-trip time. This can be seen in Figure 18, which compares the baseline TCP and the TCP with shared advertised window. Four connections are transmitted in parallel, of which all behave similarly when using the shared advertised window.

Shared advertised window significantly improves the fairness and the throughput of the slowest connection also when four parallel TCP connections are used as the workload. Surprisingly, the throughput is improved even with a shared advertised window of 2 KB, although then only 2 segments per each connection are allowed to be outstanding in the network. Therefore, it is impossible to get the three duplicate ACKs required to trigger fast retransmit. Because the packet losses due to router buffer overflow can be avoided by using an appropriately set shared advertised window, there is no need for triggering the fast retransmit. Of course, if the mobile host is communicating through the Internet, packet losses may occur regardless of the advertised window limitation. The throughput was also improved when using the advertised window size of 3 KB and 5 KB, as long as it is smaller than the pipe capacity. Figure 18 compares the startup of the slowest of the four parallel connections with the baseline TCP and with the shared advertised window limited to 5 KB (20 * MSS). The router buffer size is 20 packets in both scenarios, yielding the pipe capacity of 25 packets.

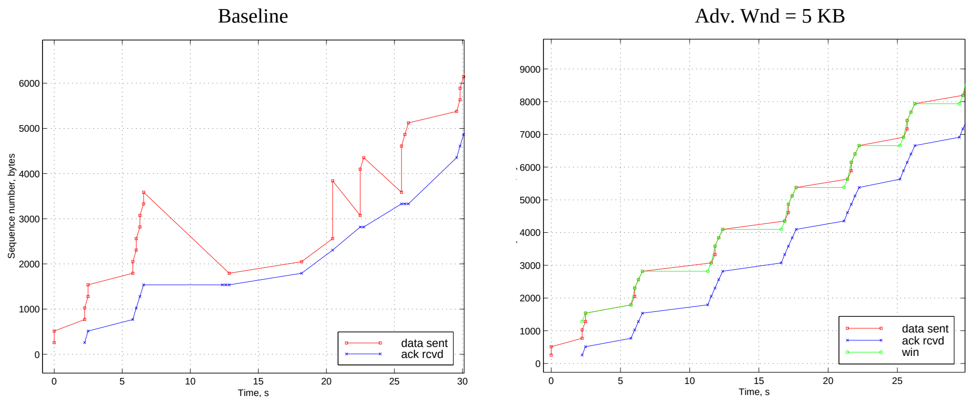

Figure 18: The slowest of the four parallel connections through a router with 20-packet buffer. Comparison of the baseline TCP and a shared advertised window of 5 KB.

6.4 Results of the baseline TCP tests through a RED router

We made a set of tests with the baseline TCP and a last-hop router which uses the *Random Early Detection (RED)* [FJ93, BCC^{+}98] algorithm for dropping the packets instead of the common tail-drop algorithm. Due to its randomness, a RED router should avoid the lock-out problem, which was present in the multiconnection tests made with the tail-drop router. The description of the RED algorithm is given in Section D.3 in Appendix D.

We noticed during our performance tests that it is very difficult to tune the RED parameters to make the last-hop router perform efficiently in our environment. RED has been developed mainly for the routers at the fixed networks with high volumes of traffic, hence it may not be suitable for our environment, in which the routers have a small separated buffer space for each user.

There are four parameters which can be used to tune the RED router for its environment. *Minimum threshold* specifies the average queue length at the router after which the packets may be dropped at a certain probability. *Maximum threshold* determines the upper limit for the average queue length. If the average queue length is higher than the maximum threshold, an incoming packet is dropped. The comparisons are made against the sliding average of the recent queue lengths, thus it is possible to set the maximum threshold to have a smaller value than the router queue capacity. *Queue weight* gives the weight of the current queue length when making the calculations for the average queue size. *Maximum probability* specifies the probability for dropping a packet when the queue length is the same than the maximum threshold. The probability is uniformly scaled depending on the queue length so that when the queue length is close to the minimum threshold, the packet drop probability is close to zero.

We decided to run tests with the RED parameter sets shown in Table 11. The table shows three RED configurations used with the medium and large router buffers. We believe that the RED algorithm is not reasonable with the small router buffer, hence we did not run tests using the router with small buffer. Parameter sets R1 and R2 start dropping packets early, but the packets are not dropped very frequently (max probability is low) and the router does not react very rapidly on the changes in the router queue length (low queue weight). With configuration R3 the router drops the packets more aggressively and is more reactive to the changes in the queue length, but does not start dropping the packets as early as configurations R1 and R2.

Table 11: RED parameters used in performance tests. The table shows minimum threshold, maximum threshold, queue weight and maximum probability chosen.

Buf	**Id**	**min thr.**	**max thr.**	**Q weight**	**max prob.**
7	R1	2	7	0.15	0.3
	R2	1	5	0.1	0.2
	R3	3	6	0.4	0.5
20	R1	7	20	0.15	0.3
	R2	5	16	0.1	0.2
	R3	10	18	0.4	0.5

Tests with a single TCP connection

The test results in Table 38 in Appendix A show that using the RED configurations R1 and R2 causes lower throughput in a 100 KB connection without additional delays than when the tail-drop algorithm was used. The problem with these RED configurations is that the algorithm reacts too slowly to the rapid growth of the router queue length during the slow start. On the other hand, the router is dropping the packets frequently after the slow start, when the queue size starts to increase again. As a result, there are more packets dropped when using RED than with the tail-drop router.

Table 12 compares the elapsed time and the number of retransmissions between the tail-drop router and the RED router with configuration R3, when the baseline TCP is used. Throughput is improved with RED configuration R3 and a router buffer size of 20 packets, when compared to the similar configuration with the tail-drop router. The reason for this is that the first packet is dropped earlier with the RED router than when using the tail-drop router. Because the length of the router queue is smaller, the sender receives the duplicate ACKs resulting from the packet losses earlier and may start retransmitting earlier. As a result, the congestion window does not get as high value as with the tail-drop router, hence there are less packets dropped during the first slow start and the fast recovery can be finished earlier than with the tail-drop router. However, the best throughput can be achieved by having a tail-drop router with a buffer size of 7 packets.

Table 12: Comparison of the single connection test results with the tail-drop router and with the RED router.

		Tail-drop		**RED R3**	
Buffer (pkts)	**Delay**	**Time (sec)**	**Rexmits**	**Time (sec)**	**Rexmits**
7	-	103.11	17.00	104.21	24.50
	20th, 6 s.	120.31	24.00	113.94	32.00
	rand. 0.01, 6 s.	132.61	47.00	137.96	53.50
20	-	106.40	27.00	103.77	23.50
	40th, 6 s.	131.04	42.00	111.62	34.00
	rand. 0.01, 6 s.	137.42	52.50	143.69	68.00

With random additional delays using the RED router resulted in lower throughput with all RED configurations and router buffer sizes tested than when using the tail-drop router with the corresponding buffer size. RED algorithm causes the number of packet losses to be higher, because the RED router drops packets more frequently than the tail-drop router. Moreover, because of the frequent packet losses the average congestion window size with RED is smaller than when using the tail-drop router, which causes smaller RTT and RTO estimates. Therefore the probability for a spurious timeout to be triggered by an additional delay is higher.

The TCP performance is improved with the test cases in which an additional delay occurs during the first slow start, because the RED router drops first packet before the additional delay. Because the three duplicate ACKs arrive to the sender before the additional delay and the corresponding spurious timeout occurs, the NewReno bugfix does not cause another retransmission timeout, as it did with the tail-drop router (see Figure 8 on page 41). However, the throughput improvement concerns only the particular test cases and the problem with the NewReno bugfix would still be present if an additional delay occured for an earlier packet.

Tests with multiple parallel TCP connections

The results of the tests with parallel connections are shown in Tables 39 - 42 in Appendix A. For most of the test cases with two parallel TCP connections using RED with the selected parameter sets does not improve the performance. Reason for the lower throughput is the same what observed with the single connection tests; the RED algorithm causes the packets to be dropped more frequently, which also increases the number of dropped retransmissions. When a retransmission is dropped, the sender will have to wait for the retransmission timeout to trigger the retransmission of the lost segment. Additionally, the retransmission timeouts cause unfairness among the connections, because the other connections can use the link without facing competition with the connection which is waiting for the retransmission timer to expire. However, the lock-out problem could be avoided by using RED.

From the different RED configurations tested, R3 provided the best throughput and least retransmissions. However, excluding the test cases with a selected additional delay location, none of the different combinations of RED configurations and router buffer sizes resulted in a better performance than what was achieved with the baseline TCP with a router buffer size of 7 packets, when two parallel TCP connections were used as a workload. If we restrict the analysis to the test cases with router buffer size of 20 packets, using RED configuration R3 improved the throughput when compared to the baseline TCP, if there was no additional delays. Table 13 compares the throughput and the fairness between the tail-drop router and the RED router. The values are measured for the slowest connection, after the first 10 KB have been transferred.

6.5 Summary

We finish the evaluation of the different TCP enhancements by comparing them with each other on the different test cases made. We briefly summarise the improvements and the

Table 13: Comparison of the throughputs of the tail-drop router and the RED router with two parallel connections.

		Tail-drop		**RED R3**	
Buffer (pkts)	**Delay**	**Throughput (bytes/sec)**	**Fairness (x 100)**	**Throughput (bytes/sec)**	**Fairness (x 100)**
7	-	411.08	100.00	332.44	93.18
	20th pkt, 6 s.	255.65	99.81	260.03	92.15
	rand. 0.01, 6 s.	309.52	99.42	281.75	93.24
20	-	254.48	94.75	302.17	98.06
	40th pkt, 6 s.	295.14	99.95	237.31	95.04
	rand.prob. 0.01, 6 s.	275.14	98.18	259.43	98.49

problems caused the different enhancements and suggest topics for further work.

Throughputs of the different TCP enhancements tested on a link without additional delays are compared in Figure 19. The figure shows that using the last-hop router with a medium buffer size yields the best throughput with the baseline TCP. When the SACK TCP is used, the throughput is better than with the baseline TCP, regardless of the router buffer size used. The throughput is improved because SACK can do several retransmissions during a single round-trip time, whereas the baseline TCP can retransmit only one segment during a round-trip time.

The highest throughput of the tested TCP enhancements can be achieved with the limited advertised window smaller than the pipe capacity between the sender and the receiver. Limiting the advertised window causes the number of packets outstanding in the network to be limited, hence by choosing a small enough advertised window size the packet losses caused by a buffer overflow can be totally avoided. Using the RED algorithm at the router improves the throughput of the baseline TCP when the large router buffer is used, but the throughput is not as good as when using the conventional tail-drop router with the medium buffer size. Using the initial congestion window of four segments results in a slight improvement of throughput.

We inspected the queue length at the last-hop router with a single TCP connection using the different TCP enhancements. Figure 20 illustrates the trace of the router queue length when no additional delays occured on the link. The figure shows a sliding average of the four recent queue length measurements at a router with a buffer size of 7 packets for a 50-second period. The queue length was measured every time a packet arrives to the router (a plot in the figure). When the queue length exceeds seven packets, a packet is

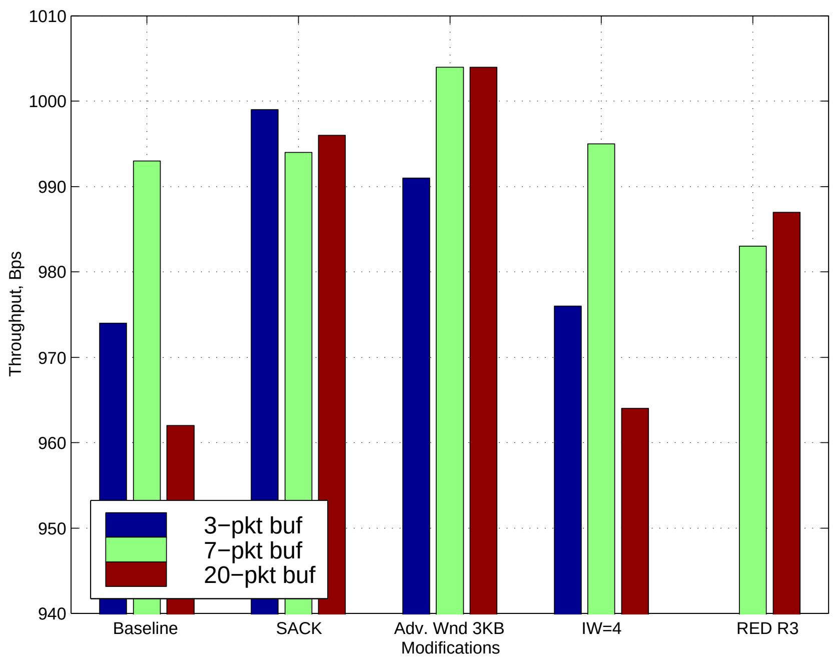

Figure 19: Comparison of the effect of different enhancements and buffer sizes on throughput without additional delay on the link.

dropped at the router.

The different phases of the TCP connection are reflected in the queue load. During the first slow start the queue length grows rapidly. The slow start is finished when several packets are dropped due to slow start overshooting. The dropped packets cause the sender to enter the fast recovery, during which the queue becomes shorter. When the partial ACKs start arriving at the sender, the sender increases the amount of outstanding data by one segment for each round-trip time, which can be seen in the queue trace. A similar policy is also followed by the congestion avoidance algorithm.

Figure 20 shows that although the SACK TCP improves the throughput, it keeps the queue full when retransmitting after the slow start overshooting. For the same reason the SACK TCP causes the number of packets dropped due to overflowing buffers to be higher. The Linux implementation of SACK does not decrease the congestion window after the slow start overshooting, which causes more packet losses during the fast recovery.

The effect of limiting the advertised window can be observed very clearly in Figure 20. When the advertised window limit of 2 KB (8 TCP segments) is used, there are never more than three packets in the router queue. From the eight packets allowed to be outstanding in the network, one is in transit on the connection path between the TCP endpoints and

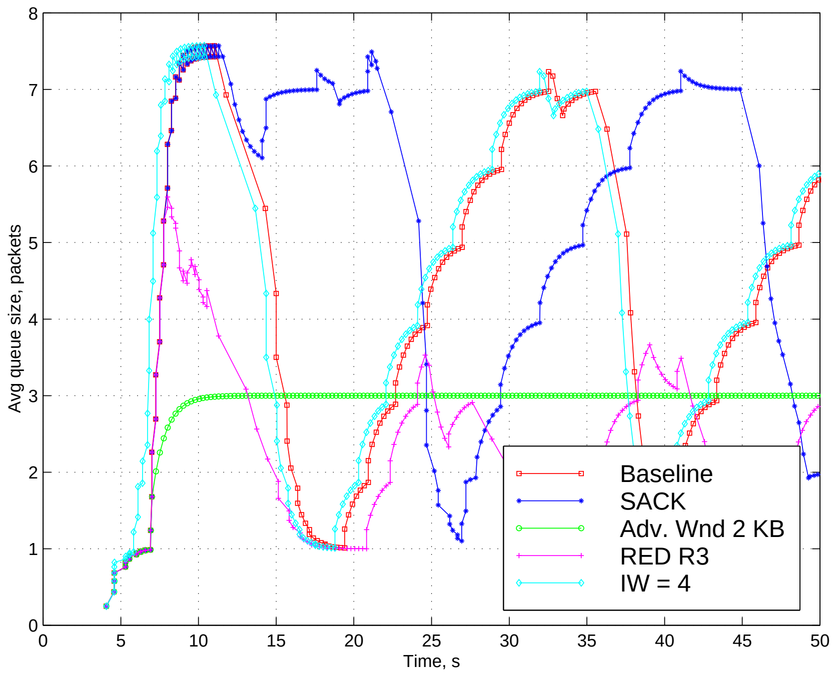

Figure 20: Trace of router queue length with various TCP variations.

four are in the link send buffer. When using the RED algorithm with configuration R3, the router queue is never full because several packets are randomly dropped before the queue is filled up. From Figure 20 we can see that the RED algorithm has the expected effect of keeping the average queue length shorter than the tail-drop router. However, because RED causes the number of packet losses to be higher, the throughput is lower with RED than with the baseline TCP. Inspecting the proper configurations and different active queue management algorithms will be continued as future work.

When there are additional randomly occuring delays on the link, the effect of the TCP enhancements differs from what was observed with the tests without additional delays. Figure 21 compares the achieved throughput with the different TCP enhancements used over a link with additional delays. The figure reveals that there are no significant differences between the different TCP enhancements when random additional delays are present. A small router buffer size yields the highest throughput, because then the queue length and the average congestion window size is smallest, hence there are least unnecessary retransmissions after a spurious timeout.

If multiple parallel TCP connections are transmitted over the bottleneck link, the connections suffer from unfairness when the baseline TCP is used. If one of the connections ceases transmitting for a short period of time, the other connections use all of the link

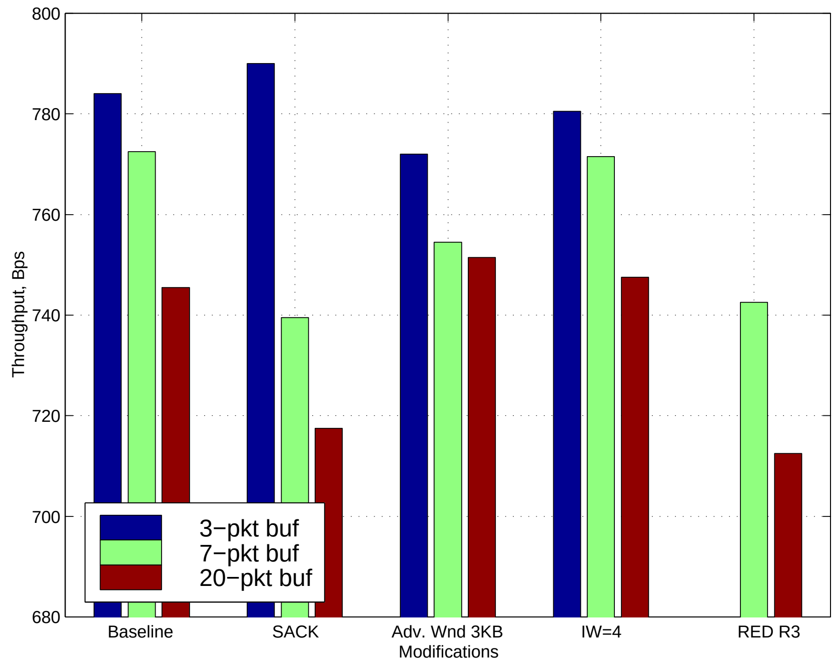

Figure 21: Comparison of the effect of different enhancements and buffer sizes on throughput with additional random delays on the link.

bandwidth, which makes it difficult for the idle connection to restart its transmission. Figure 22 shows the throughput of the slowest of the two parallel TCP connections with the different TCP enhancements. The SACK TCP did not improve the throughput for the connection finished last, because the Linux SACK sender is more aggressive and causes even more congestion at the last-hop router than what the baseline TCP does. Similarly, using the initial congestion window of four segments makes the network congested when several parallel TCP connections are used, hence it does not improve the TCP performance. Using the RED router improves the throughput only in the test cases in which some of the connections suffered from the lock-out problem.

By using the shared advertised window smaller than the pipe capacity results in almost optimal fairness. This reflects also to the throughput of the connection finished last, which is significantly higher than the throughput achieved with the baseline TCP, when inspecting the first 10 KB of the connections. This indicates that it would be a good idea to share some of the connection-specific TCP parameters with other connections using the same link interface, when the last-hop link is the bottleneck. The same applies also in the Internet for sharing the connection state between the connections destined to the same host or to the same subnetwork. This mechanism is called *Control Block Interdependence*, which has been suggested earlier for sharing the round-trip time measurements and the

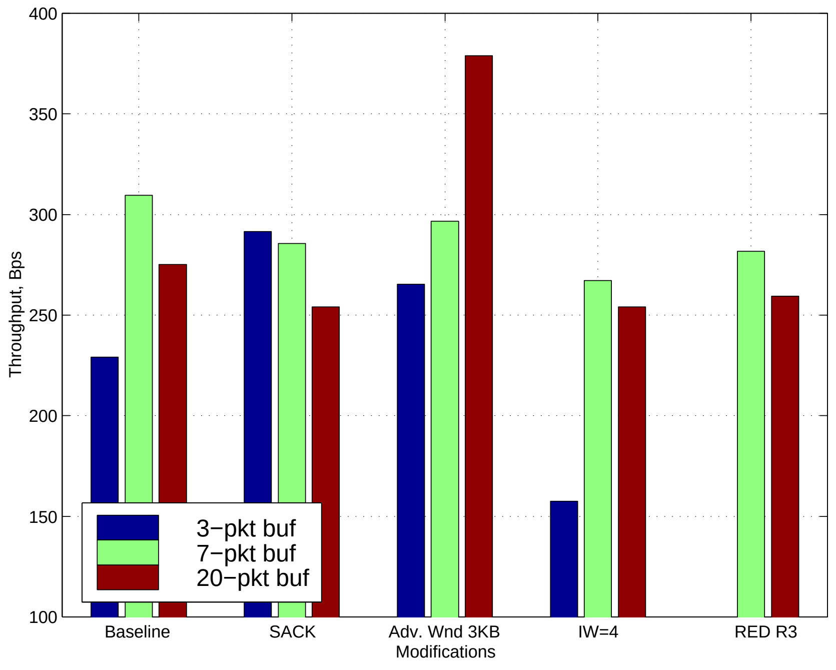

Figure 22: Comparison of the effect of different TCP enhancements and buffer sizes on throughput with additional random delays on the link. Two parallel TCP connections are used as workload.

slow start threshold [Tou97].

7 Conclusion

Our objective was to study the TCP behaviour when it is used over slow wireless links with link-level retransmissions. The two main problems identified in such environment were the spurious retransmission timeouts caused by the variable delays on the wireless link and the heavy congestion at the last-hop router. We presented some of the related studies and suggested improvements concerning these environments.

We made experimental tests using a software emulator for modelling the slow wireless link and the last-hop router. With the emulator we could control the properties of the environment and select specific test scenarios to be repeated with the different TCP implementations. We selected a number of test cases to be made with our baseline TCP implementation. Based on the results we selected the interesting test cases to be tested with the different TCP enhancements. Additionally, we made tests with randomly occuring additional delays to inspect the performance of the different TCP enhancements in an unpredictable natural environment. In addition to the tests with a single TCP connection, we used multiple parallel TCP connections in our tests to monitor the effect of competition of the shared link on the TCP performance.

We found problems in the baseline TCP behaviour when certain events occur on the connection path. If the NewReno bugfix is used, a combination of a spurious retransmission timeout and packet losses results in an additional retransmission timeout. We observed that using a small router buffer usually provides the best results when using a single TCP connection. However, with multiple parallel TCP connections the baseline TCP caused the link bandwidth to be shared unequally, especially if there were additional delays on the link with a small last-hop router buffer.

Three TCP enhancements and the RED active queue management algorithm at the last-hop router were tested and analysed in this work. Using a larger initial congestion window and the RED queue management did not significantly improve the throughput of the TCP connections from what measured with the baseline TCP. By using the SACK TCP the throughput was improved if a single TCP connection was used as the workload and no additional delays occured. Because the Linux SACK sender does not reduce its transmission rate after the packet losses, the last-hop router was severely congested when the SACK TCP was used by multiple parallel TCP connections.

We introduced a mechanism for sharing the advertised window space among the connections running in parallel. By using a shared advertised window smaller than the capacity of the pipe between the sender and the receiver, the throughput is improved from what measured using the baseline TCP with a single TCP connection, if additional delays are not present. If there are multiple parallel TCP connections, the connections get equal share of the link bandwidth and the parallel connections get higher throughput, when compared to the baseline TCP. When the parallel TCP connections use the shared advertised window, the performance is improved also when the additional delays occured on the link.

The additional delays causing the spurious retransmission timeouts are a difficult environment for a single TCP connection using the baseline TCP or the tested TCP enhancements. Although many of the TCP enhancements improved the TCP performance when packet losses due to buffer overflow were present, none of the tested enhancements improved the performance significantly, if there were additional delays on the link.

Because of the retransmission ambiguity, identifying the unnecessary retransmissions at the sending end is impossible with the information available in the basic TCP header. However, by using the TCP timestamps option [LK00] or an enhancement of the SACK algorithm [FMMP00] the unnecessary retransmissions can be recognized at the sending end. This makes it possible to improve the performance after the spurious retransmission timeouts.

In future we intend to enhance the shared advertised window implementation further. Additionally, it will be interesting to inspect different workload models over the slow wireless link. In particular, modelling interactive message exchange resembling the HTTP protocol used in the WWW transfer is important, because it is the prevalent protocol presently used in the Internet. The problems caused by the spurious retransmission timeouts inspire for seeking alternative ways of doing retransmissions after a retransmission timeout. Furthermore, it will be interesting to inspect the effects of D-SACK, which helps in detecting unnecessary retransmissions.

References

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

[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.

[AGS99] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. IETF RFC 2488, January 1999.

[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.

[BB95] Ajay Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th Conference on Distributed Computer Systems*, pages 136–143. IEEE, 1995.

[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.

[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.

[BSK95] H. Balakrishnan, S. Seshan, and R. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *Wireless Networks*, 1(4):471–481, 1995.

[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.

[CG97] J. Cai and D. J. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, pages 122–131, October 1997.

[CI94] R. Caceres and L. Iftode. The effects of mobility on reliable transport protocols. In *14th International Conference on Distributed Computer Systems*, pages 12–20, Poznan, Poland, June 1994. IEEE.

[CLM99] H. M. Chaskar, T. V. Lakshman, and U. Madhow. TCP over wireless with link level error control: Analysis and design methodology. *IEEE/ACM Transactions on Networking*, 7(5):605–615, October 1999.

[Com95] D. E. Comer. *Internetworking with TCP/IP Volume I: Principles, Protocols and Architecture.* Prentice Hall International, third edition, 1995.

[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.

[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.

[FJ92] S. Floyd and V. Jacobson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 3(3):115–156, September 1992.

[FJ93] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgment (SACK) option for TCP. IETF RFC 2883, July 2000.

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

[Hoe95] J. Hoe. Startup dynamics of TCP's congestion control and avoidance schemes. Master's thesis, MIT, 1995. Available at: http://ana-www.lcs.mit.edu/anaweb/ps-papers/hoe-thesis.ps.

[Hoe96] J. Hoe. Improving the start-up behavior of a congestion control scheme for TC P. In *ACM SIGCOMM*, August 1996.

[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.

[JLM97] V. Jacobson, C. Leres, and S. McCanne. `tcpdump`. Available at http://ee.lbl.gov/, June 1997.

[KA98] S. Kent and R. Atkinson. Security architecture for the Internet Protocol. IETF RFC 2401, November 1998.

[KFT$^+$00] P. Karn, A. Falk, J. Touch, M. Montpetit, J. Mahdavi, G. Montenegro, D. Grossman, and G. Fairhurst. Advice for internet subnetwork designers. Internet draft "draft-ietf-pilc-link-design-04.txt", November 2000. Work in progress.

[KP87] P. Karn and C. Partridge. Improving round-trip estimates in reliable transport protocols. In *Proceedings of ACM SIGCOMM '87*, pages 2–7, August 1987.

[KRL$^+$97] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko. An efficient transport service for slow wireless links. *IEEE Journal on Selected Areas In Communications*, 15(7):1337–1348, September 1997.

[Kuh] P. Kuhlberg. Effects of delays and errors on TCP-based wireless data communication. Master's thesis, Department of Computer Science, University of Helsinki. Work in progress.

[LK98] D. Lin and H. Kung. TCP fast recovery strategies: Analysis and improvements. In *IEEE Infocom*. IEEE, March 1998.

[LK00] Reiner Ludwig and Randy H. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communications Review*, 30(1), January 2000.

[LRK$^+$99] Reiner Ludwig, Bela Rathonyi, Almudena Konrad, Kimberly Oden, and Anthony Joseph. Multi-layer tracing of TCP over a reliable wireless link. In *Proceedings of the International conference on Measurement and Modeling of Computer Metrics (ACM SIGMETRICS '99)*, pages 144–154, May 1999.

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

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

[Mat97] MathWorks. `matlab` version 5. See http://www.mathworks.com/, 1997.

[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*, volume 26, October 1996.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. IETF RFC 2018, October 1996. Standards Track.

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

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

[Ost] S. Ostermann. `tcptrace`. Available at: http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html.

[PA00] V. Paxson and M. Allman. Computing TCP's retransmission timer. IETF RFC 2988, November 2000. Standards Track.

[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.

[PF97] V. Paxson and S. Floyd. Why we don't know how to simulate the Internet. In *Proceedings of the 1997 Winter Simulation Conference*, December 1997.

[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, 1981. Standard.

[PS98] S. Parker and Schmechel. Some testing tools for TCP implementors. IETF RFC 2398, August 1998.

[Sim94] W. Simpson. The point-to-point protocol (PPP). IETF RFC 1661, July 1994.

[SM99] R. Semke and J. Mahdavi. The rate-halving algorithm for TCP congestion control. IETF draft draft-mathis-tcp-ratehalving-00.txt, August 1999. Work in progress.

[SMM98] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. *ACM Computer Communication Review*, 28(4), October 1998.

[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.

[SZC90] S. Shenker, L. Zhang, and D. Clark. Some observations on the dynamics of a congestion control algorithm. *ACM Computer Communications Review*, 20(5):30–39, October 1990.

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

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

A Summary of the test results

The summary of all performance tests run are described in this section. Each scenario is tested with 20 replications, except the scenarios with random delays are tested with 50 replications. There are three different kinds of tables.

Single connection tests are done with 100 KB of unidirectional bulk data generated by `ttcp` tool. The tables describing the results of single connection tests show the buffer size and chosen delay identifying the scenario, and for each scenario first quartile (25% percentile), median (50% percentile) and third quartile (75% percentile) of the elapsed time for the connections are shown. Elapsed time is the time between the first SYN sent by the sender and the last packet received (usually a FIN acknowledgement) by the sender. Additionally, median of throughput *(tput)*, number of retransmissions *(rexmt)* and number of packets dropped *(drops)* are shown.

For tests with 2 and 4 parallel connections there are two tables shown for each test. First of the tables shows the performance of the whole 50 KB connections, and the second shows the results for the first 10 KB of the same connections. The connections are unidirectional bulk data generated by `ttcp` processes launched within 100 ms of each other (in other words, almost simultaneously). The table shows elapsed time for the slowest and the fastest of the connections started in parallel. First quartile, median and third quartile of 20 replications are shown. Additionally, median of the number of packets retransmitted are shown separately for the connection with least retransmissions *(RxMin)* and the connection with most retransmissions *(RxMax)*.

Table 14: Baseline single connection tests, 20 replications (random delays tested with 50 replications)

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
1	-	104.43	104.45	104.50	980.00	22.00	22.00
	9th, 6 s.	114.90	114.94	116.00	891.00	29.00	23.00
3	-	105.14	105.15	105.17	974.00	18.00	18.00
	12th, 1 s.	102.65	102.66	102.67	997.00	16.00	16.00
	12th, 6 s.	113.80	113.82	114.22	900.00	24.50	19.00
	30th, 6 s.	114.25	114.76	114.91	892.50	40.50	27.00
	12th, 10 s.	123.94	124.01	124.12	826.00	25.00	18.00
	30th & 50th, 10 s.	125.95	126.10	126.49	812.00	38.50	24.50
	rand. 0.01, 6 s.	120.74	130.63	138.29	784.00	42.00	25.50
7	-	103.10	103.11	103.12	993.00	17.00	17.00
	20th, 1.5 s.	104.24	105.11	105.47	974.00	16.00	16.00
	20th, 6 s.	120.19	120.31	120.42	851.00	24.00	16.00
	40th, 10 s.	116.03	116.86	119.18	876.50	34.00	20.00
	300th, 6 s.	112.51	112.52	112.53	910.00	32.00	21.00
	20th, 15 s.	140.08	140.30	141.01	730.00	24.50	16.50
	40th & 80th, 15 s.	137.20	137.93	139.40	742.00	42.00	22.00
	rand. 0.01, 6 s.	120.69	132.61	142.01	772.50	47.00	23.50
20	-	106.39	106.40	106.41	962.00	27.00	27.00
	40th, 4 s.	109.74	109.80	109.83	933.00	25.00	25.00
	40th, 6 s.	129.74	131.04	131.64	781.00	42.00	27.00
	100th, 12 s.	140.32	142.28	142.47	720.00	104.00	52.00
	300th, 6 s.	116.82	117.60	117.63	871.00	51.00	29.00
	40th, 30 s.	176.56	178.73	178.93	573.00	41.00	21.00
	100th & 250th, 20 s.	162.76	164.06	164.13	624.00	138.00	52.00
	rand. 0.01, 6 s.	121.99	137.42	141.94	745.50	52.50	26.00

Table 15: Baseline tests, 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	101.81	101.81	101.82	102.24	102.25	102.25	15.00	15.00
	12th, 1 s.	78.98	87.45	96.05	103.62	105.00	105.54	14.50	23.50
	12th, 6 s.	79.06	87.78	91.52	115.08	117.02	118.36	17.00	24.00
	30th, 6 s.	84.58	91.79	102.80	110.87	111.20	114.15	18.00	28.50
	rand. 0.01, 6 s.	92.15	100.59	109.87	122.37	133.69	148.16	25.00	33.00
7	-	99.72	99.73	99.74	102.30	102.30	102.32	9.00	14.00
	20th, 6 s.	92.10	114.61	117.72	116.86	118.48	118.61	15.00	20.00
	40th, 10 s.	96.16	96.38	105.16	114.40	115.98	116.60	15.00	24.00
	rand. 0.01, 6 s.	99.39	105.74	126.13	117.87	130.15	139.39	21.50	30.00
20	-	91.47	91.47	91.48	102.70	102.70	102.71	11.00	17.00
	40th, 4 s.	95.61	95.79	96.24	105.55	106.05	106.84	10.00	18.00
	40th, 6 s.	99.24	100.50	102.78	108.87	109.02	109.21	17.00	17.50
	100th, 12 s.	104.49	105.42	108.27	114.72	115.06	120.20	17.00	17.00
	rand. 0.01, 6 s.	110.71	124.34	135.58	122.81	134.79	145.37	20.00	29.50

Table 16: Baseline tests, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	21.12	21.12	21.12	21.36	21.36	21.37	5.00	6.00
	12th, 1 s.	17.83	17.84	18.61	33.92	35.49	40.75	2.00	12.00
	12th, 6 s.	25.29	26.22	26.24	63.99	71.30	72.74	8.00	12.00
	30th, 6 s.	25.24	25.31	25.52	36.43	41.45	44.20	7.00	18.00
	rand. 0.01, 6 s.	18.47	19.16	25.44	39.03	45.22	56.73	5.00	14.50
7	-	25.14	25.15	25.16	25.41	25.41	25.42	5.00	9.00
	20th, 6 s.	33.30	37.20	37.27	40.50	40.55	47.72	9.00	15.00
	40th, 10 s.	33.28	33.60	34.10	34.79	35.62	35.67	8.00	18.00
	rand. 0.01, 6 s.	23.82	24.84	29.98	25.97	33.69	51.35	7.00	9.00
20	-	25.40	25.41	25.41	40.77	40.77	40.78	5.00	10.00
	40th, 4 s.	28.91	29.16	30.21	38.63	40.50	41.75	5.00	9.00
	40th, 6 s.	31.34	33.41	33.48	34.18	35.19	35.20	5.00	16.00
	100th, 12 s.	24.64	24.71	37.23	43.14	43.45	46.86	5.00	15.00
	rand. 0.01, 6 s.	25.09	27.21	34.95	35.12	37.76	45.59	6.00	9.50

Table 17: Baseline tests, 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	109.46	149.07	164.03	206.38	207.52	208.58	20.50	40.50
	12th, 6 s.	160.36	175.53	189.64	220.15	221.19	224.24	35.00	47.50
	rand. 0.01, 6 s.	168.22	192.26	207.47	250.79	264.55	274.53	30.50	48.00
7	-	155.62	156.64	162.07	203.59	204.21	205.01	15.00	22.00
	20th, 6 s.	128.90	146.33	181.46	217.69	218.50	219.47	16.50	31.00
	rand. 0.01, 6 s.	137.17	165.60	200.78	250.48	264.83	284.26	27.00	42.00
20	-	171.20	171.21	171.22	203.64	203.65	203.65	10.00	18.00
	40th, 6 s.	176.85	186.70	192.66	210.60	210.78	210.90	12.00	17.00
	rand. 0.01, 6 s.	207.94	223.70	237.72	249.64	262.07	271.86	19.50	40.00

Table 18: Baseline tests, First 10 KB of 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	17.77	21.20	24.70	90.02	108.43	149.33	5.00	14.00
	12th, 6 s.	28.32	31.54	32.83	68.95	86.87	99.21	6.00	14.50
	rand. 0.01, 6 s.	29.45	32.89	38.49	86.65	140.38	198.12	6.50	15.00
7	-	24.30	24.30	24.31	110.03	137.59	140.91	4.00	13.00
	20th, 6 s.	24.29	24.72	24.76	93.48	106.51	132.79	2.00	12.00
	rand. 0.01, 6 s.	30.69	34.22	39.23	84.28	104.44	144.66	3.50	15.00
20	-	39.13	39.15	39.16	61.08	61.08	61.09	5.00	9.00
	40th, 6 s.	42.39	42.56	42.88	57.45	57.71	62.37	7.00	10.00
	rand. 0.01, 6 s.	41.69	47.01	51.04	60.37	65.97	72.57	4.00	11.00

Table 19: Single connection tests with SACK, 20 replications (random delays tested with 50 replications).

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
3	-	102.48	102.49	102.49	999.00	20.00	19.00
	12th, 1 s.	102.47	102.48	102.48	999.00	16.00	16.00
	12th, 6 s.	109.20	109.20	109.22	938.00	23.00	17.00
	30th, 6 s.	109.71	109.72	109.73	933.00	29.00	23.00
	rand. 0.01, 6 s.	120.75	129.62	140.00	790.00	43.00	26.00
7	-	103.04	103.04	103.05	994.00	26.00	22.00
	20th, 1.5 s.	104.85	104.92	104.93	976.00	15.00	15.00
	20th, 6 s.	110.81	111.80	111.90	916.00	24.00	16.00
	40th, 10 s.	113.69	113.70	113.71	901.00	34.00	27.00
	300th, 6 s.	122.53	122.77	122.86	834.00	31.00	23.00
	20th, 15 s.	118.43	118.44	118.45	865.00	19.00	15.00
	40th & 80th, 6 s.	133.08	135.08	135.38	758.00	35.50	28.00
	rand. 0.01, 6 s.	129.40	138.47	144.94	739.50	44.50	22.00
20	-	102.79	102.79	120.38	996.00	32.00	29.00
	40th, 4 s.	105.48	106.24	106.39	964.00	27.00	27.00
	40th, 6 s.	118.00	118.32	118.83	865.00	71.00	28.00
	100th, 12 s.	125.41	125.42	125.43	816.00	28.00	28.00
	300th, 6 s.	111.90	129.59	129.60	790.00	45.00	29.00
	100th & 250th, 20 s.	150.15	150.16	150.33	682.00	49.00	28.00
	rand. 0.01, 6 s.	127.85	142.74	148.84	717.50	64.00	27.00

Table 20: SACK tests, 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	95.22	96.79	97.03	102.88	103.07	104.05	26.00	31.00
	12th, 6 s.	90.58	93.94	99.20	109.86	109.99	110.83	18.00	25.00
	rand. 0.01, 6 s.	97.96	108.51	125.76	116.45	130.53	138.72	31.50	37.00
7	-	99.11	99.11	99.16	102.92	102.92	102.93	16.00	26.00
	20th, 6 s.	101.34	102.31	106.28	109.33	109.75	109.98	13.00	21.00
	rand. 0.01, 6 s.	105.45	116.89	128.08	122.92	130.46	139.64	24.50	38.00
20	-	82.22	82.23	84.24	105.94	105.95	105.95	21.00	44.00
	40th, 6 s.	72.75	77.08	82.89	107.89	108.74	109.18	17.00	19.50
	rand. 0.01, 6 s.	84.81	107.05	137.17	123.92	132.61	143.72	29.00	39.50

Table 21: SACK tests, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	20.18	20.18	20.18	26.47	26.47	26.48	10.00	14.00
	12th, 6 s.	19.73	19.73	19.74	56.55	56.79	56.80	8.00	9.00
	rand. 0.01, 6 s.	19.69	20.67	30.69	30.49	35.64	44.40	11.00	13.00
7	-	20.68	20.68	20.79	27.98	27.99	28.14	7.00	17.00
	20th, 6 s.	26.13	26.39	28.14	29.39	32.67	32.68	7.00	14.00
	rand. 0.01, 6 s.	24.09	30.29	32.41	33.28	36.39	43.68	9.50	14.00
20	-	22.48	22.48	22.48	41.86	41.86	41.87	16.00	16.00
	40th, 6 s.	27.42	27.42	27.70	47.03	47.94	54.86	15.00	18.50
	rand. 0.01, 6 s.	22.41	27.41	35.08	36.35	40.89	59.36	11.50	17.00

Table 22: SACK tests, First 10 KB of 4 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	rand. 0.01, 6 s.	24.27	28.30	33.77	120.69	197.07	217.49	10.00	20.00
7	rand. 0.01, 6 s.	28.91	34.63	43.02	82.56	100.21	163.35	8.50	14.50
20	rand. 0.01, 6 s.	38.87	39.42	47.21	64.87	73.19	83.98	7.00	16.50

Table 23: Single connection tests with shared advertised window of 2 KB, 20 replications.

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
3	-	102.02	102.03	102.03	1004.00	0.00	0.00
	12th, 1 s.	108.97	109.03	109.03	939.00	10.00	10.00
	12th, 6 s.	119.12	119.14	119.17	859.50	18.00	14.00
	30th, 6 s.	117.90	118.26	118.38	866.00	18.00	12.00
	rand. 0.01, 6 s.	122.17	134.47	144.09	761.50	31.50	17.00
7	-	102.02	102.02	102.03	1004.00	0.00	0.00
	20th, 1.5 s.	102.97	102.97	102.98	994.00	0.00	0.00
	20th, 6 s.	111.13	111.13	111.15	921.00	14.00	2.00
	40th, 10 s.	114.50	114.50	114.50	894.00	13.00	1.00
	rand. 0.01, 6 s.	119.59	132.90	140.45	771.00	35.50	3.00
20	-	102.02	102.03	102.03	1004.00	0.00	0.00
	40th, 6 s.	110.75	110.75	110.76	925.00	13.00	0.00
	100th, 12 s.	116.75	116.75	116.76	877.00	13.00	0.00
	rand. 0.01, 6 s.	119.79	133.09	141.10	770.00	34.00	0.00

Table 24: Shared advertised window of 2 KB. 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	100.60	100.60	100.61	102.06	102.06	102.07	0.00	1.00
	12th, 6 s.	94.51	95.52	102.98	108.86	108.97	109.35	13.00	15.00
	rand. 0.01, 6 s.	102.54	110.41	123.72	115.30	129.61	137.57	18.50	25.50
7	-	101.37	101.37	101.38	101.86	101.87	101.87	13.00	15.00
	20th, 6 s.	107.82	107.83	108.57	109.52	109.53	110.27	5.00	6.00
	rand. 0.01, 6 s.	116.80	127.28	137.06	118.16	131.69	138.50	16.50	20.50
20	-	101.37	101.37	101.37	101.86	101.87	101.87	5.00	6.00
	40th, 6 s.	110.59	110.60	110.61	110.84	110.85	110.86	7.00	7.00
	rand. 0.01, 6 s.	122.34	134.86	143.78	123.02	135.57	144.02	20.50	26.50

Table 25: Shared advertised window of 2 KB, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	19.91	19.91	19.92	22.69	22.69	22.70	0.00	1.00
	12th, 6 s.	19.36	19.42	21.46	34.57	42.99	44.05	5.00	8.50
	rand. 0.01, 6 s.	20.46	20.69	21.80	22.92	27.74	41.06	2.00	4.00
7	-	20.68	20.68	20.68	21.68	21.68	21.69	2.00	4.00
	20th, 6 s.	26.13	26.13	26.88	30.65	30.66	31.41	5.00	6.00
	rand. 0.01, 6 s.	20.68	21.94	29.05	21.67	27.40	35.41	5.00	6.00
20	-	20.68	20.68	20.68	21.68	21.68	21.69	5.00	6.00
	40th, 6 s.	29.15	29.15	29.16	30.15	30.15	30.16	7.00	7.00
	rand. 0.01, 6 s.	20.68	23.36	34.78	21.66	27.41	35.96	7.00	7.00

Table 26: Shared advertised window of 2 KB. 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	198.84	198.85	198.87	204.75	204.76	204.78	1.00	2.00
	12th, 6 s.	199.77	201.61	204.21	214.95	215.12	215.25	12.00	19.00
7	-	198.48	198.48	198.50	203.42	203.44	203.45	0.00	1.00
	20th, 6 s.	206.30	206.59	206.60	216.23	216.39	216.51	3.00	7.00
20	-	200.24	200.25	200.26	204.44	204.44	204.46	3.00	7.00
	40th, 6 s.	207.71	207.71	207.73	211.92	211.92	211.94	2.00	2.00

Table 27: Shared advertised window of 2 KB, First 10 KB of 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	35.99	36.00	36.00	53.14	53.15	53.16	1.00	3.00
	12th, 6 s.	34.96	34.99	38.24	64.78	70.79	73.31	1.00	9.00
	rand. 0.01, 6 s.	32.54	36.57	42.81	54.39	61.49	69.33	2.50	8.00
7	-	37.14	37.14	37.15	45.94	45.96	45.96	0.00	1.00
	20th, 6 s.	38.05	38.05	38.07	66.99	67.01	67.03	0.00	4.00
	rand. 0.01, 6 s.	38.65	40.10	49.84	51.22	53.07	66.35	1.50	4.00
20	-	38.90	38.91	38.91	42.63	42.65	42.66	1.50	4.00
	40th, 6 s.	46.36	46.37	46.38	50.09	50.10	50.12	2.00	2.00
	rand. 0.01, 6 s.	38.95	49.73	54.08	48.05	53.47	58.54	2.00	2.00

Table 28: Shared advertised window of 3 KB. Single connection, 20 replications.

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
3	-	103.35	103.36	103.36	991.00	13.00	13.00
	12th, 1 s.	103.55	103.55	103.55	989.00	15.00	15.00
	12th, 6 s.	113.81	113.83	114.22	900.00	24.00	18.00
	30th, 6 s.	109.31	109.55	109.57	935.00	17.00	16.00
	rand. 0.01, 6 s.	118.32	132.62	142.01	772.00	38.50	20.00
7	-	102.03	102.03	102.04	1004.00	0.00	0.00
	20th, 1.5 s.	106.42	106.43	106.46	962.00	5.00	5.00
	20th, 6 s.	111.44	111.46	111.48	919.00	12.00	9.00
	40th, 10 s.	117.44	117.44	117.45	872.00	15.00	5.00
	rand 0.01, 6 s.	121.13	135.81	142.02	754.50	35.00	13.00
20	-	102.02	102.03	102.03	1004.00	0.00	0.00
	40th, 6 s.	112.26	112.26	112.27	912.00	19.00	0.00
	100th, 12 s.	118.01	118.01	118.02	868.00	18.00	0.00
	rand. 0.01, 6 s.	122.50	136.29	148.67	751.50	42.50	0.00

Table 29: Shared advertised window of 3 KB. 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	100.30	100.31	100.32	102.15	102.16	102.17	12.00	13.00
	12th, 6 s.	67.88	70.26	85.27	113.86	131.36	165.87	9.00	17.50
	rand. 0.01, 6 s.	82.02	98.56	119.50	132.38	137.21	141.98	21.50	30.00
7	-	96.83	96.83	96.84	101.90	101.91	101.92	0.00	1.00
	20th, 6 s.	83.85	83.86	84.12	109.33	109.34	109.35	8.00	10.00
	rand. 0.01, 6 s.	99.25	108.01	117.26	117.80	130.13	138.30	13.50	25.50
20	-	101.37	101.37	101.38	101.86	101.87	101.88	8.00	10.00
	40th, 6 s.	111.60	111.61	111.61	112.35	112.36	112.37	10.00	10.00
	rand. 0.01, 6 s.	122.80	133.49	149.10	123.08	134.64	149.57	20.00	26.00

Table 30: Shared advertised window of 3 KB, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	19.36	19.36	19.37	23.13	23.14	23.15	3.00	5.00
	12th, 6 s.	21.06	26.23	26.26	54.92	89.26	123.96	8.00	11.00
	rand. 0.01, 6 s.	19.61	20.36	27.18	23.31	39.11	65.78	5.00	9.00
7	-	16.39	16.39	16.40	26.46	26.47	26.47	0.00	1.00
	20th, 6 s.	21.56	21.56	21.61	43.22	43.23	43.24	8.00	8.00
	rand. 0.01, 6 s.	16.39	16.40	28.04	29.47	35.04	43.10	0.00	7.50
20	-	20.17	20.18	20.18	21.93	21.93	21.94	0.00	7.50
	40th, 6 s.	30.66	30.66	30.66	32.42	32.42	32.43	10.00	10.00
	rand. 0.01, 6 s.	20.17	23.09	35.50	21.92	27.55	37.30	10.00	10.00

Table 31: Shared advertised window of 3 KB. 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	122.05	138.69	150.91	203.94	204.79	206.44	6.50	24.50
	12th, 6 s.	116.57	136.77	153.16	216.60	217.97	235.12	9.00	29.50
	rand. 0.01, 6 s.	155.83	164.67	191.05	248.03	259.50	275.11	21.00	38.50
7	-	193.51	193.52	193.58	204.32	204.32	204.38	1.00	3.00
	20th, 6 s.	186.28	191.35	193.43	220.81	221.04	221.29	6.00	13.00
	rand. 0.01, 6 s.	200.63	213.11	228.27	244.64	254.76	269.24	16.50	30.50
20	-	198.73	198.74	198.93	202.77	202.78	202.96	6.00	13.00
	40th, 6 s.	204.18	204.18	204.18	208.21	208.21	208.22	6.00	13.00
	rand. 0.01, 6 s.	378.79	390.90	416.43	380.30	396.50	418.76	106.00	162.50

Table 32: Shared advertised window of 3 KB, First 10 KB of 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	21.58	24.54	29.00	67.53	108.83	145.52	2.00	12.50
	12th, 6 s.	28.72	29.23	30.65	110.69	147.24	164.07	2.00	16.00
	rand. 0.01, 6 s.	24.36	30.70	35.28	94.60	122.03	144.62	4.00	15.00
7	-	30.65	30.66	30.67	50.01	50.02	50.04	1.00	3.00
	20th, 6 s.	37.49	42.57	42.60	93.71	100.30	105.40	1.00	6.00
	rand. 0.01, 6 s.	29.41	35.34	38.50	72.20	89.21	117.22	2.00	9.50
20	-	35.88	35.88	35.91	43.90	43.91	43.94	2.00	9.50
	40th, 6 s.	41.32	41.33	41.33	49.35	49.35	49.36	2.00	9.50
	rand. 0.01, 6 s.	35.91	53.73	57.88	49.47	65.81	71.92	2.00	9.50

Table 33: Shared advertised window of 5 KB. Single connection, 20 replications.

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
3	-	104.25	104.26	104.28	982.00	17.00	17.00
	12th, 1 s.	103.75	103.75	103.77	987.00	16.00	16.00
	12th, 6 s.	113.85	114.23	114.35	896.00	25.00	19.00
	30th, 6 s.	110.95	111.73	111.98	916.00	27.00	20.00
7	-	102.91	102.92	102.93	995.00	12.00	12.00
	20th, 1.5 s.	103.86	104.05	104.06	984.00	14.00	14.00
	20th, 6 s.	120.32	120.40	120.46	850.50	25.50	16.50
	40th, 10 s.	118.01	119.41	134.38	857.50	43.00	22.00
	rand. 0.01, 6 s.	124.08	136.08	140.55	753.00	42.50	20.50
20	-	102.02	102.02	102.03	1004.00	0.00	0.00
	40th, 6 s.	111.25	111.25	111.32	920.00	22.00	7.00
	100th, 12 s.	117.25	117.26	117.45	873.00	22.00	7.00
	rand. 0.01, 6 s.	122.34	132.25	146.51	775.50	43.00	7.00

Table 34: Shared advertised window of 5 KB. 2 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	97.46	97.47	97.48	102.53	102.55	102.56	16.00	17.00
	12th, 6 s.	70.11	81.63	86.19	116.27	117.50	118.85	15.00	19.50
7	-	98.46	98.47	98.49	102.29	102.30	102.31	8.00	10.00
	20th, 6 s.	66.08	108.97	117.09	113.69	118.22	118.39	14.00	20.50
	rand. 0.01, 6 s.	93.77	114.17	129.65	116.96	131.79	141.44	20.00	27.00
20	-	100.36	100.36	100.36	101.86	101.87	101.88	14.00	20.50
	40th, 6 s.	97.74	97.74	97.74	109.87	109.87	109.87	0.00	12.00
	rand. 0.01, 6 s.	119.53	126.55	141.86	122.18	132.13	144.14	24.00	26.00

Table 35: Shared advertised window of 5 KB, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	17.59	17.59	17.61	29.68	29.69	29.70	5.00	9.00
	12th, 6 s.	26.21	26.24	26.25	70.78	71.43	76.28	8.00	10.00
7	-	19.42	19.42	19.42	22.94	22.94	22.95	3.00	4.00
	20th, 6 s.	23.04	37.19	37.27	40.55	40.57	73.32	9.00	15.00
	rand. 0.01, 6 s.	18.10	18.12	28.17	23.90	33.33	49.38	4.00	6.50
20	-	19.67	19.67	19.67	22.18	22.18	22.19	4.00	6.50
	40th, 6 s.	26.27	26.27	26.28	35.18	35.19	35.19	0.00	12.00
	rand. 0.01, 6 s.	19.67	19.75	35.01	22.19	27.79	36.75	0.00	12.00

Table 36: Shared advertised window of 5 KB. 4 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	86.99	101.21	118.52	206.03	207.33	208.10	10.00	25.00
	12th, 6 s.	138.72	173.63	188.76	219.87	221.06	222.04	32.00	45.50
7	-	117.67	119.59	119.60	205.70	205.95	205.99	3.00	22.00
	20th, 6 s.	120.30	137.24	150.75	222.44	224.84	236.42	7.50	26.00
	rand. 0.01, 6 s.	230.00	238.56	250.16	246.19	254.33	270.16	13.50	21.00
20	-	199.70	199.71	199.87	202.92	202.93	203.12	0.00	1.00
	40th, 6 s.	209.97	210.65	210.98	211.25	211.81	211.99	1.00	8.00
	rand. 0.01, 6 s.	251.43	257.92	263.77	262.08	267.95	279.71	21.00	36.00

Table 37: Shared advertised window of 5 KB. 4 parallel connections, analysis of the first 10 KB.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	16.16	16.60	16.61	133.85	142.46	156.15	3.00	14.00
	12th, 6 s.	29.39	30.49	31.27	78.60	93.76	103.29	6.00	14.00
7	-	23.78	23.78	23.79	147.08	147.10	147.16	2.00	17.00
	20th, 6 s.	32.40	36.57	38.08	119.65	166.82	195.84	3.00	12.00
	rand. 0.01, 6 s.	40.19	45.78	52.14	46.71	53.10	70.95	3.00	12.00
20	-	37.67	37.68	37.69	45.97	45.98	46.01	0.00	1.00
	40th, 6 s.	47.95	48.56	48.96	50.86	51.76	51.86	1.00	8.00
	rand. 0.01, 6 s.	38.15	47.83	52.18	47.70	57.62	64.04	0.00	7.50

Table 38: Baseline TCP with RED router, single connections, 20 replications.

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
7	- (R1)	106.27	111.51	112.21	918.50	41.50	36.50
	20th, 6 s. (R1)	118.70	119.57	120.13	856.50	35.00	31.00
	- (R2)	111.68	114.91	119.87	891.50	49.50	39.50
	- (R3)	103.10	104.21	105.06	983.00	24.50	24.50
	20th, 6 s. (R3)	110.20	113.94	116.57	898.50	32.00	26.00
	rand. 0.01, 6 s. (R2)	129.50	145.74	152.61	702.50	61.00	37.50
	rand. 0.01, 6 s. (R3)	124.34	137.96	143.12	742.50	53.50	30.50
20	- (R1)	108.60	117.58	123.52	871.00	73.50	32.00
	40th, 6 s. (R1)	109.81	118.54	122.80	864.00	30.50	22.00
	- (R2)	104.96	113.11	121.76	905.50	40.50	30.00
	40th, 6 s. (R2)	114.27	127.22	129.68	805.00	38.50	27.50
	- (R3)	103.31	103.77	109.85	987.00	23.50	23.50
	40th, 6 s. (R3)	109.29	111.62	121.61	917.00	34.00	21.50
	rand. 0.01, 6 s. (R2)	133.77	146.42	159.18	699.00	66.00	30.00
	rand. 0.01, 6 s. (R3)	134.76	143.69	152.44	712.50	68.00	27.50

Table 39: Baseline TCP with RED router, 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
7	- (R1)	83.73	89.14	98.17	103.49	104.59	106.30	18.50	26.50
	20th, 6 s. (R1)	96.72	107.48	110.60	117.40	118.96	120.72	22.50	29.00
	- (R2)	88.33	97.44	102.89	105.16	106.84	109.09	20.50	30.00
	- (R3)	80.92	89.91	97.93	102.87	103.77	104.91	17.00	22.00
	20th, 6 s. (R3)	90.07	102.95	106.39	110.01	111.14	116.41	22.50	26.00
	rand. 0.01, 6 s. (R2)	104.05	117.83	129.15	127.44	138.91	145.52	31.00	41.50
	rand. 0.01, 6 s. (R3)	96.63	107.73	117.72	121.32	133.59	140.37	25.00	34.00
20	- (R1)	91.41	97.53	100.99	103.18	106.93	109.23	15.50	33.00
	40th, 6 s. (R1)	81.69	89.71	97.77	109.59	110.41	112.26	14.50	21.00
	- (R2)	75.74	89.63	96.51	103.70	107.12	108.91	17.50	32.00
	40th, 6 s. (R2)	87.58	97.36	106.50	109.88	110.59	114.51	18.50	23.50
	- (R3)	78.19	88.39	96.04	101.92	102.42	103.29	12.50	17.00
	40th, 6s. (R3)	79.39	91.18	105.16	109.49	109.87	112.05	15.00	22.50
	rand. 0.01, 6 s. (R2)	107.98	123.09	129.78	127.90	135.16	142.59	29.50	38.00
	rand. 0.01, 6 s. (R3)	106.28	115.97	125.56	121.06	130.98	139.06	24.00	37.00

Table 40: Baseline TCP with RED router. 2 parallel connections, analysis of the first 10 KB.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
7	- (R1)	19.16	21.03	23.85	27.34	30.47	39.12	6.00	11.50
	20th, 6 s. (R1)	30.65	35.62	36.93	37.77	38.49	43.07	10.00	14.50
	- (R2)	16.59	20.13	21.44	26.96	31.66	40.94	5.00	12.50
	- (R3)	16.92	19.37	20.24	22.82	31.34	40.17	6.00	9.50
	20th, 6 s. (R3)	20.86	21.02	25.05	36.87	39.91	45.54	8.00	11.50
	rand. 0.01, 6 s. (R2)	20.12	22.37	29.86	41.64	47.14	60.30	9.50	16.00
	rand. 0.01, 6 s. (R3)	18.71	20.17	22.39	29.52	36.85	50.46	6.00	12.00
20	- (R1)	25.15	26.73	28.69	31.91	35.04	38.27	8.00	18.00
	40th, 6 s.	26.87	27.63	29.21	35.07	44.36	56.88	9.50	16.00
	- (R2)	23.38	24.49	28.30	31.50	34.40	38.45	8.00	15.00
	40th, 6 s. (R2)	27.18	28.38	30.59	38.37	47.67	55.34	9.00	16.50
	- (R3)	21.45	27.51	29.35	31.77	34.39	38.51	7.50	10.50
	40th, 6 s. (R3)	26.73	27.90	28.86	40.08	43.67	56.66	10.00	17.00
	rand. 0.01, 6 s. (R2)	26.57	29.25	35.11	36.77	41.59	54.09	9.00	15.00
	rand. 0.01, 6 s. (R3)	25.28	30.53	36.03	37.94	39.98	52.66	8.50	13.00

Table 41: Baseline TCP with RED router, 4 parallel connections, 20 replications.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
7	- (R1)	131.91	158.01	170.47	207.58	208.77	210.77	24.00	34.00
	20th, 6 s. (R1)	140.94	152.16	178.35	222.24	228.28	237.22	26.50	38.50
	- (R2)	130.64	142.77	159.46	211.80	214.13	220.28	27.00	38.50
	- (R3)	129.42	155.21	162.04	207.22	208.44	209.24	22.50	39.00
	20th, 6 s. (R3)	147.31	169.49	174.55	220.92	222.92	233.12	29.00	44.50
	rand. 0.01, 6 s. (R2)	155.69	170.68	205.64	259.22	281.20	323.15	33.00	52.00
	rand. 0.01, 6 s. (R3)	160.44	172.24	195.69	250.91	257.86	273.42	35.00	49.50
20	- (R1)	150.02	170.16	179.05	205.11	206.20	207.03	16.00	23.00
	40th, 6 s. (R1)	149.53	167.52	184.27	211.58	212.36	213.01	15.50	25.00
	- (R2)	146.95	157.78	168.01	204.81	205.39	206.48	17.50	28.50
	40th, 6 s. (R2)	140.18	149.94	172.45	210.80	212.56	213.95	16.50	25.50
	- (R3)	154.32	175.79	187.45	204.38	205.13	205.98	14.00	25.50
	40th, 6 s. (R3)	160.45	168.15	171.70	211.12	211.70	212.40	15.50	25.50
	rand. 0.01, 6 s. (R2)	168.66	200.46	210.46	251.17	263.32	273.12	25.50	41.00
	rand. 0.01, 6 s. (R3)	199.68	210.96	219.94	248.21	255.69	271.85	23.50	41.00

Table 42: Baseline TCP with RED router. 4 parallel connections, analysis of the first 10 KB.

		elapsed time (fast)			**elapsed time (slow)**				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
7	- (R1)	23.29	26.63	28.76	67.28	92.03	109.55	4.50	11.00
	20th, 6 s. (R1)	28.32	30.26	32.63	76.65	104.55	179.59	5.00	11.50
	- (R2)	22.92	26.71	32.37	71.11	93.07	133.95	5.00	13.50
	- (R3)	20.07	24.28	28.14	74.01	90.08	113.72	4.00	12.00
	20th, 6 s. (R3)	30.20	32.13	34.49	80.42	98.98	126.88	5.00	12.00
	rand. 0.01, 6 s. (R2)	26.68	29.82	35.87	89.64	130.53	194.01	6.00	14.00
	rand. 0.01, 6 s. (R3)	28.54	34.87	38.11	85.14	114.31	138.41	6.00	16.50
20	- (R1)	26.90	29.94	31.63	56.73	76.31	85.77	3.50	11.00
	40th, 6 s. (R1)	35.03	36.66	38.81	71.60	95.88	122.18	5.00	12.00
	- (R2)	30.27	33.86	36.69	73.30	89.01	99.86	4.50	13.50
	40th, 6 s. (R2)	32.26	34.15	36.33	75.08	99.93	119.92	5.00	13.00
	- (R3)	32.81	34.84	36.94	56.88	71.32	87.02	4.00	11.00
	40th, 6 s. (R3)	33.78	36.09	39.12	64.30	70.24	91.68	4.00	11.50
	rand. 0.01, 6 s. (R2)	29.88	34.68	39.47	83.22	104.84	138.90	5.00	12.00
	rand. 0.01, 6 s. (R3)	35.26	37.98	41.48	65.88	72.37	114.19	5.50	12.50

Table 43: Initial congestion window of 4 * MSS. Single connections, 20 replications.

		elapsed time					
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**tput**	**rexmt**	**drops**
3	-	104.96	104.97	104.98	976.00	19.00	19.00
	12th, 6 s.	116.23	116.54	116.65	879.00	26.00	21.00
	rand. 0.01, 6 s.	120.76	131.22	138.27	780.50	39.50	25.50
7	-	102.91	102.92	102.93	995.00	17.00	17.00
	20th, 6 s.	109.80	109.80	109.81	933.00	26.00	16.00
	rand. 0.01, 6 s.	124.18	132.69	141.97	771.50	49.00	24.50
20	-	106.20	106.21	106.22	964.00	27.00	27.00
	40th, 6 s.	111.57	131.69	132.64	777.50	42.00	25.00
	rand. 0.01, 6 s.	121.45	136.97	143.13	747.50	49.00	27.00

Table 44: Initial congestion window of 4 * MSS. 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	55.57	55.58	55.65	124.96	125.14	125.18	9.00	13.00
	12th, 6 s.	63.20	83.42	86.59	110.02	110.76	112.51	16.50	22.00
	rand. 0.01, 6 s.	64.07	80.54	104.35	119.89	137.70	147.41	20.00	34.50
7	-	80.19	80.19	80.20	103.78	103.79	103.81	8.00	17.00
	20th, 6 s.	63.14	63.20	63.71	109.93	110.26	110.48	10.00	19.00
	rand. 0.01, 6 s.	100.39	111.50	123.54	119.12	130.54	143.55	22.00	31.50
20	-	90.96	90.97	90.99	102.70	102.70	102.73	12.00	17.00
	40th, 6 s.	67.44	70.38	78.09	108.78	109.05	109.12	15.00	17.00
	rand. 0.01, 6 s.	104.28	119.85	131.62	122.07	131.56	143.95	20.50	34.00

Table 45: Initial congestion window of 4 * MSS, First 10 KB of 2 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	13.18	13.19	13.20	82.75	82.93	82.95	5.00	9.00
	12th, 6 s.	18.75	18.75	18.77	53.27	59.34	68.18	8.00	9.00
	rand 0.01, 6 s.	12.18	12.68	19.17	50.99	65.69	84.50	6.00	9.00
7	-	22.43	22.44	22.44	28.99	29.00	29.00	4.00	10.00
	20th, 6 s.	21.76	21.77	21.79	69.56	69.89	70.11	7.00	16.00
	rand. 0.01, 6 s.	18.41	23.69	27.07	27.31	38.83	51.29	5.00	16.00
20	-	27.17	27.17	27.18	44.80	44.81	44.82	6.00	11.00
	40th, 6 s.	29.29	29.84	30.66	59.13	60.66	68.72	6.00	17.00
	rand. 0.01, 6 s.	26.93	30.13	34.44	38.35	40.84	44.64	6.00	10.00

Table 46: Initial window of 4 * MSS. 4 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	123.91	138.42	152.88	206.07	211.48	318.40	22.00	35.50
	12th, 6 s.	145.30	151.70	167.52	218.86	221.29	223.63	34.00	47.00
	rand. 0.01, 6 s.	161.21	177.76	192.86	253.41	273.13	281.55	33.00	52.50
7	-	148.52	148.57	153.18	205.65	205.81	205.85	15.00	30.50
	20th, 6 s.	127.38	153.90	180.75	218.03	219.04	220.52	21.50	35.00
	rand. 0.01, 6 s.	154.69	165.52	180.04	248.05	261.94	268.37	26.00	43.50
20	-	167.47	167.65	173.13	203.92	204.16	204.25	9.00	26.00
	40th, 6 s.	180.84	201.87	201.90	211.10	211.11	211.30	12.00	20.00

Table 47: Initial congestion window of 4 * MSS, First 10 KB of 4 parallel connections, 20 replications.

		elapsed time (fast)			elapsed time (slow)				
Buffer	**Delay**	**25 %**	**50 %**	**75 %**	**25 %**	**50 %**	**75 %**	**RxMin**	**RxMax**
3	-	23.34	23.36	23.72	100.36	143.65	276.15	6.00	15.00
	12th, 6 s.	21.62	24.15	25.53	100.42	127.15	141.27	9.00	18.00
	rand. 0.01, 6 s.	25.45	28.44	35.61	90.24	125.83	148.89	8.50	19.50
7	-	29.53	29.57	31.40	55.39	55.41	75.37	4.00	12.00
	20th, 6 s.	27.33	28.14	32.18	81.27	89.35	126.91	7.50	14.50
	rand. 0.01, 6 s.	24.76	28.83	35.80	91.15	122.10	158.78	7.00	16.00
20	-	42.11	43.29	43.33	50.31	59.57	59.58	4.00	9.00
	40th, 6 s.	39.26	39.28	39.31	63.21	63.23	74.09	7.00	14.00

B Transmission Control Protocol

We assume that the reader has a basic knowledge about Transmission Control Protocol (TCP). The basics can be found, for example, in [Ste95] and [Com95]. However, we go through the fundamentals.

B.1 General Overview

TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. The intermediate routers may discard packets, the packets can arrive to the destination out of order or the packets may be duplicated by the network. Those are the situations that TCP was made to recover from. Thus, it provides reliable, connection-oriented transportation. To provide this service, TCP has to implement facilities in the following areas: set up the connection, multiplexing, basic data transfer, reliability and flow control, precedence, and security [Pos81]. In addition to these, the protocol makes use of congestion control algorithms to monitor the congestion in the network and reduce the load in the intermediate links, if needed. These algorithms are crucial for the stability of the Internet. The congestion control algorithms are discussed in Appendix B.2.

Connections

For reliability and flow control TCP needs state information for each data stream. This includes information about window sizes, sequence numbers etc. In the beginning of the communication, the TCPs establish a connection and initialize the status information by executing a three-way handshake. Figure 23 shows the segments sent during the handshake. During the handshake, the initial sequence numbers, and possibly maximum segment sizes (MSS) and TCP options are negotiated. In Figure 23 the first transmitted packet is the `SYN` segment that tells the receiver that the sender wishes to establish a connection. In addition, the initial sequence number is provided (X in the figure). The receiver sends a `SYN` segment that contains the receiver's initial sequence number (Y), and the acknowledgment for the sender's initial sequence number (X+1, i.e. the next expected sequence number). Finally, the TCP initiator acknowledges the receiver's initial sequence number (Y+1). After this procedure the connection is established and data transfer may begin. The MSS and TCP options are "piggybacked" with the SYN messages, if used.

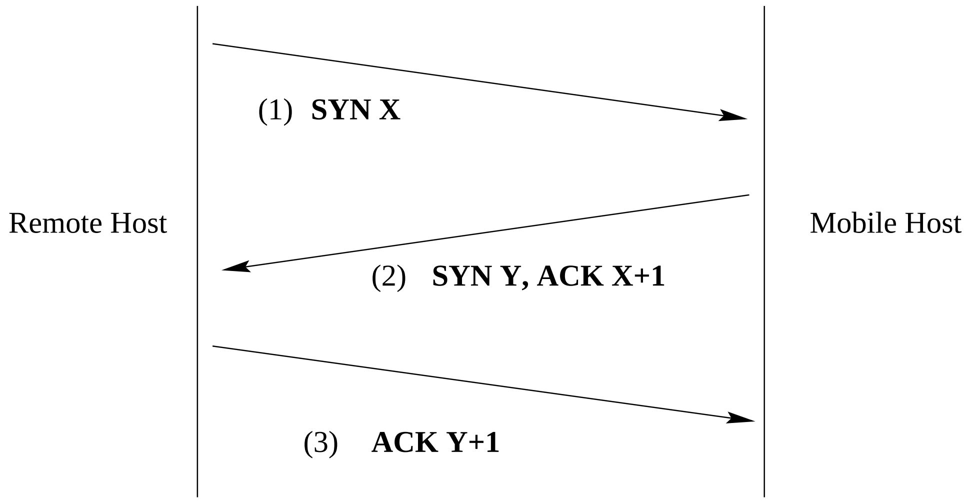

Figure 23: The three-way handshake at the beginning of the connection

Multiplexing

To allow many processes to use TCP simultaneously, TCP provides multiple ports to be used within a single host. Concatenated with the IP-address this forms a socket. A pair of sockets identifies a connection, so a single socket can be used in different connections at the same time.

Basic Data Transfer

TCP is able to transfer a continuous stream of octets in each direction between its users. It packages some number of octets into segments and forwards them to the lower layer for transportation through the Internet.

Reliable transportation

The TCP recovers from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (`ACK`) from the receiving TCP. The initial sequence numbers are negotiated between the hosts at the beginning of the connection (see Figure 23). In the `ACK` segment the next expected octet is indicated using the sequence number. The acknowledgments are cumulative, so each `ACK` acknowledges all previous segments, too.

TCP makes use of timers to monitor the flow of incoming `ACK`s. If the acknowledgment has not arrived within a certain time interval, a *retransmission timeout* (RTO) occurs and the segment is retransmitted. The TCP sender constantly monitors the *round-trip time* (RTT) and calculates the RTO value using the following equations[Jac88]:

$$\begin{aligned} Diff &= New\_RTT - SRTT \\ SRTT &= SRTT + \delta * Diff \\ DEV &= DEV + \rho * (|Diff| - DEV) \\ RTO &= SRTT + \eta * DEV \end{aligned}$$

In the above equations New_RTT is the round-trip time measured from the latest packet, $SRTT$ stands for "Smoothed round-trip time" and DEV is estimated mean deviation. δ and ρ are constants between 0 and 1 to be chosen by the implementation. [PA00] states that δ should be set to 1/8, ρ set to 1/4 and η should be set to 4. These values are also widely used in the present TCP implementations. The equations presented above were introduced in [Jac88], as the original algorithm introduced in [Pos81] turned out to be inadequate with congested internetworks. Later RFC [Bra89] states that a TCP implementation must follow the rules mentioned above. The retransmission timer must be reseted when an `ACK` is received that acknowledges new data. This way the timer will expire one round trip time later, after RTO seconds. For a more detailed description, refer to [PA00].

At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Data corruption is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

In-order delivery is achieved by buffering the received segments at the receiving end until the expected octet of data are obtained to be delivered to the receiving application. The TCP receiver does not acknowledge the incoming segments if some previous octets have not yet arrived. In that case, receiving TCP sends a *duplicate acknowledgment* every times it gets a new segment until the missing segment is received. The *duplicate acknowledgment* is an `ACK` that acknowledges the same segment as the previous `ACK` (i.e. the sequence number is the same). The behavior of the TCP sender, as well as the whole algorithm called fast retransmit/fast recovery, is described in more detail in Appendix B.2.4.

So-called *delayed acknowledgment* is used to reduce traffic over the network. An `ACK` is not sent immediately after the arrival of a data segment. This makes possible to piggy-back data or window updates. Every second full-sized segment should be acknowledged and the `ACK` can not be delayed for more than 500 ms [APS99]. A usual delay threshold for the delayed acknowledgments is 200ms which is more user friendly in interactive connections where the response time is more important. *Delayed acknowledgments* slow down the slow-start phase because the number of incoming `ACK`s is decreased, so the congestion window does not grow as quickly. To improve the feedback from the receiver, it is stated that duplicate acknowledgments should be sent immediately to the TCP sender[APS99]. Also, the receiver should send an immediate `ACK` when it receives a data segment that fills in all or part of a gap in the sequence space[APS99].

Flow control

TCP provides a means for the TCP receiver to control the amount of data sent by the sender. This prevents the sender to overfill the receiver's storage space (i.e. socket buffer) in case the receiver is not capable to consume the data as quickly as the sender provides new segments. This is achieved by returning a "window" with every `ACK` indicating a range of acceptable sequence numbers beyond the last segment successfully received. The *receiver advertised window* (`rwnd`) indicates an allowed number of octets that the sender may transmit before receiving further permission. In many implementations the socket receive buffer size determines how large a window the receiver advertises. The socket buffer size has system dependent default value but can usually be set directly by the application. In addition to the advertised window, several congestion control algorithms, such as slow-start and congestion avoidance, control the number of segments sent to the receiver.

Precedence and security

TCP makes use of the IP type of service field and security options to provide precedence and security to TCP users.

B.2 Congestion control

The basic congestion control mechanisms are described in this section. These algorithms were not included in the earliest TCP specifications.

B.2.1 Background

As noted before, IP networks may drop packets, for example, when the router buffers become full and no storage space is available for the incoming packet. It causes a TCP retransmission as the TCP senders retransmission timer expires because the acknowledgment for the packet has not arrived. The sender will send the packet again - and cause more congestion in the routers. This was the situation earlier as the congestion control algorithms were not yet in use. The receiver's advertised window was the only way to reduce the data flow.

The first congestion collapses started to occur in October 1986 when the data throughput decreased in factor-of-thousand from 32 Kbps to 40 bps[Jac88]. At that time, research was carried out to understand the background of the collapses. As a result of the study, algorithms called *slow-start* and *congestion avoidance* were introduced[Jac88]. This statement outlined the principles when designing the algorithms:

"If packet loss is (almost) always due to congestion and if a timeout is (almost) always due to a lost packet, we have a good candidate for the 'network is congested' signal."[Jac88]

B.2.2 Slow-Start and Congestion Avoidance

Slow-start and congestion avoidance were first introduced by Jacobson [Jac88] and they were quickly made mandatory [Bra89]. It allows the TCP sender to probe the capacity of the network by increasing the sending frequency to probe the network capacity. The slow start algorithm is used for this purpose at the beginning of a transfer, after repairing loss detected by the retransmission timer, and after idle periods.

To implement these two algorithms we need to add two state variables: congestion window (`cwnd`) and slow-start threshold (`ssthresh`).

cwnd is a sender-side limit on the amount of data the sender can transmit into the network before receiving an `ACK`, while the receiver's advertised window (`rwnd`) is a receiver-side limit on the amount of outstanding data. The minimum of `cwnd` and `rwnd` governs data transmission [APS99].

ssthresh is needed to determine whether to use slow-start or congestion avoidance. If `cwnd` is smaller than `ssthresh`, then slow start is used. Otherwise the TCP sender uses congestion avoidance.

The initial value of `cwnd`, called *initial window* (IW), must be no more than two segments[19]. After the connection establishment, the TCP sender transmits as many segments to the network as the value of IW permits. Before the sender receives an `ACK` the sender is blocked and cannot send any new data. After receiving the `ACK`, the number of new segments to be sent is equal to the number of "acked" segments plus one segment due to the increase of the `cwnd`.

During slow-start, a TCP sender starts with a `cwnd` of one or two segments and increases the `cwnd` by at most sender maximum segment size (SMSS) bytes for each `ACK` received that acknowledges new data. Slow-start ends when cwnd reaches ssthresh or when congestion is observed. The sender may observe the congestion in two ways: the retransmission timer expires or after receiving three consecutive *duplicate acknowledgments* (dupack). The data receiver sends a *dupack* after receiving an out-of-order segment. When the third *dupack* has arrived, the TCP sender goes to a fast retransmit/fast recovery algorithm.

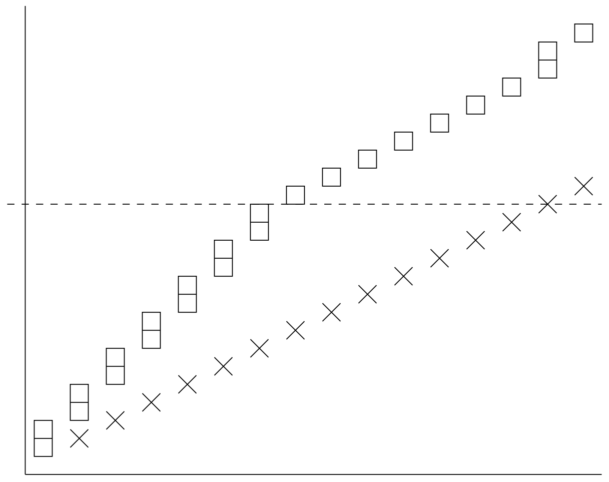

Figure 24: A trace of slow start with the IW size of two segments, without delayed acknowledgments.

In Figure 25 we see the slow-start phase ending when it reaches the receiver advertised window after 68 seconds (i.e. the point in the figure where the line *sent seqhigh* reaches the line *receiver window*).

During congestion avoidance, `cwnd` is incremented by one full-sized segment per round-trip time (RTT). That means that the `cwnd` is increased only after a full window of data is acknowledged. Congestion avoidance continues till the end of the connection or until congestion is detected by RTO. A commonly used formula to update `cwnd` is [APS99]:

$$cwnd+ = SMSS * SMSS/cwnd \tag{2}$$

[19]In experimental TCP extensions, values three and four are accepted, as well [APS99]

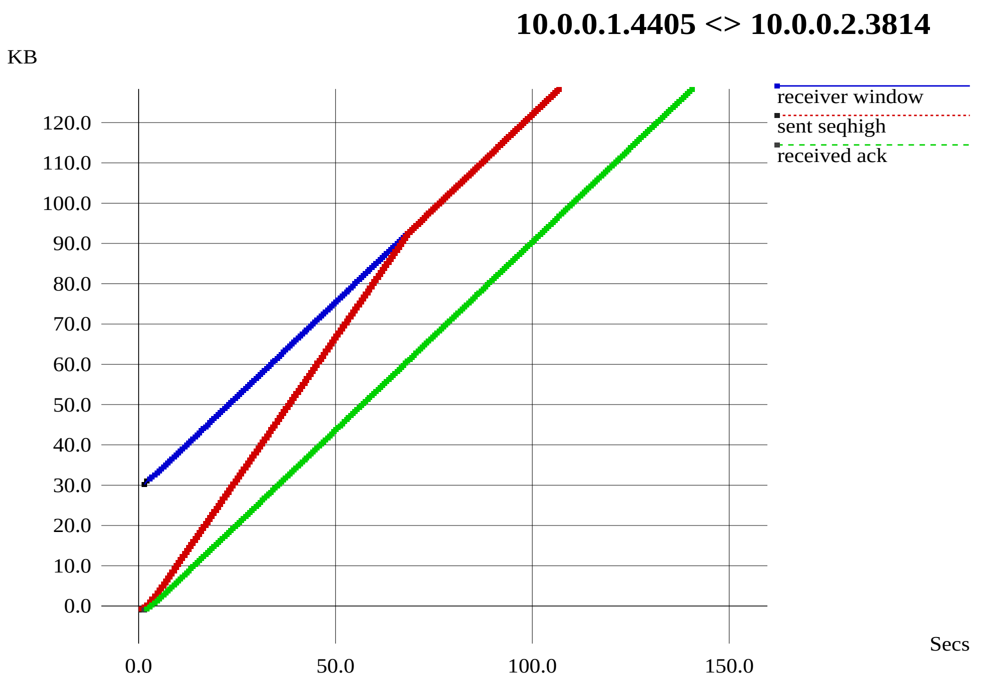

Figure 25: A 130KB transfer

Figure 24 shows this situation as the TCP sender sends two new segments the the networks after receiving an `ACK`. If *delayed acknowledgments* (explained in Appendix **??**) were in use, the sender would send three segments instead of two because the incoming `ACK` acknowledges two segments. After the `cwnd` has reached `ssthresh` the slow start is exited and congestion avoidance is invoked. After successfully sending a whole window of data (eight data segments), `cwnd` is increased by one segment.

B.2.3 Recovery from Retransmission Timeout

As noted before, TCP makes use of timers to monitor the flow of incoming `ACK`s. The equations used to achieve the RTO were described in Appendix B.1.

When a RTO occurs, the TCP sender interprets it to have two meanings. First, the segment that was not acknowledged in time is missing and, second, the loss is due to congestion in the network. Thus, the TCP sender will retransmit the segment and invoke *slow start* with a `cwnd` of one segment [APS99]. In addition, `ssthresh` is updated to be half the current number of segments outstanding in the network (*flightsize*) and the *exponential backoff* [KP87] algorithm is used to re-initialize the retransmission timer.

Because `ssthresh` is lowered, the *congestion avoidance* phase will start sooner. This way the capacity of the network is not exceeded as quickly as with a longer *slow start*. The correct formula to calculate the new value of `ssthresh` is [APS99]:

$$ssthresh = max(flightsize/2, 2 * SMSS) \tag{3}$$

Figure 26 shows an artificial trace of the congestion window before and after a retransmission timeout. The connection starts with slow start and proceeds using congestion avoidance until the retransmission timer expires. As a result, `cwnd` is set to one segment and slow start is invoked again. The new value of `ssthresh` is visible as the congestion avoidance starts in an earlier phase than before the RTO.

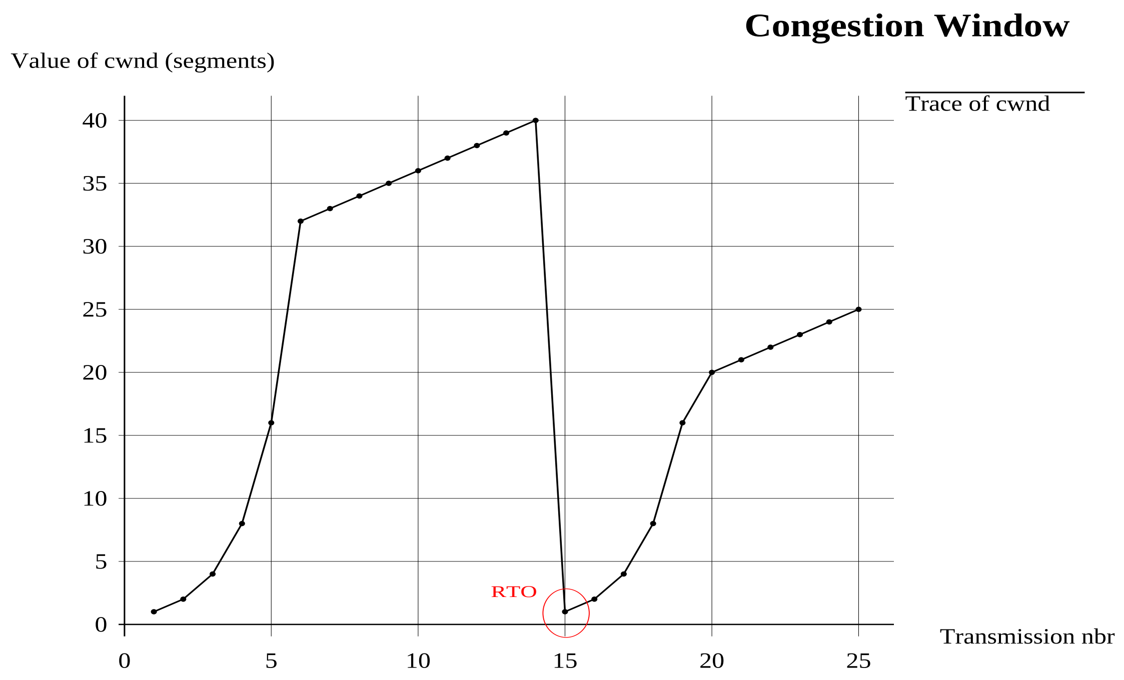

Figure 26: The trace of `cwnd` after a retransmission timeout

Each time the RTO has occurred, the new RTO value has to be the old RTO multiplied by a constant value, which must be at least 2[PA00]. This, so-called *exponential backoff* algorithm, is used after RTO has expired to avoid future RTOs to be triggered unnecessarily. Also, the retransmissions may cause more congestion to the network, so the threshold when to retransmit data should be higher every time the TCP sender has detected congestion.

The SRTT estimate should be updated only after new segments have been acknowledged, the acknowledgments for the retransmitted TCP segments should not be taken into account. This is to avoid the problem caused by an old acknowledgment arriving for the first packet just after a retransmission has been triggered by a RTO, causing the RTT measurement to be invalid.

B.2.4 Fast Retransmit/Fast Recovery

This algorithm provides a quicker way to recover from a single packet loss. The latest TCP congestion control specification [APS99] states that these algorithms *should* be implemented. Before this algorithm was in use, the retransmission timeout (RTO) was the only way to observe a packet loss.

The TCP receiver is required to send a *duplicate acknowledgment* (dupack) immediately when an out-of-order segment arrives. The *dupack* is identical to the previously sent `ACK`, i.e. the acknowledged sequence number is the same. The purpose of a *dupack* is to inform the sender that a segment has received out-of-order and which sequence number is expected. The *dupacks* can be caused by different reasons, not just a packet loss. The network may have re-ordered the data segments, or the `ACK`s (or data segments) may have been replicated by the network [Pax97b]. When the TCP sender receives the first *dupack*, it cannot yet retransmit the segment because the dupacks can be caused by a number of network problems, not just a dropped segment. There is other information in the ACK, too. The receiver can only generate one in response to a segment arrival. A *dupack* means that a segment has left the network (it is now cached at the receiver)[20]. Thus, if the sender was limited by the congestion window, a segment can now be sent. This is the reason why the sender does not have to go to into slow-start. The fast retransmit algorithm uses three *dupacks* (four identical `ACK`s without any other intervening segments) as an indication of a lost segment.

Taking these different aspects into account, the fast retransmit and fast recovery algorithms are usually implemented as follows [APS99]:

1. When the third *dupack* is received (4 identical `ACK`s without any other intervening segments) , set `ssthresh` to no more than the value given in equation 3.
2. Retransmit the lost segment. Set $cwnd = ssthresh + 3 * SMSS$. This "inflates" the congestion window by the number of segments (three) that have left the network and the receiver has buffered.
3. Increase `cwnd` by one SMSS for each additional *dupack* received.
4. Transmit a segment if allowed by the new `cwnd` or `rwnd`.
5. When the next `ACK` arrives that acknowledges new data, set `cwnd` to `ssthresh` (the value set in step 1). This is termed "deflating" the window.

[20]A large duplication of segments by the network can invalidate this conclusion.

The ACK received in step 5 should be due to the retransmission of the segment in step 1. If so, the retransmitted segment was the only one to be missing. It is well known that fast retransmit and fast recovery algorithms do not work well if multiple segments are dropped within a single window[FH99][MMFR96]. This is because the algorithm retransmits only the first segment without waiting for the RTO after the third *dupack*. While in fast recovery phase, all other packet losses are observed using the retransmission timer. A single retransmit timeout might result in the retransmission of several data packets, but each invocation of the Reno fast retransmit algorithm leads to the retransmission of only a single data packet[FH99]. A more detailed description is presented in [Hoe96], for example.

Figure 27 shows the recovery from a lost segment. The x-axis is the time and y-axis is the highest sequence number of a segment. Both, sent segments and received `ACK`s, are printed. The figure is drawn from the sender's TCP dump[21] and the dropped segment was the 15th segment. The received *dupacks* are clearly visible as a horizontal line after 5 seconds of transfer and the fourth *dupack* triggers the single retransmission. After retransmitting the expected segment the TCP sender has halved the `cwnd`. After 7.5 seconds the TCP sender may transmit new segments even if the retransmitted one is not yet acknowledged. This is due to the constant flow of *dupacks* that increase the `cwnd` to a value that allows new segments to be sent to the network. Once the new `ACK` arrives at the TCP sender, the connection continues by using *congestion avoidance*.

B.2.5 NewReno TCP modification

The basic problem of fast retransmit is that it invokes only one fast retransmission without waiting for the retransmission timer to expire. Therefore, the recovery from situations where multiple segments are dropped from a single window, is not good. There is not much information available for the TCP sender for making retransmission decisions during fast recovery. However, a response to so-called *partial acknowledgments* (described later) ameliorate this situation.

Many different modifications for the regular fast recovery algorithms exist. The NewReno modification is specified in [FH99] and it was first introduced in [Hoe95] by Janey Hoe. This modification is based on the information achieved when the first acknowledgment of new data after three consecutive duplicate acknowledgments is received. If there were multiple packets dropped, the acknowledgment for the retransmitted segment will acknowledge some but not all of the segments transmitted before the segment retransmitted by the fast retransmit. This packet is a *partial acknowledgment*.

[21] A software that is used to monitor the network traffic going through the computer

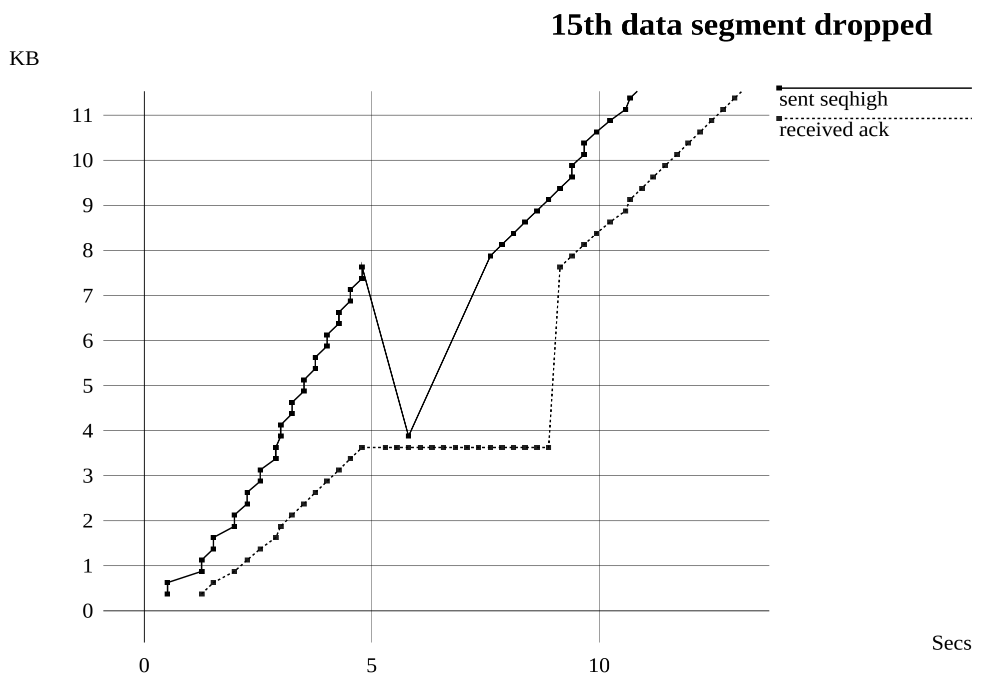

Figure 27: Recovery from a missing segment using the fast retransmit/fast recovery algorithm

NewReno modification makes use of a new variable called *recover* that is set to the highest sequence number sent before receiving three duplicate ACKs. The steps of NewReno modification are mostly the same as in regular fast retransmit and fast recovery described in Appendix B.2.4, page 104. The only changes are in step 1 where the highest sequence number sent is recorded in the variable *recover* and in step 5 where responses to partial acknowledgments are produced. The response to the partial ACKs (step 5) is explained below. If the incoming ACK acknowledges all segments up to and including the sequence number in *recover*, *cwnd* is set to *ssthresh* as in regular fast recovery. The fast recovery procedure may be exited. A second option is to set the *cwnd* to *flightsize* + MSS.

Recovery from partial acknowledgments

During the recovery period, when an ACK arrives that acknowledges new data, it could be the ACK elicited by the retransmission of step 2 (due to dupacks) or later retransmissions. Different actions are made if the ACK acknowledges all the segments sent up to the variable *recover* or if it is just a partial ACK. If the acknowledgment is a partial ACK, retransmit the first unacknowledged segment. Deflate the `cwnd` by the amount of new acknowledged data and add back one segment. This is so called "partial window deflating". Reset also the retransmission timer, but only for the first partial ACK that arrives during fast recovery. The fast recovery procedure does not end, so later duplicate ACKs received invokes steps 3 and 4.

C 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].

C.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".

C.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.

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[22].

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".

C.1.2 Recovery from RTO

Linux kernel was modified to implement "BSD style" RTO recovery[23]. The exact behavior is explained in SectionB.2.3

C.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.

C.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

[22]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.

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

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" [24] in the beginning of the connection before the *delayed acknowledgments* are taken into use.

C.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.

C.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 48 summarizes the algorithms, parameters and their values used in Baseline TCP.

C.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.

C.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.

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

Table 48: 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 C.1.1
`cwnd` after exiting NewReno	*flightsize + SMSS* (Appendix C.1.1)
NewReno "Bugfix"	*Less Careful* variant (Appendix C.1.1)
Recovery from RTO	'BSD' style (Appendix B.2.3)
Delayed `ACK` threshold	200ms (Appendix C.1.4)
Quickacks	Two segments in the beginning of the connection
Advertised window (`rwnd`)	32Kbytes (Appendix C.1.5)
Control Block Interdependence	Disabled by default (Appendix C.1.6)
SACK	SACK option is disabled
Timestamps	timestamps are disabled

- **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`.

C.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 non-conformant behavior for example when several packets are dropped in the middle of the current FlightSize.

D TCP Enhancements

In this section we discuss enhancements 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. Finally, the active queue management in the router buffer is described.

D.1 TCP Control Parameters

D.1.1 Increasing initial congestion 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.

D.1.2 Receiver's advertised 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 should limit the advertised window [**?**]. 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.

D.1.3 Maximum segment size

The Maximum Segment Size affects TCP performance [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.

We expect that TCP throughput increases with a larger MSS in our environment. The router will drop less packets, because the router buffer limit is in terms of packets, not bytes. Our error model also favors larger packets, because the error loss probability is independent of packet size. Due to these factors we cannot directly compare the results of tests with increased MSS with other optimizations.

D.2 TCP Optimizations

D.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].

D.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 D.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

```
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 28: The general algorithm of the Random Early Detection (RED).

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.

E Test arrangements

In this section we describe the characteristics of the environment where we run the performance tests. The characteristics are somewhat similar to data transfer over a GSM link, but the goal of the performance tests is not to exactly model the behaviour of GSM or any other wireless communication technology. The environment is emulated using Seawind wireless network emulator.

E.1 Seawind emulator

In this section we describe *Seawind* [AGKM98], a real-time software network emulator developed at the University of Helsinki. Networks with various properties can be emulated with Seawind by altering different simulation parameters affecting, for example, the bandwidth, latency and reliabilty of the emulated network. Seawind can be programmed to run several test configurations automatically, making it possible to run tests for several hours without human interruption.

E.1.1 General overview

Seawind is a real-time emulator which can be run on any common Unix system. It is transparent to the network applications which are used to generate traffic into the emulated network, and therefore any application can be used in performance tests to test different workload patterns. Seawind can be distributed on multiple hosts, which is an important property, because our goal is to achieve accurate results on top of non-realtime operating systems, and thus avoid competing processes to exist in the hosts which are used in the emulation.

Seawind consists of a number of components with a specified role in the emulation. The Seawind architecture is illustrated in figure 29. For the user, it provides a *Graphical User Interface (GUI)*, which can be used to define the test parameters and control the test runs. GUI is closely tied with *Control Tool (CT)* which controls the execution of automated test runs, passing the appropriate information to the different components and collecting the log information generated by the components. Two types of log information is collected: *Seawind log*, which shows a detailed description (e.g. queue size, delays and other events for each packet) of the actions taken by Seawind in the emulation, and *filter log* which shows the relevant protocol information about the packets injected to Seawind. For example, when TCP traffic is transmitted through Seawind, the filter log output is similar to the output generated by the widely used `tcpdump` [JLM97] tool. The filter log is generated by both connection endpoints and all SPs used for the simulation.

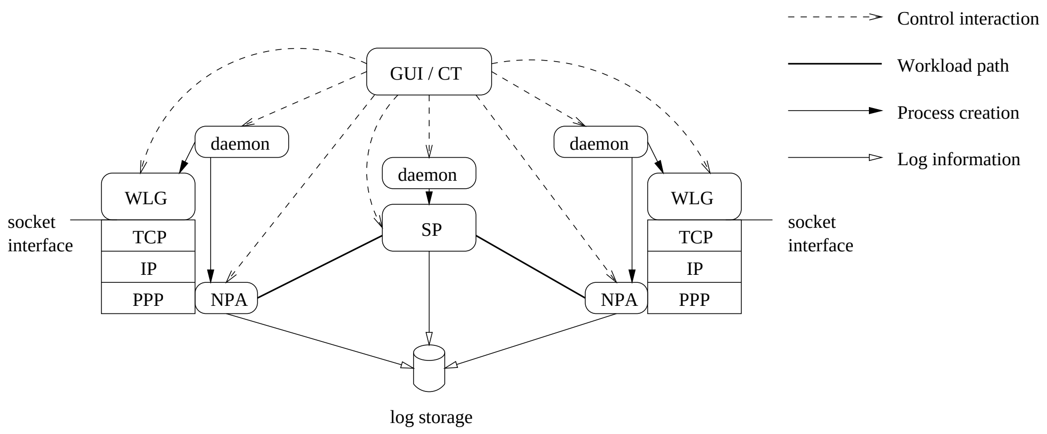

Figure 29: Architecture of the Seawind emulator.

Simulation of the target network is done by *Simulation Process (SP)*, which can be used to simulate a link with given rate and reliability properties, as well as a limited input buffer separately at both ends of the link. More complicated networks can be emulated by attaching several SPs as a pipeline of components, of which each can be used to model, for example, a node or a subnetwork in the connection path. *Network Protocol Adapters (NPA)* are located at both connection endpoint hosts and they take care of routing the data generated by applications through the pipeline of SPs to the NPA at the other end of the emulated network path. The NPAs catch the data after it has been handled by the network code of the operating system, thus making the transfer transparent to the applications. In effect, the applications behave as if they really were used over the emulated link. Additionally, NPAs take care of inserting the data into link layer frames using a specified link layer protocol, such as PPP [Sim94]. *Workload Generator Controller (WLGC)* controls the applications at the endpoint hosts, executing the applications automatically and collecting the output generated by the application. WLGCs are needed to make it possible to run automated tests. Additionally, we denominate the applications (or tools) which are used to generate the workload with a generic term, *Workload Generator (WLG)*. *Seawind daemon* is required in the hosts that are to be used in emulation to manage the TCP connections used to pass the control information and to create and tear down the components used during the emulation.

Various configuration files can be defined using the GUI to define the properties of the different components described above. In addition to having configuration files for each SP used in emulation, WLGs and NPAs at both ends have their own configuration file. The parameters used for these components depend on the tool used for workload generation and the link layer protocol used at the NPA. Additionally, there is a network configuration file, which defines how the components are distributed in the hosts used in the emulation. The combination of these configuration files is called a *configuration set.*

Because Simulation Process is the core of the emulation, we describe its functionality in more detail. The internal logic of SP can be also thought as a pipeline, because different operations are done for each packet in a defined order. Figure 30 shows the emulation events triggered for each packet arriving in SP and finally transmitted out to the next SP or to the NPA. All of the events shown in the figure are optional and can be skipped from the emulation. Both directions of the traffic flow are processed through the similar set of events independently.

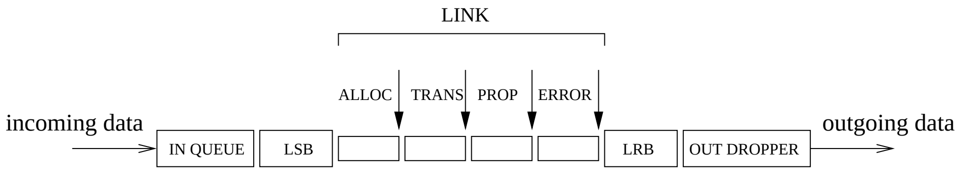

Figure 30: Ordering of events for a packet in a simulation process.

When a packet arrives to SP, it is appended to the end of the *input queue.* If the size of the input queue is limited and the input queue is full, the packet can either be dropped (thus implementing a *drop tail* policy), or the receiving of new data may be suspended until there is more room in the input queue again. If there is room for packets in the *link send buffer (LSB)*, the packets are taken from the head of the input queue and appended to the link send buffer. In effect, the input queue will not be filled up before the link send buffer is full.

The packets are taken from the link send buffer to the virtual link one at a time. When a packet is on a link, it can be affected by various delays before it is transmitted out from the SP, or dropped to emulate a transmission error on the link. First, an *allocation delay* can be issued for the packet, which occurs when the packet arrives at an empty link. After the allocation delay is finished, the packet is put under *transmission delay*, which is used to model a defined bandwidth of the link. The length of the transmission delay depends on the size of the packet and the sending rate the user has chosen. After the transmission delay is finished, the packet is put under a *propagation delay.* The length of the propagation delay is the same for every packet, regardless of the size of the packet. At the same time, the next packet can be taken from the link send buffer under a transmission delay (an allocation delay is not needed, because the link is not idle).

After the propagation delay, an *error delay* can be issued on the packet. This is commonly used to model link layer retransmissions on a reliable link layer protocol. Error delay occurs on a packet randomly, which makes it possible for packets to get reordered at this point of emulation. However, the packets are ordered again at the *link receive buffer (LRB)* and therefore, packets following the error-delayed packet will also be affected by the delay to some extent. Alternatively to the error delay, the packet may also be dropped or corrupted by a certain probability at this point.

An alternative location to drop a packet is just before it would be sent out (output dropper in figure 30). Various distributions can be chosen for the probability for dropping a packet at the output dropper.

Seawind can also be used to predefine various configuration sets that can be tested automatically. The group of configuration sets that are used in a test run are called *test set*. This is an important feature, because each test run usually takes minutes to complete, as the test environments are emulated in a real-time basis. Each configuration set can be tested repeatedly for a defined number of replications (we call one repetition a *basic test*). After all replications have been run, the next configuration set in the test set is chosen and a number of test runs are run with it. Logs of each configuration set are collected into a separate file to be analysed later.

E.1.2 Emulation parameters

In this subsection we describe the most important parameters of Seawind which are used to emulate various environments in the IWTCP performance tests. SP parameters are chosen separately for uplink and downlink traffic. NPA parameters are common in both flow directions, but are chosen separately for mobile and remote ends.

Simulation Process

The following three parameters define the properties of the *input queue*, in which the packets arrive first, when SP receives them. The input queue can be used to emulate, for example, the last-hop router buffer.

There are three types of parameters. Some of the parameters contain a set of literal values, of which one is chosen. Other parameters have a single numeric value. A third type of parameters are *distribution* parameters, for which a random distribution with parameters specific to that distribution is chosen. The actual parameter values are chosen randomly based on the selected distribution. The following distributions may be chosen: *static*, *uniform*, *normal*, *lognormal*, *exponential*, *hyperexponential*, *2-phase markov*, *beta*, *gamma*, *cauchy* and *user*. User distribution is an external file containing numeric values, which the random function uses uses sequentially. User distribution can be used to repeat a predefined set of events. Additionally, e.g. single packet drops and single delays can be caused using the user distribution.

- **queue_max_length** defines the maximum length of the input queue in a number of packets. If this parameter is not defined, the input queue length is unlimited.
- **queue_overflow_handling** defines what to do when new packets arrive and the input queue is full. There are three choices that can be made. When *DROP* is selected, SP drops packets when they do not fit in the input queue. The *STOP* mode causes the router to stop reading when the queue becomes full. *FLOW CONTROL* can also be selected, in which case the SP blocks the neighbouring component (another SP or NPA) from sending new data until there is more room in the SP's input buffer.
- **queue_drop_policy** defines the algorithm to be used in deciding when to drop packets and which packets are dropped. Currently there are two policies that can be chosen. *TAIL* is the traditional tail-drop policy and *RED* is the RED drop algorithm [FJ93] and if it is chosen, some additional RED parameters need to be specified. *min threshold* specifies the threshold for the number of packets in the input queue, after which the SP starts dropping packets by a certain packet drop probability. *max threshold* specifies the number of packets allowed in a queue after which all packets are dropped from the queue. Thresholds are not compared to the actual queue size, but to a moving average of recent queue length samples. *queue weight* defines how much each queue length measurement affects the moving average compared to the threshold values. The value is defined as a fraction of the total queue length. *max probability* defines the maximum probability for a packet to be dropped when the average queue length is between the thresholds. A detailed description about the RED algorithm can be found in [FJ93].

Following two parameters define the sizes of link buffers (LSB and LRB in figure 30). Sizes are defined in bytes. Link buffers store total packets, even if there would be space in the buffer to store a fraction of the next packet.

- **link_send_buffer_size** can be thought of as an extension to the input queue, and is located next to the input queue, at the sending end of the link.

- **link_receive_buffer_size** defines the size of the link receive buffer, located at the receiving end of the link.

The next set of parameters define the bandwidth and latency of the emulated link.

- **rate_base** defines the basic rate in which the data can be transmitted to the link. It affects the transmission delay calculated for each packet.
- **available_rate** is defined as a multiplier to the *rate_base* described above. This is a distribution parameter and a new random value is chosen following the specified distribution after intervals defined with the *rate_change_interval* parameter. The resulting transmission rate at the emulated link is *rate_base * avail*, *avail* being randomly selected as defined with this parameter. If this parameter is not defined, a static transmission rate is used, as defined with *rate_base* parameter.
- **rate_change_interval** defines the distribution for time intervals in which the available rate is changed.
- **mtu** parameter for SP defines the size of the units SP reads with a single `read()` call from the incoming data socket.
- **propagation_delay** defines the length of the propagation delay which affects each packet.

The following parameters define the various delays shown in figure 30 and the two locations in which the errors can be affected (*error dropped* and *output dropper*).

- **allocation_delay** is a distribution parameter, applied as described above. The distribution of allocation delay length is given with this parameter. If this parameter is not defined, allocation delay does not occur at all.
- **error_handling** defines the type of action taken by SP when an error occurs for a packet. The errors are caused at the end of the emulated link, after the other delays have been finished. The possible types of error handling are dropping the packet when an error occurs (*DROP*), delaying the packet for a time specified with the *error_delay_function* parameter (*DELAY*) or corrupting the packet without dropping or delaying it at this point (*FORWARD*). If the error parameters are not defined, no delays, packet drops or data corruption occur at this point of the link.
- **error_rate_type** defines the unit against which the error probability is defined. The errors can be either bit errors *BIT* in which case the value given with *error_probability* parameter is the bit error rate of the link, or the error probability can be defined per packet *UNIT*.

- **error_probability** is a distribution parameter defining the probability of error, either per unit or per bit, depending on the value of the *error_rate_type* parameter.
- **error_delay_function** is used only when the *error_handling* parameter is set to *DELAY*. It defines the distribution of error delay lengths to be applied to the packets which are affected by the error delay.
- **output_dropper** is a distribution parameter which defines a probability for each packet to be dropped at the output dropper.

Network Protocol Adapter

The following parameters affect the behaviour of the PPP NPA located at both ends of the connection path.

- **mtu** and **mru** are parameters for PPP, defining the maximum size data unit the PPP will transmit and receive from the device interface. TCP MSS depends on this parameter, and is usually the *mtu* negotiated by the endpoint subtraced by the TCP/IP header length.
- **buflen** defines the maximum size of data block that is read by a single `read()` call by NPA from the SP or from the PPP daemon.

Example of a parameter set

A GSM-like link with a RLP-like protocol and a last-hop router with a buffer size of 7 packets would be emulated with Seawind by setting the values described in table 49. The table shows SP settings separately for uplink and downlink flow. Additionally, the chosen NPA values are shown. These values cause a TCP MSS of 256 bytes to be used.

Note that the given parameters are approximations of a GSM-like link and this setting makes simplifying assumptions (e.g. for the delays). The SP queue for the downlink is used to emulate the last-hop router buffer. No queue length limit is specified for the uplink, but it could be used, for example, to emulate a buffer in a wireless device interface. Link buffer sizes are somewhat close to the size used in the RLP protocol. Additional delays are created randomly using error delays. Delays occur at a per-packet probability of 0.01 and their length is uniformly distributed between 500 and 6000 milliseconds. These delays would emulate e.g. link layer retransmissions in case some data is corrupted.

Table 49: An example of chosen parameter values when emulating a GSM-like link.

Parameter name	**Downlink value**	**Uplink value**
queue_max_length	7	-
queue_overflow_handling	DROP	-
queue_drop_policy	TAIL	-
link_send_buffer_size	1220 bytes	1220 bytes
link_receive_buffer_size	1220 bytes	1220 bytes
rate_base	9600 bps	9600 bps
available_rate	-	-
rate_change_interval	-	-
mtu	512 bytes	512 bytes
propagation_delay	200 ms	200 ms
allocation_delay	-	-
error_handling	DELAY	DELAY
error_rate_type	UNIT	UNIT
error_probability	0.01	0.01
error_delay_function	uniform(500 ms, 6000 ms)	uniform(500 ms, 6000 ms)
output_dropper	-	-
NPA: mtu	296 bytes	296 bytes
NPA: mru	296 bytes	296 bytes
NPA: buflen	4096 bytes	4096 bytes

E.1.3 Discussion

The issue of emulation accuracy is worth discussion. Linux, as a non-realtime operating system can not guarantee an exact response time for user space application, such as Seawind. The critical issue affecting the simulation accuracy is the accuracy of the sleep times issued from the operating system. When a Linux process is put to sleep (e.g. during the delay of a packet) for a specified amount of time, it is usually woken up a few milliseconds late of the time issued. The exact amount of oversleeping varies, so the solution is not as simple as just subtracting a certain amount of milliseconds from the wanted sleep time.

We have implemented delays so that each sleep issued by Seawind is a parametrised amount of milliseconds shorter than the actual amount to be slept. The estimated oversleep time is chosen to be large enough to ensure that the simulation process is usually woken up before the actual wakeup time is due. We have set this estimate to 7 milliseconds. When the process is woken up, it checks from the system clock how many milliseconds it still has to wait before the accurate sleep time is finished. The process spends the rest of the delay time in a busy loop, exiting it at the time when the issued delay is finished. Additionally, the timestamps after each sleep are written into the Seawind log, so that the exact sleep times can be monitored and it is ensured that the results are accurate. Of course, there must not be any CPU intensive processes at the same host as SP during the test runs.

Another concern related to simulation accuracy is that the PPP frames are transmitted on top of the TCP protocol between the NPAs and the SP. The frames are small in our tests (PPP MTU of 296 bytes is usually used), so there is a risk that the *Nagle's algorithm* [Nag84] is activated, causing a frame to be delayed at the sender until more frames are issued to be sent. Linux allows the Nagle's algorithm to be turned off by a dedicated socket option, and we have disabled the Nagle's algorithm for the internal traffic of Seawind.

E.2 Test setup

In this section we describe the environment used in the IWTCP performance tests.

E.2.1 Emulation environment

The IWTCP performance tests were run in a isolated LAN (10Mbps Ethernet) with four network hosts. The machines in the network are 400-Mhz Intel Celerons, running Linux RedHat 6.1. Three of the machines are used in a single test run. One machine acts as a mobile end host, another as a remote end host and one machine runs the SP. The mobile and remote end machines have Linux kernel version 2.3.99-pre9, which we have modified afterwards (see Appendix C for details). The SP host has an unmodified version 2.2.14 of the kernel. The TCP behaviour of the SP host does not have an effect on the emulation results, as long as it receives and transmits the Seawind data timely (e.g. the Nagle algorithm is disabled for Seawind emulation data traffic).

E.2.2 Logging

SP generates a log of the actions made during a test. For each log event there is a timestamp of the event in microseconds, flow direction for which the event occured, event type, packet id and event description shown. Log events are generated for various reasons:

- **Arrival of packet.** Each packet received by SP cause this log event. In addition to the common information described above, the size of the packet (including PPP overhead), IP address of the source, length of the IP packet (including the header), TCP port of the source, TCP sequence number and TCP acknowledgement number are printed.

  For each packet received, the size of the input queue is printed each time a packet arrives.
- **Queue drops.** Each packet dropped from the input queue generates a log event. The reason for dropping is printed in addition to the standard output (e.g. Queue overflow, RED probability hit or RED max threshold exceeded)
- **Delays.** Each delay that affected packet are written in the log. In addition to the standard information, the delay type (allocation, transmission, propagation or error) and the delay length are printed. If the delay event was triggered significantly too late, a warning is printed.
- **Random drops.** If a packet is dropped either by the error dropper or by the output dropper, an event is created to the log.
- **Rate changes.** If the transmission rate is changed, a log event is generated. In addition to the standard information, the new transmission rate is printed.
- **Packet transmissions.** After a packet has traversed through the SP emulation process, it is sent out. A log event is generated for each packet transmitted and released by the SP. Additionally, the time elapsed from the last SP event to the return of `write()` call is printed.

In addition to SP log, the output of WLGs is written to a dedicated log file. This file contains the application level output, e.g. time measurements made by `ttcp` tool.

Filter logs can be received from SPs and NPAs (i.e. at the connection endpoints). When TCP traffic is used, filter logs are simply output of the `tcpdump` tool [JLM97]. For each TCP segment timestamp in microseconds, the sending and receiving IP address and TCP port, sequence number, acknowledgement number and TCP flags are shown. If there were TCP options included, they are also printed.

E.2.3 Workload generation

Any application using the standard socket interface could be used as a workload generator for Seawind. The application can be attached to Seawind by a plugin script with a Seawind-compatible command line interface. By executing this script Seawind repeatedly executes the WLG tool automatically, as specified in the configuration generated by the user. The WLG applications can be started to run in parallel to make several simultaneous connections open through the link emulated by Seawind. Seawind is transparent to the WLG applications, so the networking code of the applications need not be modified in any way.

We mostly use slightly modified `ttcp` as a WLG tool. `ttcp` is a small tool generating bulk traffic in a single connection. The transmitting `ttcp` writes data blocks of the specified size to the TCP socket a specified number of times. The receiving `ttcp` reads blocks of the specified size from a TCP socket, until the other end closes the connection. Our modification of `ttcp` can also generate bidirectional bulk traffic, in which case there are transmitting and receiving `ttcp` processes at both ends of the connection. However, a single processes is used for both flow directions.

The following parameters are the most used ones in the ttcp-WLG:

- **buffer_length**. The size of the block to be read or written with a single call to the TCP socket interface. Using a value divisible by the TCP MSS to avoid the silly window syndrome is recommended.
- **number_of_buffers**. The number of *buffer_length* - sized blocks to be transmitted to the network. This parameter is used for the transmitting `ttcp`.
- **send_sock_buffer_size**. The size of the sending socket buffer. By default this is 32 KB.
- **receive_sock_buffer_size**. The size of the receiving socket buffer. By default this is 32 KB.

Usually only *buffer_length* and *number_of_buffers* have been specified in IWTCP tests. For example, to create 100 kilobytes worth of bulk data, we could set *buffer_length* to 1024 bytes and *number_of_buffers* to 100.