Effect of Delays and Packet Drops on TCP-based Wireless Data Communication

Panu Kuhlberg

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Panu Kuhlberg

Työn nimi — Arbetets titel — Title
Effect of Delays and Packet Drops on TCP-Based Wireless Data Communication

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	December 2000	81 p. + Appx.

Tiivistelmä — Referat — Abstract

This thesis represents a comprehensive study of the performance of TCP in an environment typical for slow wireless links. The effect of the excess delays and corruption-related packet drops are studied, and the performance implications of different bandwidths, including bandwidth asymmetry and variance during a connection, are examined. The link characteristics are emulated over a real-time software emulator that allows control over the link characteristics. The modeled links do not suffer from congestion, as there is no intermediate router, which can drop packets due to buffer exhaustion. The performance and behavior of different TCP enhancements are compared to a modified Linux TCP implementation, which is considered as the baseline TCP. The TCP enhancements include limiting the receiver's advertised window, the SACK TCP option, and an initial window of four segments. In addition, experiments with disabling the delayed acknowledgments are performed. As a result, detailed analyses of the TCP enhancements are provided.

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

Avainsanat — Nyckelord — Keywords
Wireless networks, mobile computing, performance, TCP, drops, delays

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

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Wireless networks are problematic environments for data communications. The nature of wireless links is quite different compared to wireline networks; their latency and error-prone characteristics make it a challenging environment for providing efficient transport. Excluding many satellite links and some other specific wireless links, the bandwidth is typically relatively small. This makes the optimization even more important for achieving better usability and throughput for end users [MDK+00].

Packet losses may occur in the wireless environment more often than in wireline networks because of multiple reasons. The weather conditions and surrounding buildings may cause interference resulting in packet losses as well as the hand offs in cellular wireless networks. Such conditions can also cause excess delays as the radio link layer may locally retransmit the corrupted segments. Examples of networks that exhibit asymmetry include wireless cable modem networks and the General Packet Radio Service (GPRS) [BW97]. In these networks the bandwidth towards the mobile client is usually much lower than in the opposite direction [BPK97].

Because the underlying IP-protocol [Pos81a] may drop or duplicate packets, the reliability must be offered at higher layers. The Transmission Control Protocol (TCP) [Pos81b][Ste95] is widely used by Internet applications. The reliability TCP provides over the 'best effort' IP networks is the key aspect that has made TCP so useful and the most widely used form of networking between computers [Hoe95]. The protocol specification is fairly old (RFC 793, dated September 1981) in terms of data communication. That fact has also caused problems because the networks of today are very different from the networks that TCP was built to work on. A lot of supplementary work and specifications have later been addressed to TCP.

It is a well known fact, that TCP performs poorly over wireless links [KRL+97] [CI95] [BSAK95]. TCP was specified to perform over fixed links where the latency and error probability are usually much lower than in the wireless links. Therefore, TCP interprets all packet losses as notifications of congestion in the network. The corrective actions taken by the sending TCP, such as lowering the packet transmission rate, reduce the load in intermediate links and network components. Those actions, however, tend to cause a suboptimal performance over wireless links; the packet losses and the delays usually are not caused by congestion, but are due to bit errors, local retransmissions or hand offs.

There have been various proposals that ameliorate the TCP performance in certain scenarios. Some of those enhancements are already included in the updated TCP specifications, such as the proposal for a larger initial window [APS99]. Others are still experimental recommendations, that need more studying before they can possibly be approved as a part of the TCP standards. Examples of such modifications are a limited receiver window [DMKM00] and disabled delayed acks [MDK+00].

Our intention was to test the performance of TCP over a slow wireless link that causes various errors or delays. In addition, the bandwidth was altered and could be asymmetric. The link was configured to roughly emulate a GSM link [Rah93], but this study was meant to be not just GSM specific, but rather "slow wireless" specific. Between the two hosts there were no intermediate routers that could drop packets due to buffer exhaustion.

We used a modified Linux TCP implementation as a reference implementation and we repeated the tests with different TCP modifications to achieve knowledge about their suitability for a wireless environment. Some other proposals, such Mowgli [KRL+97] and I-TCP [BB95], suggest enhancements that break the end-to-end connection by placing a TCP-aware proxy between the fixed and wireless network. We did not try to cover these proposals as we concentrated only on studying end-to-end TCP modifications in a plain wireless environment.

The empirical analysis of the tests is divided into five different areas. First, we concentrate on packet drops on the link and study the recovery of TCP in an error-prone environment. We study single packet drops as well as random packet drops. Second, we analyze the end-to-end performance when a predefined packet is excessively delayed on the link. Third, the effects of asymmetric and variable bandwidth are analyzed in an error-free link environment. Finally, the combinations of packet drops due to corruption and excess delays as well as asymmetric bandwidth in an error-prone environment are studied.

The contents of this thesis is the following. The wireless networking environment is described in Section 2. The Transmission Control Protocol and its problems over a slow wireless link are outlined and possible enhancements are introduced in Section 3. The test arrangements and test cases are described in Section 4 and the test results are presented and analyzed in Section 5. Section 6 gives a summary of this study. Appendix A provides the full test results. Appendix B describes the TCP in more details, Appendix C introduces our Baseline TCP that is used as a reference implementation when comparing the test results. Appendix D introduces different TCP enhancements for wireless environment more widely. Appendix E gives a detailed description of the *Seawind* emulator.

2 Wireless Networking Environment

This section describes wireless networking environments in general. There are many different network architectures where the wireless link may be used. We outline the common components and architectures used in such an environment. Our focus is on the Internet achitecture and in partical on the transport layer protocol that is used to achieve an end-to-end connection above the network layer. A traffic analysis in the Internet backbone reported that on an average 80 % of the flows used Transmission Control Protocol (TCP) as the transport protocol [TMW97]. Even if we consider a wireless networking environment, the mobile hosts most probably use TCP as the end-to-end protocol because the most popular Internet applications, namely World Wide Web (WWW), e-mail and file transfer, rely on TCP as the transport layer protocol [Lud00].

2.1 Wireless environment

Due to technical development there are many possible environments for personal data communication. There is, though, a major division between two environments; the characteristics and properties of wireless communications differ substantially from wireline communication.

Many different wireless technologies exist and they have different properties. The communication between end hosts was earlier limited by the fact that there had to be a physical wire between the hosts. Nowadays, the communication can be based on radio or micro waves, infrared or other media. In addition to the communication medium, the wireless links differ a lot in bandwidth, latency, and error characteristics. Wireless LANs (WLAN) offer a bandwidth of megabits per second [Goo97] and some wireless Wide Area Networks (W-WAN), such as GSM [Rah93], have a line rate of some tens of kilobits per second. The mobility of the user is one important issue that affects the usability of the wireless network. *Base stations*, often called access points, connect the mobile host to the fixed network through a wireless link. All the communication between the mobile host and the fixed network goes through a base station. A *hand off* occurs when a mobile host switches base stations. This may happen if the mobile host goes out of the coverage area of the former base station, for example. Stationary or slowly moving terminals are easier to serve as the hand offs from a base station to another do not happen often. A geosynchronous satellite can cover one-third of the Earth's surface and W-LANs provide high bandwidth over a range measured in tens of meters [Goo97]. [MDK$^+$00] gives a naming scheme for two different types of networks. Long Fat Networks (LFNs) are networks that have high latency (i.e they are ´long´) and the delay bandwidth product is large. This includes geosynchronous satellite links, for example. Long Thin Networks (LTN) have a high latency, but the link capacity is usually rather small. The LFNs and LTNs have similar characteristics, but do not share all the problem areas. For example, LFNs, such as many satellite links, may often have a delay-bandwidth product above 64 KBytes, in which case they cause additional problems to TCP. These problems are further discussed in [BBJ92] and [PS97].

In this study, we focus on LTNs, slow W-WAN links, that use radiowaves as the transport medium. GSM is a good example of such a network as it has millions of users across Europe and Asia, and roaming is possible between different operators. There exist other similar networks, such as Cellular Digital Packet Data (CDPD)[Sal99] and Code Division Multiple Access (CDMA)[Lee91]. Later in this study we use the word wireless network to mean a W-WAN of that type.

A wireless link may be used as an access link to an existing fixed network, usually being the Internet or a private intranet. In this scenario, the wireless link is often much slower than the accessed fixed network. There is usually a last-hop router in front of the wireless link, and it is between two totally different networks. One is slow and error-prone and the other is much faster and more reliable but it may loose packets due to congestion. As from an end-to-end point-of-view, it is a challenging environment to provide reliable and efficient data transportation. This network architecture is shown in Figure 1.

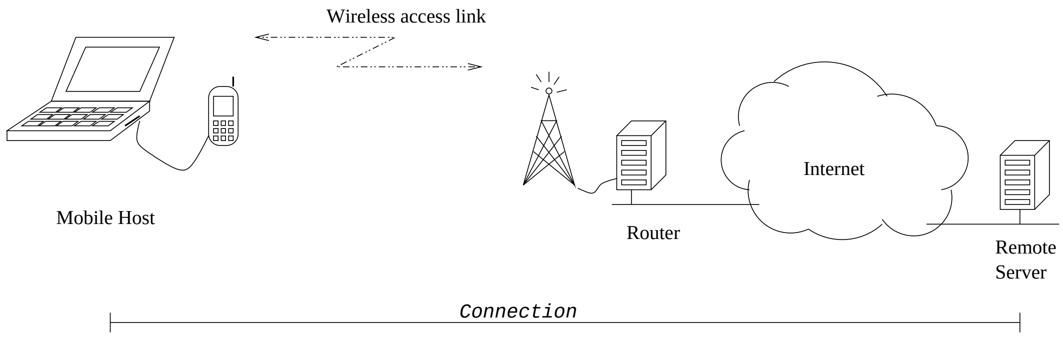


Figure 1: A wireless host connected to a remote server across the Internet

If we consider a proxy server that is located directly at the other side of the wireless link, we notice that the end-to-end connection may be split into two different parts that are communicating separately. For example, when the mobile host asks for a web page that is currently in the buffers of a web proxy, the proxy may send the data to the mobile host without accessing the global Internet. In this situation we have only one link that needs to be working efficiently, and the possible congestion problems of the fixed network are not relevant.

There are different proxies to solve various problems. A web-proxy may be used as a cache to give faster responses to the clients. This reduces the load of the remote host and entire external network. A Performance Enhancing Proxy (PEP) [BKG+00] is used to improve the performance of the Internet protocols on network paths where native performance suffers due to characteristics of a link or subnetwork on the path [BKG+00]. This provides a means to divide the connection between the two different networks. The split connections are used for easier optimization of the network environment, as the networks with different properties are separated. A simple and efficient solution might be to use only different TCP parameters over the wireless link. I-TCP [BB95] is an example of a single PEP split connection implementation.

Transparent proxies do not require any modifications to the end systems, transport endpoints, or applications. Some other proxies have a different protocol stack for the wireless link for better performance, like the Mowgli system [KRL+97], for example. Figure 2 shows the situation where the connection is split by using a proxy. The TCP parameters, or even the protocols, may vary between connections 1 and 2.

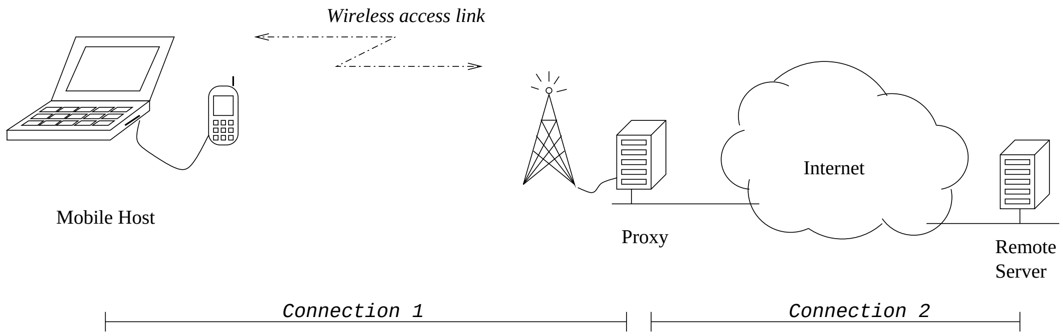


Figure 2: Split connection: a wireless host gets the data from the proxy

When using transparent PEPs, the mobile host is unaware of the split connection. From the mobile host point-of-view, the remote host is situated right next to the wireless link, as the proxy "imitates" the remote server. If the mobile host is aware of the proxy, the connection may be even more enhanced to achieve better performance over the wireless link. The connection parameters or the protocols used by the mobile host affect only the plain wireless link as the proxy establishes a new connection to the fixed side of the network. Thus, the problems of fixed networks are not to be considered at the mobile host as the proxy handles the external connections. Figure 3 shows the situation where the remote host is located right next to the wireless link. In principle, this is the same situation as the mobile host sees it, if a transparent proxy is in use.

Another common scenario includes a private intranet, which is accessed through a wireless link. The remote host is situated next to the last-hop router, and there is no need (or

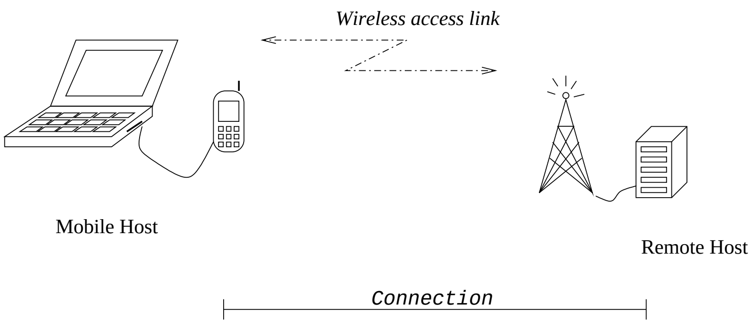


Figure 3: The remote host next to the wireless link

perhaps even any possibility) to access the global Internet. The router is subject to buffer exhaustion as the link capacity of the fixed network is typically much bigger than of the wireless link. Therefore the router may discard a packet as the queue limit expands over the storage space. The end-to-end connection is more dependent of the wireless link properties, as there are minimal amounts of packet losses due to congestion in the fixed network. This kind of environment is further studied in [Sar01], [Gur00] and [SP98], for example. The problems of this scenario are also outlined in [DMKM00].

This study focuses on an environment, where a mobile host is directly connected to a remote server over a wireless link. The link does not have intermediate routers that can discard packet due to buffer exhaustion, for example. The studied network is further described in Section 4.3.

2.2 Properties of wireless links

Usually the bandwidth of wireless links is some tens of kilobytes per second. This includes at least links with a bandwidth up to 56 Kbits/second. Such low-bandwidth links are widely deployed as dial-up modem links to access the global Internet. Our main interest is in GSM-like data links where the bandwidth is 9.6 Kbps [MP92]. It is clear that on such links all unnecessary retransmissions are undesirable. The bandwidth may also vary during the connection. High Speed Circuit Switched Data (HSCSD) in GSM may offer multiple timeslots for a single user and, using different encoding, the line rate up to 30-40Kbps can be offered. In addition to that, the new General Packet Radio System (GPRS) [BW97] networks can allocate one or several timeslots for a connection and thus alter the bandwidth.

Usually the propagation delay in the wireless networks is much longer than in wireline networks. The propagation delay means the time spent for a single bit to traverse from one host to the other. This delay is usually 100-300 ms in W-WAN networks. The latency decreases the user's ability to work interactively as the requests and replies are not available as quickly as when using a wireline connection. When calculating the round-trip time (RTT), the transmission delay has to be taken into account. Transmission delay is the time needed for the network to transmit a packet to the network with given bandwidth. The length of the transmission delay is proportional to the line rate and packet length. For example, a packet of 256 bytes (2048 bits) is transmitted to a 4000bits/second network in 0.512 seconds. Thus, the RTT is the sum of the propagation delays for both up- and downlink and transmission delays for up- and downlink.

Wireless networks are more error-prone than wireline networks[MDK+00]. There are two basic approaches to deal with the erroneous link. Forward error correction (FEC) is used to correct the bit errors by using an encoding algorithm that detects and corrects the bit errors at the receiving end. All the frames are delivered to the upper layer and end-to-end protocols need to deal with the errors that are not suppressed by FEC. This approach is called transparent data transmission in GSM. The other possibility is to use radio link level error detection (FEC) and link level retransmissions (Automatic Repeat Request, ARQ) to ensure that the frames are eventually delivered without errors. When a decoder observes a corrupted frame, the frame retransmitted. This scheme is called non-transparent data service. The maximum number of retransmits for a single frame is a link-specific parameter. The disadvantage of the link-level error detection is that it causes excess delays, while using the transparent mode the transmission rate is constant. There is a question, should the link level allow re-ordering. While waiting for the corrupted frame to be retransmitted, the following frames cannot be delivered if out-of-order delivery is not accepted on the link layer. This causes more delays as the subsequent frames are delayed even if they were not corrupted.

In GSM, the bit error rate is required to be less than 10^{-8} if a non-transparent mode is used [KRL+97]. Otherwise, in the transparent mode, the bit errors do not trigger local retransmissions on the link level and the bit error rate is allowed to be as high as 10^{-3}. As noted above, the radio link level retransmissions provide good reliability, but cause excess delays. Table 1 summarizes the two different approaches.

When the user is mobile, the coverage area of the base station may be exceeded. Therefore, a hand off should happen that switches the current base station to an other. During the hand off, the base stations need to exchange connection status information of the mobile host. Due to this procedure, the end hosts notice excess delays as the packets are not transferred during the hand off.

In general, wireless networks suffer from non-congestion related packet losses or excess delays due to link level error recovery and hand offs [BSAK95].

Table 1: Characteristics of Data Services in GSM

Data Service	**Technique**	**Transmission rate**	**Bit error rate**
Transparent	Forward Error Correction (FEC)	Constant	10^{-3}
Non-transparent	FEC + ARQ	Variable	10^{-8}

3 TCP Over a Slow Wireless Link

This section provides a short summary of the Transmission Control Protocol (TCP) [Pos81b], which is the transport protocol studied in the thesis. The problems with TCP over a slow wireless link are outlined. It is a well know fact that TCP does not perform in an optimal way over links that are error-prone and that have variable delays. The reader can find this conclusion from many different papers, such as [DMK$^+$00],[BSAK95] and [MDK$^+$00]. Finally, we describe possible enhancements that have been suggested in various studies. A more detailed description of TCP can be found in Appendix B.

3.1 Transmission control protocol

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 [Pos81b]. 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 reliability TCP offers is based on *cumulative acknowledgments* and *retransmission timers*. Each transmitted octet is assigned with a unique sequence number. The TCP receiver acknowledges the next expected octet. Due to the fact that the acknowledgments `(ACKs)` are cumulative, an `ACK` acknowledges all the previous octets, too. If an `ACK` has not reached the TCP sender within a certain period of time, the retransmission timer will expire and cause a retransmission timeout (RTO), and the segment is retransmitted. The TCP receiver sends a *duplicate acknowledgment* (dupack), which is an identical `ACK` to the previous one, when it receives an out-of-order (OFO) segment. The OFO segments are buffered at the receiver side until a continuous block of octets can be delivered to the application layer.

Slow start and *Congestion Avoidance* are the primary congestion control algorithms that govern the amount of data the TCP sender may transmit. *Congestion window* `(cwnd)` is the variable that gives the upper limit for the number of outstanding segments in the network that are not yet acknowledged. After the connection establishment, slow start is used to probe the capacity of the network by increasing the `cwnd` by one segment upon every new `ACK`.

After the `cwnd` has reached the slow start threshold (`ssthresh`), the congestion control algorithm is changed from slow start to congestion avoidance where `cwnd` is increased by one sender maximum segment size (SMSS) per round-trip time (RTT). Upon a RTO, *slow start* and *exponential backoff* are invoked; the `cwnd` is set to one MSS, and the retransmission timers are backed off by multiplying them by two.

Fast retransmit/fast recovery is an algorithm that uses the information achieved when three consecutive dupacks have arrived from the TCP receiver. This indicates that a segment has been dropped in the network. Thus, the segment is retransmitted without waiting for the retransmission timer to expire. The TCP sender enters fast recovery instead of slow start. When entering the fast recovery algorithm, the `cwnd` is halved. Every additional dupack is interpreted as an indication that a packet has left the network. Therefore, each dupack temporarily increases `cwnd` by SMSS. The fast recovery is exited when an incoming `ACK` acknowledges new data. In that case, `cwnd` is deflated back to the halved size and the connection continues by using congestion avoidance. Fast recovery algorithm is able to recover from one packet drop in a single window without waiting for a retransmission timer to expire [APS99].

The *NewReno* TCP modification enhances the fast recovery algorithm by introducing a way to recover from multiple losses from a single window without waiting for a RTO [FH99]. A retransmission upon a *partial acknowledgment* allows the TCP sender to recover from multiple packet losses without waiting for RTO to occur. A partial `ACK` is an incoming `ACK` during fast recovery that acknowledges new data, but not all outstanding segments that have been sent before the fast retransmit.

3.2 Problems due to corruption

The TCP sender makes deductions about the available bandwidth based on the feedback sent by the TCP receiver [DMK+00]. The incoming `ACK` segments report from successful transfer or packet losses. The retransmission timer is used to make these assumptions more precise. TCP cannot distinguish between a packet loss due to congestion and a packet loss due to corruption. As a result, TCP treats every packet loss as a sign of congestion in the network [DMK+00] [BSAK95] [BPSK96]. This principle has worked very well in the Internet community, and has allowed the Internet to expand to its current level without a drastic decrease in usability. However, it does not take into account the wireless networks that have higher bit error-rates. As stated in [DMK+00], the users connected to these networks may not be able to transmit and receive at anything like available bandwidth because their TCP connections are spending time in congestion avoidance procedures, or even slow-start procedures, that were triggered by a transmission error in the absence of congestion. Especially long fat networks (LFNs) suffer from this problem since the large delay bandwidth product requires a long slow start phase before the full capacity of the link can be achieved [PS97]. If a packet is dropped due to corruption at the beginning of the transfer, the probing algorithm becomes linear rather than exponential, and the time taken to fill the pipe grows even more.

Once a packet loss is detected by receiving three dupacks, the `cwnd` is halved, and thus, the transmission rate is decreased by 50 %. If the packet loss was due to congestion, the reduction of the transmission rate would be reasonable, but often in wireless environments, the packet losses occur due to corruption. If the congestion window was much larger than the delay-bandwidth product, the single packet drop does not have any effect on the throughput as the pipe is kept full. The problem arises, when several packets are dropped during the connection, each of them halving the current `cwnd`. This is a common case on lossy wireless links. While the wireless link's capacity may be quite small, the degradation of the transmission rate due to non-existent congestion is undesirable. If the `cwnd` is smaller than four segments, the recovery using the fast retransmit/fast recovery algorithm is not possible, as there are not enough segments in the network that would trigger three dupacks. In this situation, the TCP sender would have to wait for a RTO and go to a time-consuming slow start. It has been reported that over 85% of RTOs are due to small congestion windows that prevent fast retransmissions [LK98]. Although, the study was made in 1995, the results indicate that small `cwnd`s contrive real problems on the performance, as fast retransmissions are not always possible.

The NewReno TCP modification is able to recover from multiple packet losses from a single window without going into slow start. This ameliorates the situation, as an RTO can be avoided. Nevertheless, the fundamental problem of wireless networking is not solved, as, in the absence of congestion, the available bandwidth may be underestimated if the transmission rate is lowered due to corruption [Lud00].

3.3 Problems due to excess delays

The TCP retransmission timer sets an upper limit for the RTT. It is derived from RTT samples, taking into account the RTT variation (see Appendix B for details). If the RTT suddenly increases, and a segment is excessively delayed, the RTT estimation may be exceeded. As a result, a RTO occurs and the segment is retransmitted. Because the segment was only excessively delayed, not dropped, the retransmission are needless. Nevertheless, when RTO occurs, the TCP sender interprets it as a notification of congestion in the network and lowers the packet sending rate by invoking slow start with a `cwnd` of one segment, and the TCP sender is forced into a *go-back-N* retransmission mode [LK00]. In this scenario the `cwnd` reduction is not desirable because there was no congestion. The go-back-N behavior triggers a large number of retransmissions that should be avoided; when retransmitting after a spurious RTO, the sender retransmits the whole window of data unnecessarily as no packet were dropped. This scenario may lead to a yet greater loss of performance, as the retransmitted segments generate dupacks that may trigger a false fast retransmit. This scenario is further discussed and analyzed in the delay tests in Section 5.3.

TCP suffers from *retransmission ambiguity* [KP87]. It means, that the TCP sender does not know whether an `ACK` arrives for a retransmitted segment or for the original one that was interpreted to be dropped. In the case where a full window is unnecessarily retransmitted due to a spurious RTO, the incoming `ACK`s that are in fact acknowledging the first window of data are misread as acknowledgments for the retransmitted window.

If the connection is in congestion avoidance while the RTO happens, the number of transmitted segments in the network grows aggressively due to go-back-N behavior. The TCP sender invokes slow-start that transmits twice as many segments into the network (assuming each segment is acknowledged separately) while the original segments are draining from the queue [LK00]. This may lead to packet drops at the intermediate routers, as they become overloaded by the amount of new packets. As the studied environment does not include a intermediate router, this issue is not considered. Similar problems may occur if the link interface buffer has limited amount of storage space. However, in this study the link interface buffer is assumed to have unlimited capacity.

3.4 Problems due to asymmetric and variable bandwidth

As defined in [BPK97], *a network is said to exhibit network asymmetry with respect to TCP performance, if the throughput achieved is not solely a function of the link and traffic characteristics of the forward (i.e downlink) direction, but depends significantly on those of the reverse (i.e uplink) direction as well.* It is stated that asymmetry does include latency and packet error rate as well as bandwidth. We declare that in this study we concentrate on bandwidth asymmetry only.

The asymmetry in pure bandwidth is not the asymmetry that directly affects TCP performance. *Normalized asymmetry ratio, k*, is defined as the ratio of the pure bandwidths divided by the ratio of the packet sizes used in uplink and downlink [LMS97]. For example, if the bandwidth for the downlink is 1Mbps and for uplink 10Kbps, the pure bandwidth ratio is then 100. If the data segment size is 1000 bytes and the `ACK` segment is 40 bytes, their ratio is 25. So, k is 100/25=4. As TCP is *self clocked* [Jac88], it uses the arrival rate of incoming `ACK`s to control the packet flow in the opposite direction. Therefore, any interference in the feedback could impair the performance. This means that if there are more incoming `ACK`s than one for every $k = 4^{th}$ data segment, the uplink direction will get saturated before the downlink does [BPK97].

If the delayed ack algorithm is not in use and the `ACK`s are not dropped, it has been noticed that whenever $k > 1$, the TCP self-clocking mechanism breaks down [BP00]. When two back-to-back sent segments arrive at the receiver, the time interval between the packets correlates with the downlink bandwidth. The `ACK`s that are sent in response, should maintain the same spacing all the way back to the data sender, letting the TCP sender send new data at the same spacing. As the bandwidth of the uplink (i.e. `ACK` path) is much lower than of the downlink ($k > 1$), the spacing of the `ACK`s is not the same as it was while sending them when they reach the sender. The performance on the link is no longer dependent of the downlink path, but by the rate of incoming `ACK`s. The growth of the `cwnd` slows, too.

If an intermediate router on the `ACK` path has limited amount of buffer space, the router queue gets full as the transmission window grows. As a result, some `ACK`s will get dropped. This causes bursts at the sender side, as each `ACK` acknowledges more data than usually. Moreover, the bursts may cause congestion on the data path. The lost of `ACK` segments lead to less efficient use of fast retransmit/fast recovery algorithms, as the information based on dupacks is not as accurate [BP00]. Since the studied environment does not include an intermediate router, and link interface queue is assumed to have unlimited capacity, their effects are not further discussed.

In general, the slow uplink path that carries the acknowledgments can significantly slow down the growth of the TCP sender congestion window and affect the utilization rate of the link [BPK97]. More difficulties arise if the uplink path may drop `ACK`s due to congestion in the intermediate routers, for example [BPK97] [LMS97] [BP00].

The variation in the bandwidth during the connection makes the RTO calculations inaccurate and may cause unnecessary retransmissions. If the link's transmission rate drops significantly during a connection, the TCP sender might end at a RTO. This scenario would be similar to the excess delay described in the previous section.

3.5 Suggested enhancements

There are many different proposals for enhancing TCP performance over wireless links. In general, they can be classified into three different categories [BPSK96]. End-to-end modifications use well-selected TCP parameters and options that are suitable for wireless links. This usually requires that the (wireless) host is aware of the link characteristics. Secondly, split-connection protocols, that break the end-to-end connection into two parts, separate the wireless link from the wireline network for greater control of the wireless link. Thirdly, link-layer protocols that provide local reliability are used to minimize the problems due to bit-corruption. We describe the various enhancements suggested in the litterature. For a more detailed description about end-to-end modifications employed in this study, please refer to Appendix D.

For link-layer protocols, one solution is that the link-layer provides a *persistently reliable error correction* [Lud00]. That is, all packets are retransmitted until they are correctly received. This way, the number of transmission errors due to corruption can be reduced. If the number of local retransmissions is limited, the link is semi-reliable. For example, the Radio Link Protocol (RLP) [ETS95] in GSM tries to deliver all packets without errors by using selective rejection and checkpoints and local retransmissions. The number of retransmits is a network-specific parameter that can be configured [Lud00]. When the retransmission threshold is reached, the link is reset, and all packets that are currently at the link-layer are discarded. This leads to wasting available bandwidth because the transport layer has to retransmit the packets in the same manner as if there would be no link-level error recovery. However, if the link-layer persistently tries to recover from errors, the transport layer might end up retransmitting the packets due to a RTO, as the local retransmissions take time.

When the connection is split into two distinct parts, wireless link and fixed network, an intermediate *proxy* may take advantage of knowing the link properties. Indirect TCP (I-TCP) [BB95] is an example of such a proxy that is functioning at the transport level. The *Mowgli system* [KRA96] offers an indirect approach that involves all protocol layers. It gives an option to use a wireless-specific protocol that replaces the TCP/IP stack. The header overhead is reduced to 1-3 bytes in common scenarios. It can also work transparently, so that regular applications can be used without recompilation.

Split solutions may provide better throughput and performance [BKG+00]. However, such proxies are not the preferred solution for wireless networking in general as the end-to-end connection sometimes might be violated. It is suggested that split connections should not be used unless there are no end-to-end mechanisms available that can provide similar performance enhancements [BKG+00].

ACK *congestion control* (ACC) [BP00] introduces mechanisms for the receiving end to minimize the problems due to bandwidth asymmetry. It is a TCP enhancement that limits the number of ACKs sent by the TCP receiver. It is not a pure end-to-end modification, as it also requires the intermediate routers to apply active queue management. If the network is asymmetric, the bottleneck router on the ACK path is subject to buffer exhaustion. ACC makes use of the *Explicit congestion notification* (ECN) [FR99] bit in the packet headers. In conjunction with the Random Early Detection algorithm (RED)[1] [FJ93] at the router, the router can inform the TCP sender that the router buffers are becoming full by setting the ECN-bit in the packet headers. The data sender echoes the ECN-bit in the following segment to the TCP receiver. Therefore, the TCP receiver understands to reduce the rate at which it sends ACKs. It is proposed that the TCP receiver should send an ACK for every d segments. The variable d is the delayed-ack factor that is dynamically varying according to the ECN-bits in the packet headers. Once a packet with the ECN-bit is received, the receiver increases d multiplicatively. Similarly, for each subsequent RTT during it does not receive a packet with the ECN-bit, d is decreased linearly. Thus, d imitates the behavior of the regular congestion control.

The limited transmit TCP enhancement [ABF01] proposes mechanisms that allow the TCP sender to recover from a packet loss even if the cwnd is less than four segments. This scheme has been suggested earlier by the authors of Net Reno [LK98]. They suggest that the TCP sender should send a new segment upon the first two dupacks. The receiving end reacts to the new segments by sending dupacks that invoke a fast retransmission at the sender side.

The Eifel algorithm [LK00] was designed for making TCP robust against spurious retransmissions. As TCP suffers from *retransmission ambiguity*, it cannot distinguish an ACK for the original transmission of a segment from the ACK for its retransmission. Eifel makes use of the TCP timestamp option [BBJ92] to correctly separate the different ACKs.

The limited receiver window [DMKM00] is one possible enhancement in end-to-end category that does not need any changes at the sender side. The delay bandwidth-product is usually quite small in slow wireless links. If we consider a link that has a RTT of 0.5 seconds while the bandwidth is 9600bps, the capacity of the link is 600 bytes in maximum. Typically, the default receiver's advertised window is quite big, being 32Kbytes or even 64Kbytes. This means, that the TCP sender is able to probe for a non-existent bandwidth by growing the congestion window above the delay bandwidth product. An unnecessarily

[1] RED gateways detect the upcoming congestion by observing the moving average of the queue size.

large congestion window inflates the RTO and may also cause a packet loss in the intermediate routers due to buffer exhaustion [DMKM00]. All packet drops should be avoided on a slow link, as the transmission rate is dropped by 50 %.

Selective Acknowledgment (SACK) [MMFR96] is a TCP option that is currently in the "Standards Track" category in IETF[2]. As the regular fast recovery algorithm is not able to recover from multiple packet losses in a single window without going into slow start, its performance over error-prone wireless links is suboptimal. Due to the fact that the TCP acknowledgments are cumulative, there is no way to acknowledge separate blocks of segments that have been correctly received, but are out-of-order. The SACK TCP option [MMFR96] gives a possibility for the TCP receiver to inform the data sender which segments have been received. This information can be "piggybacked" in a duplicate ack segment that is sent when an out-of-order segment is received. It is recommended by [DMK+00] for erroneous links.

The larger Initial Window [APS99] is an enhancement that needs more evaluation. The congestion control specifications require the TCP implementations to use an initial window (IW) of one or two segments [APS99]. However, experimental TCP extensions are allowed to use a larger initial window, such as three or four segments. When using an IW of one segment, the slow start phase is time-consuming. A packet loss at the beginning of the connection would lead to a RTO, as there are not enough packets in the network to produce three duplicate acknowledgments that would trigger the fast retransmit/fast recovery algorithm. When using a larger IW, the congestion window would be faster in a "safe" size (i.e four segments or more). This optimization is further discussed in [MDK+00].

Disabled delayed acks are used to increase the `cwnd` faster during the initial slow start. During the slow start, each incoming `ACK` increases the congestion window by one SMSS [APS99]. If delayed acknowledgments are in use, the number of `ACK`s is smaller, as one `ACK` acknowledges two segments. By disabling the delayed acknowledgments at the receiver side, the TCP sender receives an `ACK` for each transmitted segment.

[2]Internet Engineering Task Force is the standardization body of Internet standards.

4 Test Arrangements

This section describes our test arrangements for running the TCP performance test with the *baseline TCP* and different TCP modifications.

4.1 Test objectives

The objective of this thesis is to study the TCP performance implications of those link characteristics that are typical for slow wireless links. And, in addition to that, to study the effect of various TCP parameters and to implement some experimental TCP performance enhancements and study the impact of these enhancements.

We measure and analyze the effects of an unreliable link, which causes packet losses due to corruption, and a persistently reliable link, which cause excessive delays instead of packet losses, to a baseline TCP implementation that represents the "best current practice" in the Internet community. We then employ various mechanisms to improve TCP performance and compare the measured performance to that of the baseline TCP. The network characteristics are emulated using *Seawind* emulator. For a detailed description about *Seawind*, please refer to Appendix E.1.

It is clear that there is no one single modification that gives an optimal result for all the different test cases. However, we try to get a clear picture about the effects of different modifications. A TCP modification has to work well in all test cases, and can especially not lower the throughput drastically in certain cases. Using one modification we may get an almost optimal performance on some selected test cases, but bad with other test cases.

4.2 Test environment

We used three different computers on a closed LAN to run the tests. Two of the computers were working as end hosts and the *Seawind* emulator was running on the third host. The computers are equipped with a 400 MHz Intel Celeron processor. The end hosts are running Linux Red Hat 6.0 as their operating system and the emulator is running on a Linux Red Hat 6.2. As the emulation is done in real-time, it is important that the whole testing environment does not produce wrong results due to performance capacity problems. Therefore, we dedicated these computers to our testing purpose only and there were no CPU consuming applications running. There was no other disturbing traffic in the network. While the capacity of the LAN was 10Mbps, Seawind stored and forwarded the packets at a maximum rate of 40Kbps. We believe that these test conditions were sufficient to guarantee the correct results and avoid the wrong analysis due to failed test execution. Due to the fact that we are not using a real-time operating system, the accuracy of the various delays cannot be fully guaranteed. Therefore, the emulator writes a warning to a log file, if a scheduled delay was longer than planned. We used a warning for all the events, that overslept for more than 10 ms. For detailed description about *Seawind*, please see Appendix E.1. Figure 4 shows the testing environment.

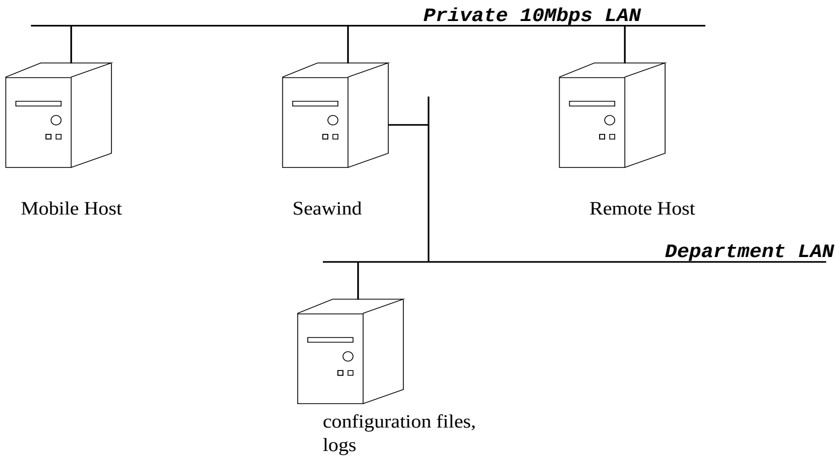


Figure 4: The test network

Figure 5 shows the protocol layering of the test environment. Both end hosts are running *ttcp* as the workload generator (highlighted in the figure). The real-time *Seawind* emulator is in between the two workload hosts. At the workload hosts, normal TCP/IP packets are encapsulated at the link layer in Point-to-Point (PPP) [Sim94] frames and transmitted to the *Network Adaptor* (NPA). NPA encapsulates a PPP frame to a regular TCP/IP packet, and sends it to the emulator. Thus, the packet structure between workload hosts and the emulator is TCP/IP/PPP/TCP/IP. Seawind detects the original IP packet that includes the TCP payload created at the workload generator (*ttcp*) by decapsulating the topmost TCP/IP packet. After that, the *Simulation Process* (SP) invokes the mechanisms to emulate the wireless link according the given parameters. *SP* is the core of Seawind that emulates the wireless link. The *ttcp*s are unaware of the encapsulation, and transmit all segments as if it was a regular TCP connection. They are also unaware of the *Seawind* emulator; they send and receive the TCP segments as if the link was a slow wireless link with emulated parameters.

The emulator is run on a Linux kernel 2.2.17. The possible TCP implementation problems in kernel 2.2.17 are not relevant, as the end-to-end TCP connection that was under study, was between the end hosts. Therefore, all the TCP enhancement are implemented only at the workload hosts as they are the acting hosts in TCP point-of-view. The TCP implementation issues are discussed later in this section.

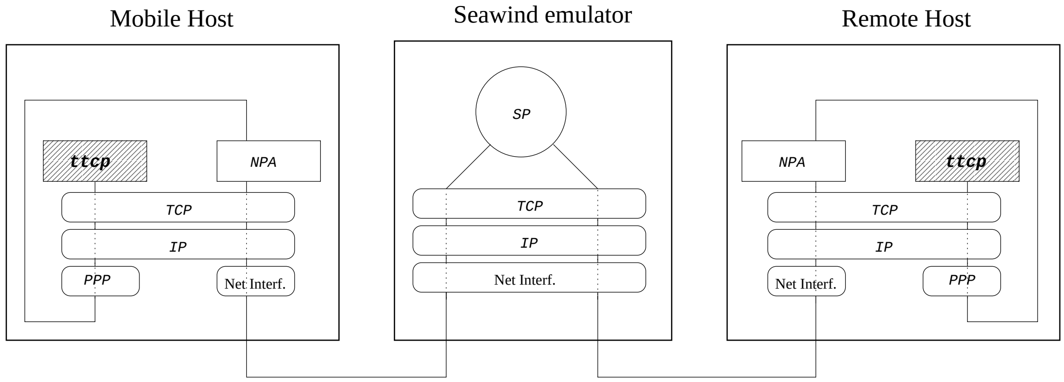


Figure 5: Seawind protocol stack

The measurement data was collected from various points. Seawind produced its own log file that keeps track of all events that the emulation process executes, and *tcpdumps*[3] [JLM97] were collected from end-hosts and the emulator. We used *tcptrace* [Ost] and *tracelook* [PS98] for analyzing the tcpdumps. Some of the measurement tools we made ourselves to achieve statistics, such as percentiles and minimum and maximum values of certain variables. *Matlab* [Mat97] was used for graphical presentation of the results.

On end hosts we use a modified Linux kernel, which is based on the Linux kernel 2.3.99-pre9. It was the latest development kernel version available at the time when we decided to start implementing new TCP modifications. The reasons for choosing Linux as the operating system were obvious. We had good previous knowledge of Linux, it is a popular operating system, and the source code is available. We are aware of the common belief that the Linux TCP implementation is not following all the standards and specifications. Therefore, we have run a considerable amount of preliminary tests to gain knowledge about the possible problems. The misbehavior problems we have found are corrected, and we believe that our kernel implements TCP in a way described by [Pos81b], [Bra89], [APS99] and [FH99]. We have also implemented some of the modifications ourselves, such as larger initial window, disabled delayed acks, and receiver window sharing. We call our TCP implementation *Baseline TCP* as it represents the current "state-of-the-art" implementation and we think it follows the given TCP specifications. Thus, it can be used as a reference when comparing the performance of different TCP modifications. The features of *Baseline TCP*, including a list of corrected bugs and implemented modifications, are described in Appendix C. Appendix E describes the test environment, arrangements and the *Seawind* emulator in more detail.

[3]tcpdump is a program that collects all the data going through a defined network interface

4.3 Wireless link modeling

Because modeling a specific network architecture is not easy due to the number of possible variables, we used a general model for modeling the wireless network. The basic idea is to emulate only the wireless link, but no other network components. The properties of the emulated network are selected to be similar to GSM. However, this study is not GSM-specific, but rather slow W-WAN specific as the details in GSM architecture are not taken into account.

The emulated network consists of two hosts that communicate over a slow wireless link. The link may contain packet drops, and excess delays due to hand offs or local retransmissions. The excess delays and packet drops are considered only at the downlink direction, although some limited set of tests are run over a link that may drop packets at the uplink direction, too. The link does not allow packet re-ordering. The *link send buffer* (LSB) and *link receive buffer* (LRB) are for buffering the out-of-order packets if packets are locally retransmitted due to errors. We assume that there are no intermediate routers between the two hosts.

Figure 6 shows the emulated wireless link. The hosts are called *Mobile Host* and *Remote Host*. The link between the two hosts is a slow wireless link. The usual emulated bandwidth is 9600bps, but this varies in some test cases. The propagation delay is set to 200ms, which is typical for such wireless environments.

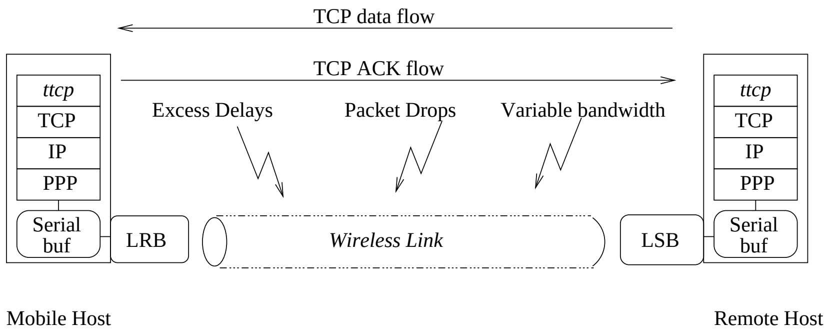


Figure 6: Emulated network

Figure 7 illustrates the emulation setup in Seawind. The serial buffers of the end hosts were emulated in the input buffers of Seawind. A serial buffer overflow was not considered, as the input buffers were set to be arbitrary large. Both link send and receive buffers were set to 1220 bytes. If a packet is excessively delayed, a burst of segments will follow; the packets in the link receive buffers are delivered to the TCP receiver back to back. This is a simplification of the real world; in some cases, as with many W-WAN link interfaces, the packet are delivered at the rate of the link interface, not at an unlimited rate.

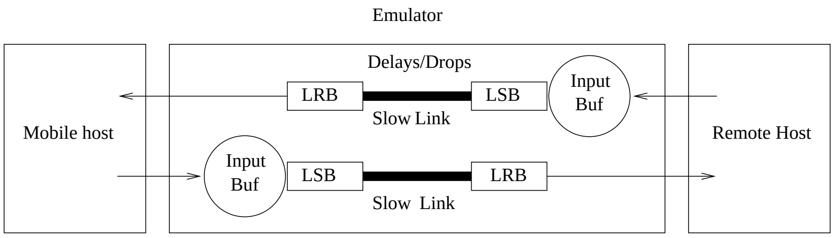


Figure 7: Emulation setup

Workload and ttcp parameters

Our main focus was on bulk data transfers, therefore we did 100 Kbytes transfers. For the data we gathered, it was easy to calculate the throughput for smaller transfers, too. A single unidirectional connection was used in all cases. Because each end represented a wireless host and there are no routers in between, it did not have any effect which way we transferred the data. For naming purposes only, we call the other end mobile host and the other end remote host and do all the unidirectional transfers in the downlink direction (i.e. from remote host to mobile host).

The TCP senders socket buffer size was left to its default value that is 64 Kbytes. However, Linux's TCP does not advertise windows larger than 32 Kbytes, so that is the limit the other end gets to know. The TCP receiver's socket buffer size is one of the enhancements we tested and therefore its value can be changed. In most cases the default size is used.

When the NPA encapsulates the IP packets that contain the workload, it sets a *Maximum Transfer Unit* (MTU) for the PPP frame. The MTU is set to 296 bytes, which leads to a MSS of 256 bytes[4]. A MSS of 256 bytes is typical for slow links [Jac90]. The PPP overhead is typically eight bytes[5]. Thus, the PPP frames that are transmitted over the wireless link are 304 bytes.

4.4 Tested TCP enhancements

We used our *Baseline TCP* as our reference TCP implementation. Refer to Appendix C for full description of included algorithms and parameters. The tested TCP enhancements are outlined below.

[4]A MTU of 296 bytes includes 256 bytes of TCP payload, 20 bytes of TCP headers and 20 bytes is IP headers.

[5]The PPP header overhead may vary due to byte-stuffing.

SACK option The use of SACK is highly endorsed in situations where multiple segments may be dropped in a single window [MDK+00]. Therefore, it is a good modification to be tested against Baseline TCP and other enhancements. It has been stated that the reference TCP implementation (i.e. the Baseline TCP) should include the expected future TCP algorithms [AF99]. SACK is clearly such a TCP enhancement, as it is in the "Standards Track" category of IETF. According to a resent study, 40% of the host connected to a web server used the SACK option [All00]. However, the SACK was not considered to be included in the Baseline TCP, as it is not implemented in a regular way. The Linux SACK TCP makes use of *forward acknowledgment* algorithm (FACK) [MM96], and in addition, the details of the implementation may have an impact on the performance.

Limited receiver window We selected two window sizes to be tested. Because the delay-bandwidth product of the link is less than three segments, a 2 Kbytes advertised window (i.e a window of eight segments, when using a MSS of 256 bytes) is big enough to fill the pipe. If there are many packet drops during a connection, the `cwnd` may be suppressed into a value that is less than four segments. In such a case the fast retransmit cannot happen. Therefore, we also tested the effects of a receiver window of 4 Kbytes. The limited receiver window is mentioned as a possible enhancement for slow links [DMKM00].

Disabled delayed acks This modification acknowledges some predefined number of segments separately without waiting for the delayed ack threshold to be exceeded or another data segment to arrive. Disabling many delayed acks at the beginning of the connection is not encouraged, but we wanted to have a clear case whether it helps the TCP sender to exit the early slow start faster, and would it be justified to use such a modification in the future. This modification is used only in limited test cases, where the bandwidth is increased, because using a bandwidth of 9600bps, the transmission delay for a 256 byte segment is more than 200ms, which is the delayed ack threshold. An informational IETF RFC [MDK+00] suggests to acknowledge the first incoming segment without any delay on receivers behind a long thin link. This enhancement should be used only with a justified reason and if the consequences are well understood; it is stated in TCP Congestion Control Specifications that "delayed acknowledgments *should* be used by a TCP receiver".

Larger initial window The initial window of four segments was tested. The current specifications allows using an initial window of four segments in experimental implementations as we are using. The current understanding is that it may be beneficial to use a larger initial window than two segments [PN98][SP98].

Maximum segment size The default MSS is set to 256 bytes. However, limited number of tests are run to gain knowledge about the effects of a larger MSS of 512 bytes. This modification is not considered as a TCP enhancement.

Table 2 outlines the parameters of the TCP enhancements.

Table 2: Summary of the TCP enhancements

Item	TCP enhancement						
	Baseline	**SACK**	**Larger IW**	`rwnd`**2KB**	`rwnd`**4KB**	**Dis.**`dACKs`	**MSS512**
`NewReno`	Yes	-	Yes	Yes	Yes	Yes	Yes
`Initial window`	2	2	4	2	2	2	2
`SACK`	Off	On	Off	Off	Off	Off	Off
`rwnd`	32KB	32KB	32KB	2KB	4KB	32KB	32KB
`MSS`	256B	256B	256B	256KB	256KB	256B	512B

4.5 Test cases

The tests are divided into five major categories according to emulated link characteristics: packet drops tests, delay tests, combined packet drops and delays, bandwidth tests, and combined bandwidth and packet drops tests. This section describes the exact tests that are run. To achieve correct information for the statistics, the number of test replications is 20. If a test case includes randomness in the emulation, the number of the replications is increased to 50.

Packet drop tests

In these test cases we analyze how the different modifications affect the performance when a data packet is lost. First, we drop only a single packet during the transfer. The dropped packet is altered to gain knowledge about possible problems. The different packets dropped are: first SYN, first data segment, second, third, fifth, seventh, tenth, 15th, 20th, 30th, 40th, 60th, 80th, 100th, 120th, 140th, 160th, 180th, 200th, 240th, 300th, 340th and 400th data segment. Next, we use random drops according to a uniform distribution where the packet drop rate is defined. The data packet drop rates are selected to simulate good, mediocre and bad link conditions. Packet drop rates for the data segments are 2%, 5% and 10%, respectively. In addition to the Baseline TCP, the tests are repeated with different TCP enhancements to study their impact on the performance. The included TCP enhancements are SACK, larger IW of four segments, limited `rwnd` of 2 Kbytes and 4 Kbytes. However, single packet drop tests are not run with a limited `rwnd` of 4 Kbytes. The random tests are repeated by using a larger MSS of 512 bytes. In addition, all the random packet drop tests are repeated by applying the packet drops also to the uplink direction. Table 3 summarizes the packet drop tests. *UL* and *DL* denote uplink and downlink, respectively.

Table 3: The packet drop tests

Packet(s) dropped	**Tested TCP enhancements**
A single data packet (DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB
2% prob.(DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB
5% prob.(DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB
10% prob.(DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB
2% prob.(UL+DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB
5% prob.(UL+DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB
10% prob.(UL+DL)	Baseline TCP, SACK, IW=4, `rwnd`=2KB, `rwnd`=4KB

The number of different single packet drop tests is 23. Each test cases were executed with four different TCP enhancement (including the Baseline TCP). In total, each 92 different single packet drop test is replicated 20 times producing 1840 basic tests.

The number of different random drop test cases is six. Each test case is executed with five different TCP enhancement, and repeated with a MSS of 512 bytes. In total, each 60 different random packet drop test is replicated 50 times producing 3000 basic tests.

Delay tests

In these tests we delay the packet in the wireless link before it reaches the receiver. We use three different lengths for delayed packet in different test cases. In the fist test case the delay is set to be little less than a delay that would cause a retransmission due to RTO. In the second test case the delay will cause one retransmission for the delayed packet and the third case the delay will cause two consecutive retransmissions for the delayed packet. The delay thresholds were achieved by running preliminary tests.

The goal is to find a threshold for the retransmission timer to expire and monitor the behavior of the TCP sender. We remind that all retransmissions in this environment are unnecessary because no packets are lost, only delayed. Therefore, TCP may behave improperly or ineffectively.

The packets that are delayed are: first SYN, first data segment, third, fifth, seventh, tenth, 15th, 20th, 30th, 40th, 60th, 80th, 100th, 120th, 140th, 160th, 180th, 200th, 300th, and 400th data segment.

The TCP settings we used to run the tests are:
1) Baseline TCP (IW = 2, MSS 256)
2) limited `rwnd` = 4 KB
For the limited `rwnd` tests we run only the tests that invoke one RTO. Table 4 summarizes the delay tests.

Table 4: The delay tests

Packet delayed	**Delay length**	**Tested TCP enhancements**
A single data packet (DL)	less than RTO	Baseline TCP
A single data packet (DL)	one RTO	Baseline TCP, `rwnd`=2KB
A single data packet (DL)	two consecutive RTOs	Baseline TCP

The number of different single packet drop tests is 20*3=60 of which 20 is executed with two TCP enhancement, and 40 with only the Baseline TCP. In total, each 80 different delay test is replicated 20 times producing 1600 basic tests.

Combining errors and delays

In these cases we combine errors and delays. The sending TCP will retransmit packets due to RTO (i.e excess delays) and duplicate ACKs (i.e. lost packets). The errors are created using packet drop rates of 2%, 5% and 10%. The probability of a delay is 1% and if it happens, the length of the delay is always 6 seconds. The length of the delay was chosen according to the preliminary tests and it usually causes a retransmission timeout.

The TCP enhancement used with these tests are:
1) Baseline TCP
2) Larger initial window (IW) of four segments
3) SACK-option enabled
4) limited receiver's advertised window `rwnd` = 4KB

Table 5 summarizes all the different tests. Each of these tests is run 50 times because of the randomness. All the delays and packet drops are executed only in the downlink direction (i.e. data path). The number of test cases is 12 and the number of basic tests is 600 in total.

Table 5: Tests with packet drops and delays

Packet drop rate	**Delay prob.**	**Tested TCP enhancements**
2%	1%	Baseline TCP, IW=4, SACK, `rwnd`=4KB
5%	1%	Baseline TCP, IW=4, SACK, `rwnd`=4KB
10%	1%	Baseline TCP, IW=4, SACK, `rwnd`=4KB

Bandwidth tests

Tests with different bandwidths are made. The emulated link does not have excess delays or packet drops. The bandwidth may change during the connection and static asymmetric bandwidth is tested. We monitor only the effects of one rate change in the bandwidth during the connection. The change is made in two different phases of the connection: slow-start and steady state.

The different static bandwidths that we use to run the tests are:
- (1/3) * 9600bps i.e 9600bps for the uplink and 28800bps for the downlink
- (2/2) * 9600bps (i.e. 19200bps/19200bps)
- 14400bps/38400bps and
- (2/2) * 14400bps (i.e. 28800bps/28800bps)

The static bandwidth tests are repeated with the following TCP enhancements:
1) Baseline TCP
2) IW = 4
3) `rwnd` = 2 Kbytes
4) `rwnd` = 4 Kbytes
5) Disabled delayed `ACK`s up to two, four, six and eight segments. In addition, disabled delayed `ACK`s are applied for the whole connection.

The disabled delayed `ACK`s are performed only over a 14400bps/38400bps link.

Table 24 gives a summary of the static bandwidth tests. The first column of the table, *Bw (uplink)*, gives the bandwidth of the uplink on the emulated link and the second column, *Bw (downlink)*, gives the downlink bandwidth. The abbreviation *Dis. d*`ACK`*s* stands for the TCP enhancement (5), disabled delayed acks. The normal behavior of delayed acknowledgments for the Baseline TCP is explained in Appendix C.

The number of different static bandwidth tests cases is 24, when taking the different TCP enhancement into account. Each test is repeated 20 times. In total, 480 basic tests are run.

Table 6: Static bandwidth tests

Bw (uplink)	Bw (downlink)	Tested TCP enhancements
9600bps	28800bps	Baseline TCP, IW=4, `rwnd`=2KB, `rwnd`=4KB
19200bps	19200bps	Baseline TCP, IW=4, `rwnd`=2KB, `rwnd`=4KB
14400bps	38400bps	Baseline TCP, IW=4, `rwnd`=2KB, `rwnd`=4KB, Dis. `dACKs`
28800bps	28800bps	Baseline TCP, IW=4, `rwnd`=2KB, `rwnd`=4KB

The variable bandwidth tests are:
- from (1/3) * 9600bps to (1/1) * 9600bps,
- from (2/2) * 14400bps to (2/2) * 9600bps,
- from (2/2) * 14400bps to (1/1) * 14400bps,
- from (2/2) * 9600bps to (1/1) * 9600bps and
- from (1/1) * 14400bps to (1/1) * 9600bps

In variable bandwidth tests the bandwidth is changed at two different moments: during slow start and during steady state.

The tested TCP enhancements are:
1) Baseline TCP
2) Larger IW of four segments
3) Limited `rwnd` of 4 Kbytes

Table 7 shows the variable bandwidth tests. The first column, *Bw1*, tells the bandwidth on the link (for uplink/downlink) before the change. The *Bw2* column outlines the bandwidth on the link after the bandwidth has been changed. All the changes are from a faster link to a slower one as it is a normal behavior on some wireless links, such as GSM, when the link conditions becomes worse.

There are 24 different variable bandwidth tests and using 20 replications the total number of basic tests is then 480.

Combining errors and bandwidth

In these cases we combine errors and different static (and asymmetric) bandwidths. The errors are created on random-basis according to different packet drop rates. The packet drop rates are 2%, 5% and 10%. Packets are dropped only on the downlink direction, so no `ACK`s are dropped. The bandwidths we are using in these tests are (uplink/downlink):
- (1/3) * 9600bps (i.e. 9600bps/28800bps)

Table 7: Variable bandwidth tests

Bw1 (up/down)	**Bw2 (up/down)**	**Change**	**Tested TCP enhancements**
9600/28800	9600/9600	slow start	Baseline TCP,IW=4, `rwnd`=4KB
9600/28800	9600/9600	steady state	Baseline TCP,IW=4, `rwnd`=4KB
28800/28800	19200/19200	slow start	Baseline TCP,IW=4, `rwnd`=4Kb
28800/28800	19200/19200	slow start	Baseline TCP,IW=4, `rwnd`=4Kb
28800/28800	14400/14400	slow start	Baseline TCP, IW=4, `rwnd`=4KB
28800/28800	14400/14400	steady state	Baseline TCP, IW=4, `rwnd`=4KB
19200/19200	9600/9600	slow start	Baseline TCP, IW=4, `rwnd`=4KB
19200/19200	9600/9600	steady state	Baseline TCP, IW=4, `rwnd`=4KB

- (2/2) * 9600bps (i.e. 19200bps/19200bps)
- (2/2) * 14400bps (i.e. 28800bps/28800bps) and
- 14400bps/38400bps

The different TCP setting for these tests are:
1) Baseline TCP
2) IW = 4
3) SACK-option enabled
4) rwnd = 2KB

Table 8 summarizes the tests. The column *Bandwidth (uplink/downlink)* describes the bandwidth used for a test case. There are 3000 basic tests in 60 different test cases in this test run, as the number of replications is 50.

Discussion It is believed that the large amount of tests provides detailed information about the effects of different link characteristics. The tests cover extensively the modeled environments. Thus, the conclusions of the test results are more accurate. The total number of test cases is 332 , including the tests with different TCP enhancements. Overall, over 10000 basic tests are run, each transferring 100 Kbytes over the emulated link.

Table 8: Tests with packet drops and different bandwidths

Packet drop rate	**Bandwidth (up/down)**	**Tested TCP enhancements**
2%	9600/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
2%	19200/19200	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
2%	28800/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
2%	14400/38400	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
5%	9600/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
5%	19200/19200	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
5%	28800/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
5%	14400/38400	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
10%	9600/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
10%	19200/19200	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
10%	28800/28800	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB
10%	14400/38400	Baseline TCP, IW=4, SACK, `rwnd`=2KB, `rwnd`=4KB

5 Test Results and Analysis

This section introduces the results of the tests that are outlined in Section 4.5. The tests are analyzed, and the reasoning for the problems or benefits of the TCP modifications is given separately for each test scenario. The test results are described by first reporting the behavior and the performance of Baseline TCP , and then the behavior and the performance of the TCP enhancements.

5.1 Ideal link condition

The connection between the two hosts under ideal link conditions is unfortunately a rare case in real world. The link conditions are said to be ideal, if there are no other delays on the link than the transmission delay, propagation delay, and a possible queueing delay, and no segments are lost due to congestion or corruption during the whole 100 Kbytes transfer. The reason for providing such a case is, that it is a good reference when comparing the effects of excess delays and packet drops on other test cases that are executed. Figure 8 shows a trace of transfer under ideal link conditions, taken from the sender side tcpdump. Thus, this graph represents the scenario, as the TCP sender sees it. If a segment is dropped on the link, the TCP sender is not aware of it, until it gets feedback from the TCP receiver. Therefore, all the segments that the TCP sender has sent, are visible in the graph, even if they are dropped on the link. The same thing applies to excess delays. This same format of the graph is used on all the following figures, when showing traces of connections. The X-axis is the time in seconds. In the graphs, the time is started upon the initially sent *data* segments; the segments sent during the connection establishment (i.e the three-way handshake) are not plotted nor the time spent in this procedure taken into account, when setting the initial time axis. The Y-axis represents the sequence numbers relative to the initial value. Each sequence number equals one byte. The line labeled "data sent" includes markers at the highest sequence number of a data segment. Each marker represents a data segment, that is sent. The line "ack rcvd" (acknowledgment received) has markers on the next expected octet, which are equal to the corresponding `ACK`, indicating that all segments, up to that point, are correctly received by the TCP receiver. "Win" is the line that represents the highest sequence number accepted by the TCP receiver (i.e receiver's advertised window, `rwnd`). This line is not present in all of the graphs, since it does not always have an effect on the behavior of the TCP sender.

Using the Baseline TCP, the connection time from the first `SYN`-segment to the last `FIN`-segment received is 102.05 seconds in average. The total number of data segments is 400, while there are no retransmissions. The RTTs for the first and last data segments are approximately 0.7 and 30 seconds, respectively. The RTT gets inflated due to a large `cwnd`. The `cwnd` grows to 128 segments, and is limited by the `rwnd` after 32 Kbytes have been acknowledged by the receiver. The growth of the RTT due to a large `cwnd` may lead to problems, which are discussed later. The total throughput is 1003 bytes/second. However, if the initial window size is set to four segments, the connection time is 101.85 seconds (i.e. 0.2 seconds shorter than with the Baseline TCP), and the throughput is 1005 bytes/second. The full capacity of the link is in use from the very beginning of the connection, as the delay-bandwidth product is 840 bytes[6]. Thus, an IW of two segments cannot fill the link, but four segments can. This scenario is further discussed in the following chapters, as it is a factor that affects the performance on several occasions. Table 13 in Appendix A summarizes the connections under ideal link conditions.

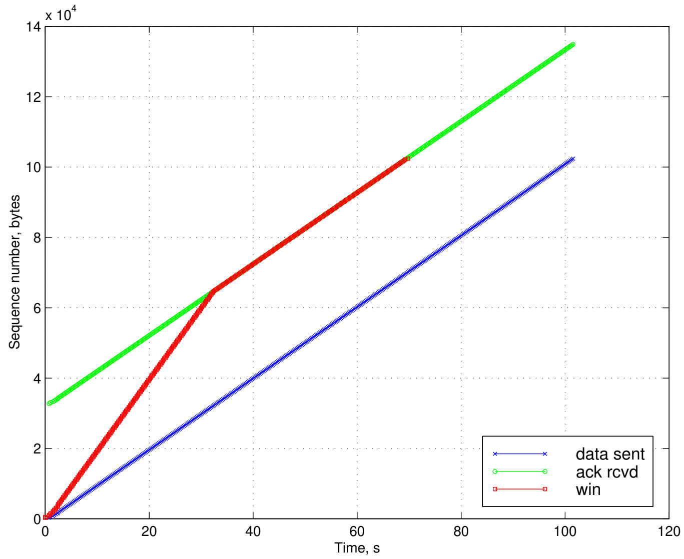


Figure 8: Optimal transfer using the Baseline TCP

5.2 Effect of packet drop

The results of packet drop tests are given separately for single drops and random drops. A summary of all packet drop tests is given at the end of this section.

[6]A data packet consists of 256 bytes of TCP payload, 40 bytes of TCP and IP headers, and 8 bytes of PPP overhead = 304 bytes. The sum of propagation delays (2*200ms) and transmission delays for a 304 byte data segment and a 48 byte `ACK` are approximately 700ms. Thus, the delay-bandwidth product is $0.7s * 1200bytes/s \approx 840$ bytes

Single packet drops

Baseline TCP In general, the recovery from single packet drops is nearly optimal when Baseline TCP is used. The NewReno algorithm is triggered after receiving three duplicate acknowledgments and the lost segment is then retransmitted. After receiving the cumulative `ACK` that acknowledges all outstanding packets when the third dupack was received, the algorithm is exited and the connection continues using the congestion avoidance algorithm. The only limiting factor on these cases is the reduction of the `cwnd` and the entering of congestion avoidance instead of continuing the slow start. However, these actions do not diminish the throughput in most of the test cases, since the link is usually filled with the maximum number of segments even after the packet loss. Under those circumstances, the median connection time increases only less than 0.3 seconds, when comparing to the ideal case, where no segments were dropped. The connection time increases due to the transmission delay of the retransmitted segment. Figure 9 shows the trace of the connection where the 40th data segment was dropped.

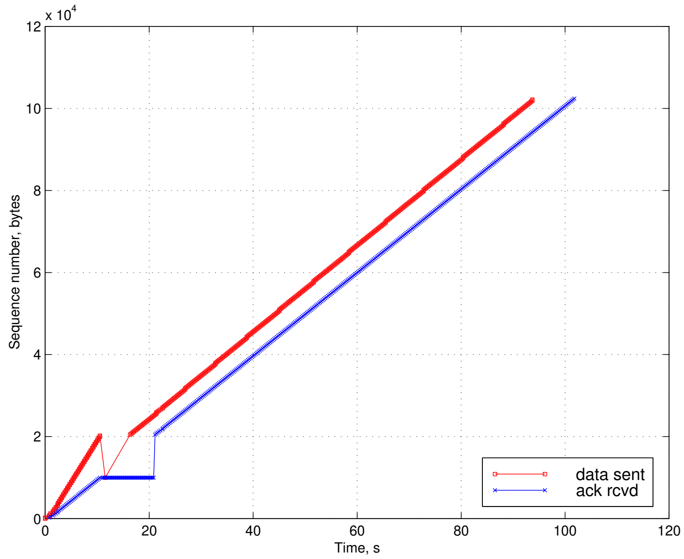


Figure 9: 40th packet dropped

There are two distinct cases, where the recovery is suboptimal. First, if there are less than four data packets in the network, a fast retransmit is not possible due to the limited number of possible dupacks in case of a packet drop; the TCP receiver needs to receive three out-of-order segments to send three dupacks that trigger a fast retransmit at the TCP sender. The TCP sender has to recover from a missing segment by waiting for the retransmission timer to expire. Second, if the `cwnd` has grown well beyond the delay-bandwidth product, an RTO can occur for the retransmitted packet before the `ACK` arrives. We explain these two cases in more detail in the following chapters.

The reasoning for another minor degradation in the performance is that the link capacity is not always in full use. The delay-bandwidth product of the link is 840 bytes, as previously explained in Section 5.1. If the pipe is full, a new `ACK` is sent every 304bytes/1200bytes/sec = 0.25 second, as it is the transmission delay of a data segment[7]. Since the transmission of an `ACK` segment takes only 48bytes/1200bytes/sec + 0.2 sec = 0.24 seconds, the `ACK` is transferred to the TCP sender before a new `ACK` has been created. This means that only one `ACK` segment is on the link at any time. If the pipe capacity is 840 bytes, three data segments are needed to fill the pipe[8]. Thus, the pipe is not fully utilized, if the `cwnd` has not grown into six segments before the three dupacks are received, because the `cwnd` is halved upon a fast retransmit; a `cwnd` of six segments leads to a `cwnd` of three segments, which keeps the full link capacity in use. We can see this degradation happen, when the third data segment is dropped. This leads to a 0.65 second longer total connection time.

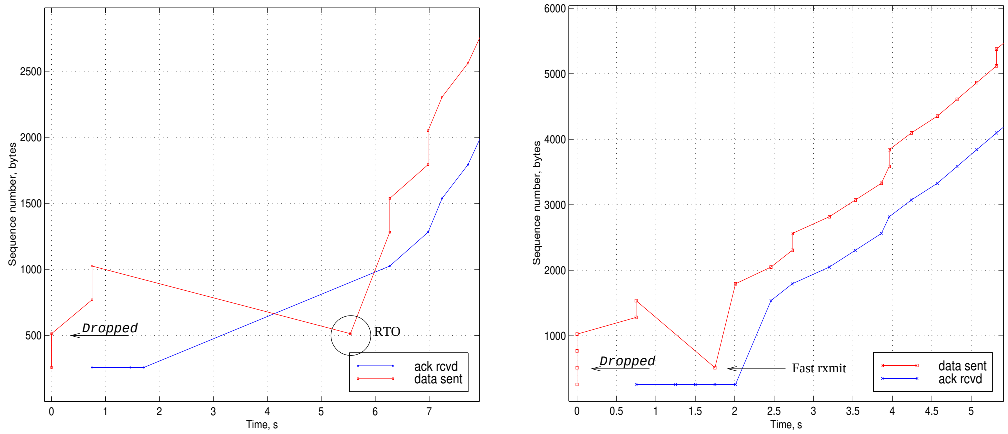
If the dropped segment is the first, second, or the last data segment of the connection, a fast retransmit is not possible, as the number of packets in the network is less than four[9]. To recover for a single packet drop in these three cases, the TCP sender has to wait until the retransmission timer expires. Figure 10(a) shows a microscopic view from the beginning of a connection where the second data packet was dropped. If the initial window were one segment, the TCP sender could recover only by going into an RTO for the third dropped data segment, too. At the end of the transfer, the RTT and RTO values get inflated. The retransmission timer expires 40 seconds after the initial transfer of the last data segment. This leads to a loss of eight seconds in overall connection time, as the RTT for the segment would have been 32 seconds without the loss.

If the `cwnd` is relatively big, 60 segments or more, the `ACK` that acknowledges the retransmitted packet and all outstanding data at the point when the three dupacks were received by the TCP sender, does not reach the TCP sender before its retransmission timer expires. The out-of-order segments at the receiving end do not allow the receiver to send a new `ACK` to the sender, but only dupacks. When the TCP sender does not get a new acknowledgment in time, the segment is retransmitted. Figure 11 shows this scenario, that invokes multiple unnecessary retransmissions, and increases the connection time by 14% (14.4 seconds). The segment that is retransmitted due to RTO, is highlighted in the figure (number 1). Because the retransmission timer expires, *slow start* is invoked and continued after receiving the cumulative `ACK` (number 2). This `ACK` does not acknowledge the segments that were transmitted during the recovery period (number 3), but the segments that

[7]This does not hold, if delayed acknowledgments are in use. In that case only every second segment would trigger a new `ACK`.

[8]$840bytes/304bytes \approx 2.8segments$

[9]This also applies to the third last and second last segments, but these cases are not included in the tests.



(a) Baseline TCP

(b) Initial window of 4 segments

Figure 10: The recovery when the second data segment is dropped.

were sent before the three dupacks were received. Therefore, the *slow start* begins from the next expected octet indicated by the `ACK` (number 2) and the segments (number 3) are retransmitted. The retransmitted segments cause dupacks, because the data receiver gets them twice. The third dupack triggers NewReno fast retransmit which causes an unnecessary retransmission (number 4). This retransmission could have been avoided, if so-called *Careful* version of the NewReno bugfix [FH99] was implemented[10]. The *Careful* variant does not enter fast retransmit/fast recovery if the last acknowledged segment is not higher than the last segment sent before the RTO occurred (the highest segment indicated by number 3). Due to this retransmission, TCP sender enters fast recovery and the next `ACK`s are interpreted as *partial acks*, and more segments are retransmitted. When this scenario ends, the connection continues using *congestion avoidance*.

If the `cwnd` reaches the `rwnd` before the packet is dropped, the scenario changes. Figure 12 shows this case. After receiving the third dupack, the dropped packet is retransmitted. By that point, the `cwnd` has already reached the `rwnd`, and thus, new packets can not be sent during the recovery period (number 1 in the figure). The retransmission timer expires for the retransmitted packet as in the previous scenario. Because the data sender could not send any new data to the network, the next incoming `ACK` acknowledges all segments the sender had sent (number 2). The *slow start* is invoked and only new data is sent to the TCP receiver. The later part of the connection goes without any problems with reduced `ssthresh`. *Congestion avoidance* is entered right before all the data was sent to the network (number 3).

[10] Baseline TCP implements the *Less Careful* version of the NewReno bugfix. See Appendix C for more detailed information.

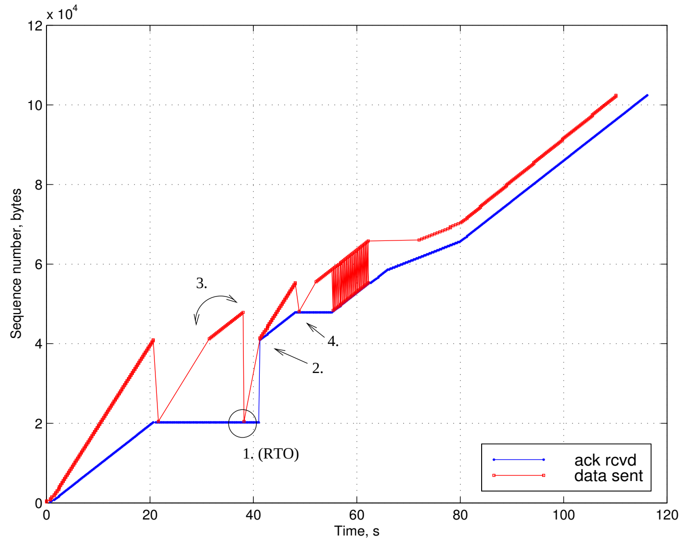


Figure 11: 80th packet dropped. The large `cwnd` causes an RTO for the retransmitted packet.

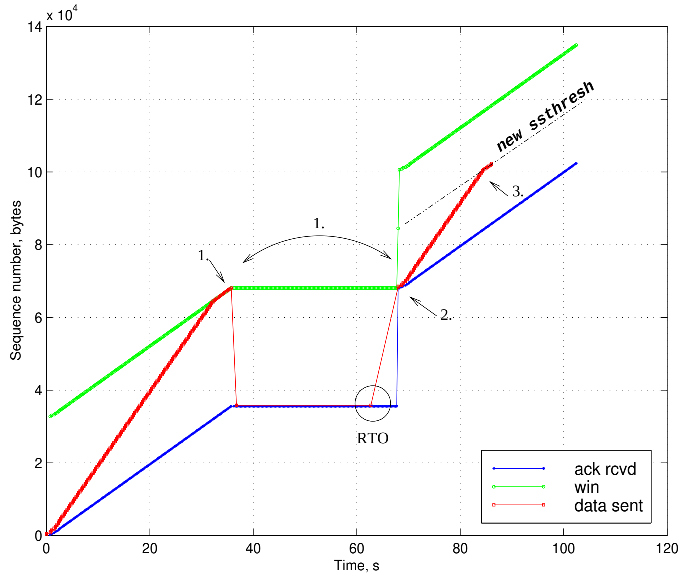


Figure 12: 140th packet dropped.

If the TCP sender has already transmitted all the transferred data upon the third dupack, the missing segment is the only segment that is retransmitted. Thus, the recovery is as optimal as it can be.

Larger initial window Excluding the problematic scenarios, the pipe is kept full all the time due to the larger initial window (IW), and thus, the median connection times are approximately 0.2 seconds shorter than in the Baseline tests. Because the four initially sent segments fill the pipe, the link capacity is in full use from the very beginning of the connection, as the four initially sent segments fill the pipe.

When using an IW of four segments, the RTO can be avoided for the drop of the first or second data packet unlike for the Baseline TCP case that was explained earlier. Figure 10(b) shows the scenario, when the second data segment was dropped. The TCP sender can recover from the drop by triggering a fast retransmit because there are enough outstanding segments in the network to cause three dupacks.

The use of an IW of four segments does not give other benefits. The problem with too large a `cwnd` (explained earlier) still exists, leading to numerous unnecessary retransmissions. As the size of the `cwnd` is initially larger, the problem occurs a little earlier than with the Baseline TCP. However, the reasoning behind the problem is exactly the same.

Limited receiver window Limiting the receiver's advertised window to 2 Kbytes leads to a maximum `cwnd` of eight segments[11]. The limited receiver window does not benefit the very beginning of the connection in any way, and thus, an RTO occurs if the first or second data segment is dropped. The results are the same as with Baseline TCP. Also, the RTO can not be avoided when the last data segment is dropped. However, the retransmission timeout is much shorter than in the Baseline TCP tests, because the RTT calculations have not been inflated by an arbitrary big `cwnd`. The RTO causes only an extra delay of 0.6 seconds in the overall connection time, while with the Baseline TCP, the extra delay was 8.8 seconds.

Figure 13 shows a common scenario when using a limited receiver window. The `cwnd` reaches the `rwnd` after the first six segments are acknowledged by the receiver (number 1 in the figure). After that point, the transmissions of new segments during the fast recovery phase is impossible, since the `rwnd` does not allow any new segments to be sent before the retransmitted segment is acknowledged (number 2). This causes a slight deterioration of the performance, because the link capacity is not in full use for one RTT after exiting the fast recovery. The new value of the `cwnd` is derived from the number of packets in flight. Because there are currently no outstanding segments on the link, the new value of `cwnd` is set to two segments[12] (number 3). Because the link's capacity is not in full use if there are less than three outstanding segments, the `cwnd` of two segments leads to a minor loss of the performance. The median throughput is 994 bytes/second in a general situation, as

[11] When using a MSS of 256 bytes, 8*(MSS) = 2 Kbytes.

[12] The exact value for the `cwnd` is calculated from the equation:
`cwnd` = $max(min(flightsize+1, ssthresh), 2)$. Since the flightsize+1=1, the maximum of 0 and 2 is selected to be the new value of the `cwnd`, i.e `cwnd`=2. For a more detailed description about calculating the values for the `cwnd` and ssthresh, please refer to the Appendices B.2.4, B.2.5 and C.1.1.

it is 1001 bytes/second when using Baseline TCP. The difference is less than 0.7 seconds in total connection time.

By limiting the `rwnd` to 2 Kbytes, the RTO due to too large a `cwnd` can be avoided, because `rwnd` limits the growth of the `cwnd` to eight segments. Therefore, the throughput while using the limited receiver window stays steady in most parts of the connection, excluding the RTO scenarios at the beginning and the end of the connection. However, if the last data segment is dropped, the RTO occurs sooner than with the Baseline TCP. The RTO calculations are not inflated due to a large `cwnd`, as the `rwnd` gives an upper limit for it.

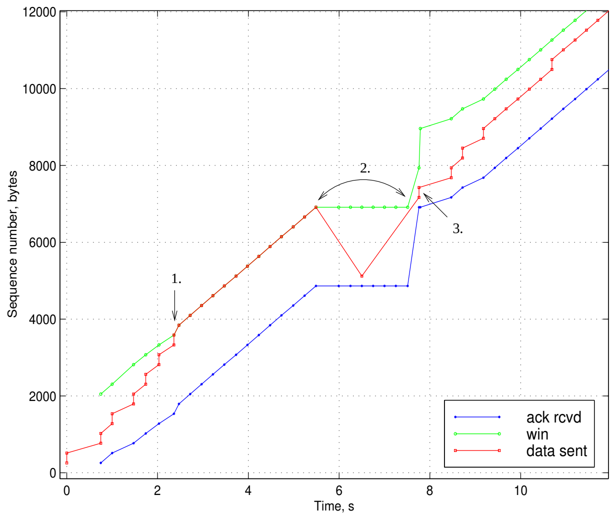
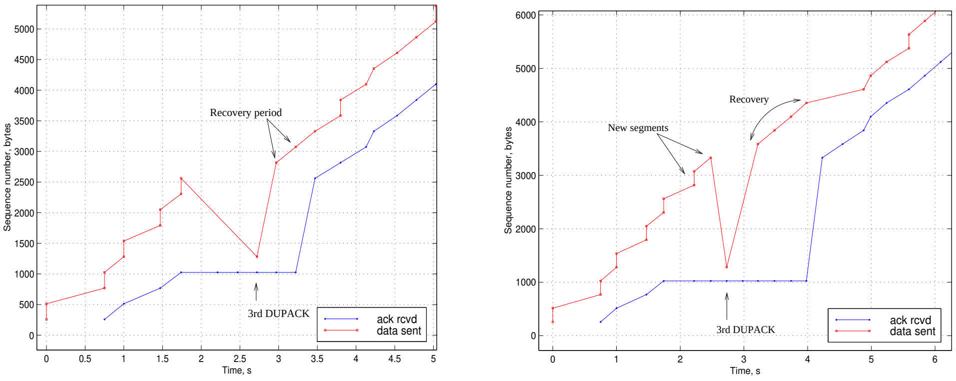


Figure 13: A limited receiver window of 2 Kbytes. The TCP sender does not use the full capacity of the link, as the number of outstanding segments after the fast recovery is only two.

SACK The SACK option was optimized to recover from multiple packet losses in a single window [FH99]. The third dupack triggers a fast retransmit, for both Baseline TCP and SACK. Therefore, the SACK results are almost identical to Baseline TCP, even in the scenarios, where multiple segments were unnecessarily retransmitted. However, there are minor differences in the recovery. The SACK TCP sender is able the transmit new data upon the first two dupacks. This behavior is due to FACK algorithm included in SACK. The dupacks provide information about the received segments that are *not* acknowledged in the regular `ACK`-field. The extra information is carried in the SACK blocks of an `ACK` segment. The TCP sender may transmit new segments to the network, as it knows a segment has reached the receiver. Figure 14 compares the recovery with the Baseline TCP and SACK, when the fifth data segment is dropped. The segments injected to the network upon the two dupacks are shown in Figure 14(b). As a result, the new segments trigger more dupacks as the Baseline TCP, and more segments are sent during the recovery period (see Figure 14(b)). In this scenario, the performance implications are minimal, as the capacity of the link is in full use in both of the cases. However, SACK prevents the TCP sender to fall to an RTO if the first data segment is dropped, as the number of possible dupacks increase along the new segments injected. This behavior imitates the *Limited Transmit* TCP enhancement [ABF01].



(a) Baseline TCP

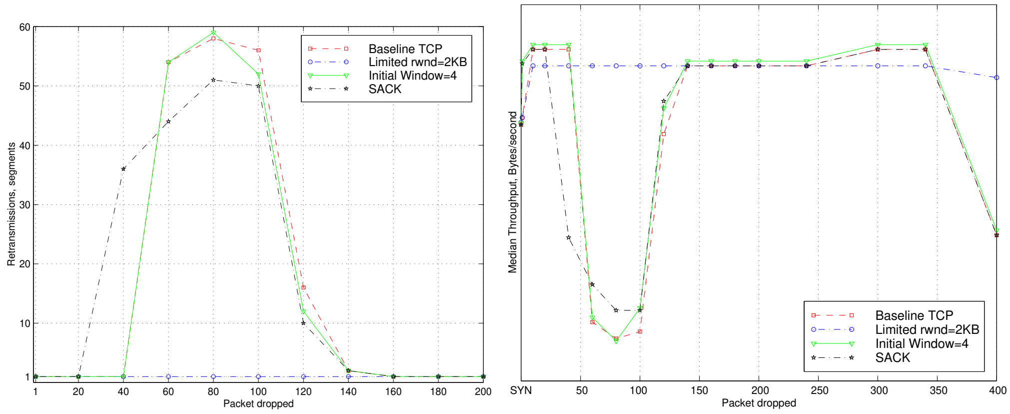
(b) SACK

Figure 14: The recovery of Baseline TCP adn SACK TCP when the fifth data segment is dropped.

The statistics for the single packet losses can be seen in Tables 14-16 in Appendix A.

Discussion Considering that only one data segment was dropped during the connection, one retransmission should have been invoked. Baseline TCP, the larger initial window and SACK TCP retransmitted more than one segment on some occasions. With the limited `rwnd` of 2 Kbytes TCP retransmitted one segment in all the test cases. Figure 15(a) summarizes all the retransmission invoked with the tested modifications. Notice, that packets up to the 200th are only plotted in the graph. The other packet drop tests are not shown, because they all resulted only one retransmission. The differences between SACK and the Baseline TCP are due to implementation issues of the SACK algorithm is Linux. The behavior is the same, but the number of outstaning segments in the network is different, as SACK lowers the `ssthresh` twice if an RTO occurs during the fast recovery period, and Baseline TCP reduces it only once. Figure 15(b) indicates the median throughputs of the different TCP enhancements.

The problems due to a large `cwnd` do not limit to an unnecessary RTO. If the `cwnd` has grown well beyond the delay-bandwidth product, the RTT estimations, and especially the



(a) The number of retransmissions

(b) The median throughputs of the TCP enhancements

Figure 15: The performance of Baseline TCP, `rwnd` of 2 KB, IW of four segments and SACK TCP when a single packet is dropped.

RTO calculations, get heavily inflated. If the TCP sender has to rely on the retransmission timer to be able to retransmit a missing segment at the end of the connection for more than once, the time needed becomes excessively long. This scenario is further discussed in the chapters that follow.

Random packet drops

Packet drop rates of 2%, 5% and 10% were studied. They represent good, mediocre and bad link conditions, respectively. The focus is on the drops that occur only in the downlink direction, while the `ACK` path (i.e uplink) contained no errors. The tests were executed also over an erroneous link where both up and downlink contained packet drops. In addition to the Baseline TCP, SACK option, limited `rwnd` of 2 Kbytes and 4 Kbytes, and IW of four segments were tested. The tests were repeated using a MSS of 512 bytes.

Baseline TCP It was shown in the previous chapter, that NewReno recovers well from a single dropped segment. If there are multiple segments that are dropped from a single window, NewReno is able to retransmit one segment per RTT [FF96].

If the `cwnd` is reduced to less than four segments, the recovery is not possible without going into RTO. Each drop reduces the sender data transmission rate by 50% by halving the `cwnd` upon the third dupack. A general rule for the maximum number of segments NewReno is able to recover from, is difficult to provide. The effect of a dropped segment depends on the current situation of the TCP sender. For example, in a common case where two segments are dropped from a window, the transmission rate is not halved twice, because NewReno can recover from the situation by sending one retransmission per RTT[13]. If the segments are dropped in sequential windows, both of the drops reduce the `cwnd`. The more there are dropped segments on the link, the more likely they will eventually reduce the `cwnd` to less than four segments. In those cases the TCP sender has to wait for the RTO to expire to send new segments to the network.

In the single drop cases, a fast recovery was not possible, if the dropped data segments include the first, second, or last three data segments. While there may be multiple segments dropped, this rule is extended to include all the possible scenarios, where the number of possible dupacks is less than three.

There is, however, a special scenario that NewReno is not able to recover from, even the segments are dropped from the same window. If a retransmitted segment is dropped, the TCP sender has no possibilities but to wait for the RTO to expire. After the RTO, a slow start has to be invoked, according to the TCP congestion control specifications [APS99]. If the TCP sender has to rely on the retransmission timer later during the same recovery phase, the time needed for the RTO is twice as much as for the previous RTO. This is due to the fact that the retransmission timer is backed off by multiplying the current value by two upon every RTO. The backed off value is reset only after the reception of an `ACK` that acknowledges new data (i.e. the acknowledged sequence number is higher than the sequence number of the last segment sent before entering the recovery period). The following example shows that even a two segment drop may collapse the performance.

If the last data segment of a bulk data transfer is dropped, the TCP sender has to wait for an RTO before it can retransmit the segment, because there are no outstanding segments that could trigger three dupacks. After RTO, the sender retransmits the segment, which it is lost again. After waiting for another RTO, the last segment is delivered successfully. These two dropped segments caused two RTOs. If we think of the single packet drop test introduced earlier, RTT for the last segment was 32 seconds, and the corresponding RTO value, calculated from the RTT, was 40 seconds. Due to *exponential backoff*, the second timeout occurs only after 80 seconds of idle time. As the connection time is approximately 110 seconds with one loss, these two well selected losses can increase the connection time by 73%, from 110 seconds to 190 seconds.

We have outlined the possible problems that may occur during a connection with an error-prone link, when using the NewReno TCP modification. As all the other TCP

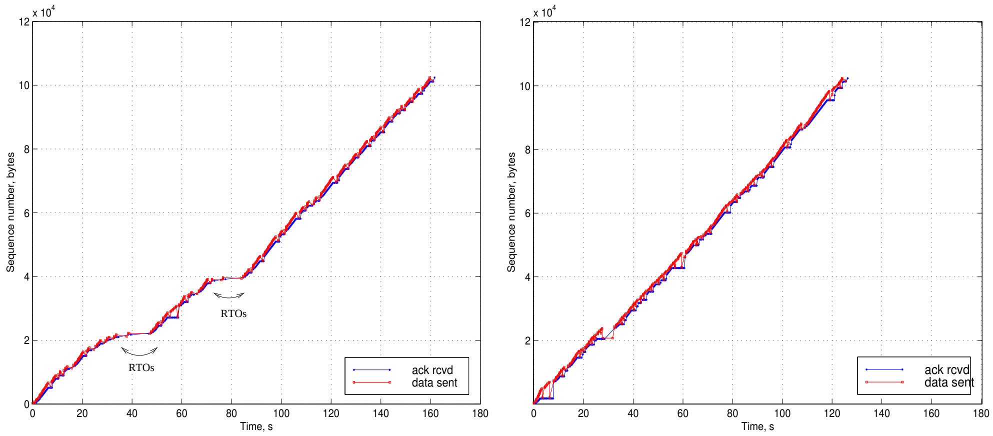
[13] If the second dropped segment was the one that was retransmitted, this does not apply. See next chapter for details.

modifications we tested, except SACK, make use of NewReno, the similar problems are observed with those modifications, as well. As there still are differences between the performance of those modifications, they are further analyzed in later chapters.

Under good link conditions (i.e the packet drop rate was 2%), the recovery was good. As the number of dropped segments is small, a situation where an RTO is needed to recover from a segment loss is unlikely to happen, but possible. The median connection time of the transfers was 106.04 seconds; only four seconds (3.9%) longer than under ideal link conditions. The throughput was 965.5 bytes/second.

If the packet drop rate was 5%, the RTOs occurred more often. The `cwnd` was reduced to a small value, and an RTO was needed to recover from a lost segment. The median connection time was increaced by 12.7% (13.51 seconds) and 17.1% (17.5 seconds) when compared to the tests over good and ideal conditions, respectively.

Under bad link conditions with 10% packet drop rate, the median connection time increased by 40.3% (from 119.55 seconds to 167.73 seconds) compared to the connection time under mediocre link conditions. The large number of dropped segments often caused the TCP sender to rely on the retransmission timer to be able to retransmit a segment. The size of the `cwnd` was relatively small throughout the connection, as the dropped segments required the halving of the transmission rate. In the absence of congestion, this reduction caused a highly suboptimal performance. The TCP sender was constantly doing recovery procedures, and hardly got to send new segments to the network. Even several consecutive RTOs are likely to happen, as the retransmissions might get dropped, as well.



(a) Baseline TCP (b) SACK

Figure 16: Using Baseline TCP, a packet drop rate of 10% leads to multiple RTOs. The SACK option recovers without any RTOs.

SACK If there are multiple packet losses from a single window, the SACK option helps the TCP sender recover in a aggressive way. Due to the *cumulative acknowledgments*, without the SACK option, the TCP receiver cannot acknowledge all the segments that are received, if some segments, up to the highest sequence number, are received. The additional information given in the SACK blocks help the TCP sender to deduce the correctly received segments, which still have not been acknowledged by a regular `ACK`. Figure 17 shows a microscopic view of a trace under bad link conditions. Notice, that the sender is able to immediately send new segments to the network after a fast retransmit, as the SACK blocks carry information about missing segments. By using SACK, multiple retransmission during one RTT are possible. This saves time, and the probability of several consecutive RTOs is highly unlikely, as the number of outstanding segments is usually high. The Baseline TCP was able to retransmit only one segment during one RTT. As the packet drop rate augments, the losses of the retrasmitted segments become more likely.

As the Linux SACK implementation uses *forward acknowledgments*, the sender transmits more new segments than the correct value of the `cwnd` would imply. This behavior is considered to be due to the Linux SACK implementation, and it should not be applied to the SACK option in general. However, the benefits of SACK are still evident, as it allows multiple retransmission during one RTT.

Figures 16(a) and 16(b) show the differences between the behavior of the TCP sender, while (a) uses Baseline TCP, and (b) uses SACK. In 16(a), several consecutive RTOs occurred due to small `cwnd` and lost retransmissions.

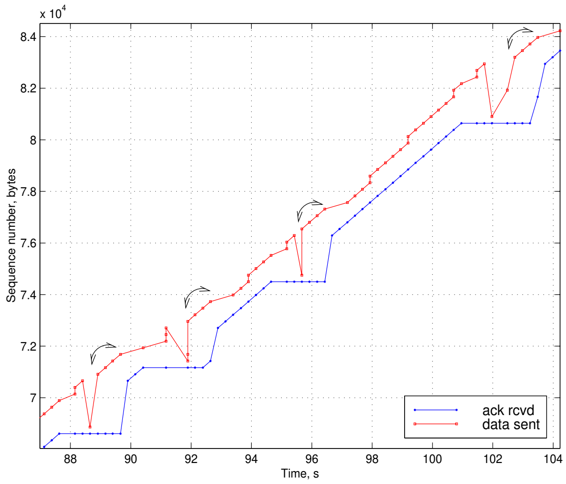


Figure 17: SACK is able to retransmit aggressively and to send new segments immediately after a retransmission.

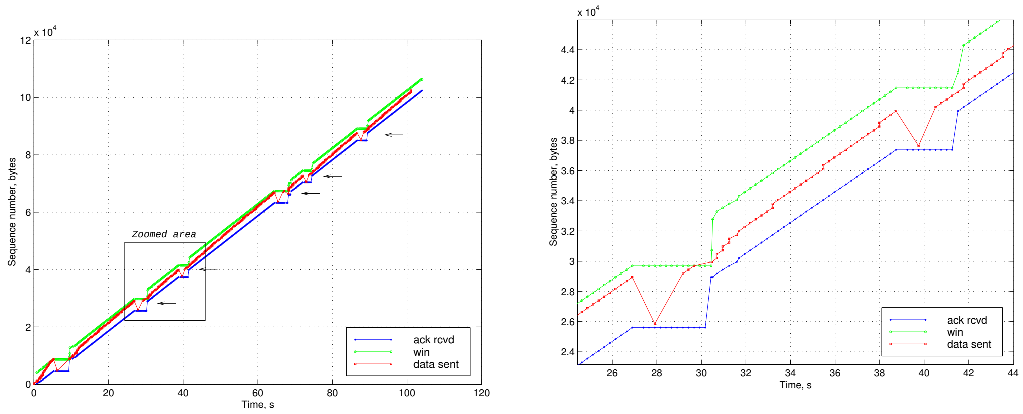
Limited receiver window Receiver windows of 2 Kbytes (eight segments) and four Kbytes (16 segments) were tested. When using a `rwnd` of 2 Kbytes, under good link conditions, the minor degradation of the performance resulted in the underutilization of the link. This is the same scenario that was outlined in single packet drop tests; usually, the `cwnd` was being limited by the `rwnd`, and thus, when recovering from a missing segment, the number of outstanding segments was only two, which does not take the full capacity of the link in use. Each lost segment lowers the throughput, and, as there are multiple dropped segments, this causes the median connection time to grow by 3.9% (4.17 seconds) compared to the Baseline TCP.

This scenario is not as common if the `rwnd` is set to 4 Kbytes. The `cwnd` is not usually limited by the `rwnd`, since each dropped segment halves the `cwnd`. It is unlikely that it will grow again to 16 segments, which is the limit given by the TCP receiver. Because the TCP sender is able to transmit new segments during the recovery period, the `cwnd` is typically set to three segments, or more, which is enough to fill the link capacity. Figure 18(a) shows a trace, where only at the beginning of the connection the `cwnd` has reached the `rwnd` and, thus, is not using the available capacity of the link. The following fast recoveries are exited by using a larger `cwnd` as new segments were transmitted during the recovery period, and the TCP sender makes use of the full link capacity. Figure 18(b) is a microscopic view from the middle of the connection. It is noticeable that the number of outstanding segments, after the fast recovery, is more than two.

The `rwnd` of 2 Kbytes is too low, if the number of dropped segments is high during a transfer. The initially small size of the `cwnd` leads eventually to an RTO, as the lost segments reduce the `cwnd`. By using a `rwnd` of 4 Kbytes, this problem is avoided. The performance achieved by limiting the receiver's window to 4 Kbytes, is approximately the same as of the Baseline TCP's.

Larger initial window If the `cwnd` grows faster to safe value (i.e more than three segments), a fast retransmit is possible without relying on the time-consuming retransmission timer. The tests show, however, that the benefits are minimal. Under good link conditions, the larger initial window makes use of the full link capacity from the very beginning. This seems to be the only benefit; the segments are dropped relatively seldom, so the `cwnd` is not subject to be reduced under four segments. The difference in median connection time was less than one second in favor of the larger initial window, when compared to the Baseline TCP.

The tests, which were run over a link that dropped 5% of the data segments, did not show any additional improvement in the recovery process. In fact, the results were a little worse



(a) The full connection

(b) A zoomed view

Figure 18: The `cwnd` reaches the `rwnd` only at the beginning of the connection. On later drops, the TCP sender can send new segments to the network during a fast recovery.

than the ones of the Baseline TCP. Even if this may be due to insufficient sampling size, 50 replications, it is believed, that under such link conditions, a larger initial window of four segments does not give remarkable benefits.

If the link is heavily unreliable, the performance is again better compared to the Baseline TCP. An RTO at the beginning of a transfer costs three seconds, as it is the initial value set for the retransmission timer to expire. Therefore, as there are initially four segments on the link, the RTO can be avoided even if the first data segment is lost. The TCP receiver triggers three dupacks for the `SYN` segment, and the first data segment is then fast retransmitted. Hence, the benefits of a larger initial window are more evident at the beginning of the connection. As the link conditions become worse, the possibility of a dropped packet from the initial window is more likely than over mediocre link conditions. It is interpreted to be the reason for the performance differences over mediocre and bad link conditions.

Larger MSS The tests were repeated using a MSS of 512 bytes. The results were similar, and no other problems were found for any TCP modification. It should be noticed that the connection time for a transfer is shorter than for the Baseline tests that were introduced at the beginning of this section. The reason for better throughput is that the header overhead is smaller. For every segment, TCP, IP and PPP headers are to be added before the transfer. Therefore, as there is a need to transfer only 200 segments, instead of 400, only half the amount of headers need to be transferred. The difference in pure bytes to be transferred is then $200 * (20 + 20 + 8) = 9600$ bytes. The time it takes to transfer

9600 bytes over a 9600bits/second link is 8 seconds. However, the use of larger MSS due to smaller header overhead is not desirable, as the response time gets higher, and it makes it harder for the end-user to work interactively. In addition, upon a dropped segment, the number of bytes to be retransmitted, increases.

Discussion The analysis of these tests were based on the test cases, where only the data segments were dropped. A link, where the `ACK`-path also contains packet drops, suffers from the same situations that were described earlier in this section. In addition to these, three new observations were made. First, a loss of an `ACK` segment may lead to an RTO at the TCP sender, if the number of received dupacks is less than three. The larger size of the `cwnd` does not guarantee that three dupacks will eventually arrive, in case of a packet drop. Second, retransmission timeouts are more likely, as the acknowledgment of a segment may be lost. Third, if several consecutive `ACK`s are dropped on the link, the TCP sender eventually receives an `ACK` that acknowledges multiple segments. This leads to bursts of data to be transmitted, as the `cwnd` is suddenly decreased by multiple segments. Such bursts should be avoided, because they may cause an overflow in an intermediate router. In the environment under study, however, there was no router that suffered from buffer exhaustion. That environment is studied in [Gur00] and [Sar01], for example.

Even if a loss of an `ACK` segment may lead to new problems, the consequences are not usually very severe; as the acknowledgments are cumulative, every `ACK` acknowledges the segments that the previous `ACK` acknowledged. Obviously, this is not the case for data segments. Therefore, data segments that are dropped are much more likely to produce performance problems, since every data segment has to be delivered successfully to the TCP receiver. The results of the tests, where only the data segments were dropped and where also `ACK` segments may be lost, corroborated this deduction. The tests gave similar median connection times over good and mediocre link conditions. However, if the link conditions were bad (packet drop rate of 10%), the dropped `ACK`s caused more RTOs, as the size of the `cwnd` was small due to multiple dropped data segments.

The TCP sender has to receive three dupacks to invoke a fast retransmit. If the `cwnd` is less than four segments upon a packet loss, the fast retransmit is not possible. A new TCP enhancement, called *Limited Transmit* [ABF01], is likely to help the TCP sender in situations, where the `cwnd` is less than four segments upon a packet loss, or when a large number of segments is lost in a single transmission window. The enhancement lets the TCP sender transmit a new segment upon the arrival of both of the first two dupacks. Hence, a fast retransmit is possible even if the `cwnd` is as low as two segments.

It has been stated that the `rwnd` should not be too large comparing to the delay-bandwidth product of the link [DMKM00]. The tests showed that in absence of congestion, too small a `rwnd` reduces the throughput significantly, even the `rwnd` was over twice as large as the capacity of the link, when tested over mediocre link conditions. If there would be an intermediate router, the performance of a limited `rwnd` might be better over a error-prone link.

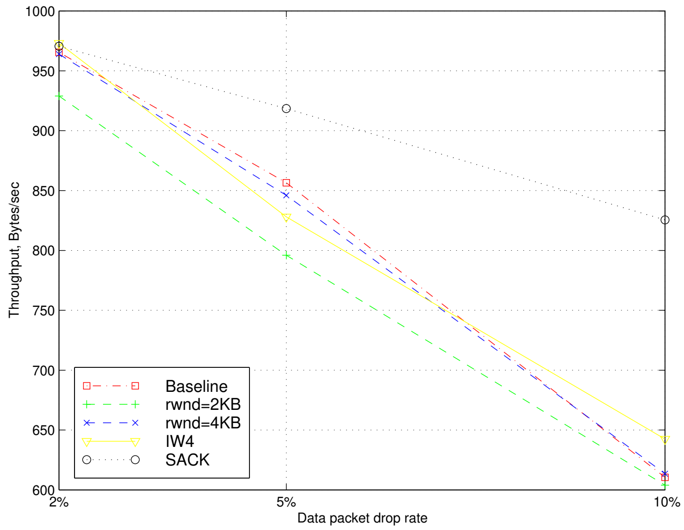


Figure 19: The achieved throughput for each tested TCP modification

Table 18 summarizes the test results for the random drops tests.

Summary of the packet drop tests

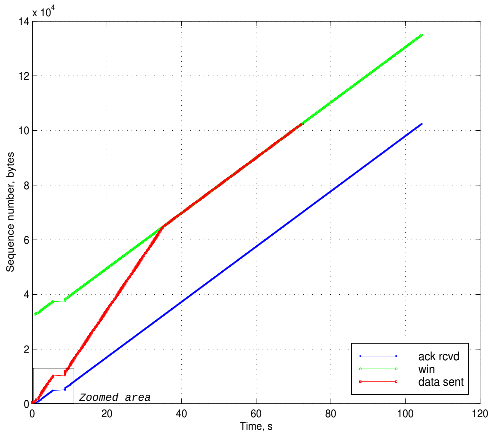
In general, Baseline TCP is able to recover well from a single packet drop. Too large a `cwnd` causes numerous unnecessary retransmissions. Limiting the receiver window may have a positive effect on the performance as too large a `cwnd` may cause unnecessary retransmissions. However, it does not ameliorate the situation when the link condition becomes worse. The SACK option is the best TCP enhancement to use in such an environment. When reducing the receiver window, it must not be too small; a window of 16 packets (4 Kbytes when using a MSS of 256) does not hinder the performance compared to the Baseline TCP in the studied environment. The overall benefits of a larger initial window are still debatable, but the results indicate that it ameliorates the performance at least in situations, where a packet is dropped from the initial window. The *Careful* variant of the NewReno bugfix, and *Limited Transfer* could be the TCP enhancements that would be interesting to study in the future.

5.3 Effect of delays

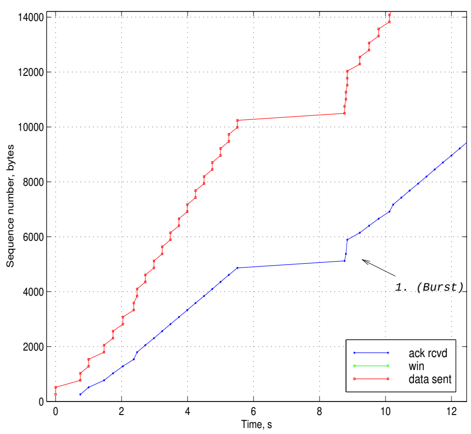
This section provides the analysis for the delay tests. A single segment was delayed. The occurrence of the delay was altered so that the both phases of a connection, slow start and steady state, were investigated. Three different delay lengths were tested with each delayed packet. One delay length was selected to invoke a retransmission timeout for that packet, one for invoking two consecutive retransmissions and one that does not trigger the retransmission timeout. Baseline TCP and a limited receiver window of 4 Kbytes were tested.

Baseline TCP If the delay does not cause an RTO, no segments are retransmitted. Figure 20(a) shows a connection where the 20th data segment was delayed for 3000 milliseconds. The excess delay causes an idle period, but does not affect the performance in any other way, i.e. no segments are retransmitted, and the connection continues by using slow start. Figure 20(b) is a microscopic view from the beginning of the connection. A small burst of `ACK`s follow the delay (number 1. in the figure). This is due to the link buffers. The emulated link ARQ uses these buffers to store data until they are acknowledged. Therefore, the following segments were already received by the TCP receiver, but not delivered to the upper (TCP/IP) layer to provide the segments in order. Notice, that the link is configured to avoid re-ordering, as described in Section 4.3. When the delayed segment is correctly received, all the segments in the link buffers are delivered at the same time. The link buffer size is 1220 bytes, so it can hold three segments, in addition to the locally retransmitted one. The small burst is the only consequence that the excess delay affects.

The basic problem of TCP is that it cannot distinguish between a spurious timeout due to an excess delay, and a retransmission timeout due to a lost segment. If an RTO occurs due to an excess delay, the recovery is always the same: the whole window of data is retransmitted unnecessarily. Figure 21 shows the recovery of the TCP sender when the 20th data was excessively delayed for 3300 milliseconds. The RTO expires right before the corresponding `ACK` would have arrived , and slow start is invoked (number 1 in the figure). A burst of `ACK`s follows (number 2). These are due to the link buffers, explained earlier. Since the TCP sender suffers from *retransmission ambiguity*, the `ACK`s are interpreted as acknowledgments for the retransmissions. As a consequence, the whole window of data is retransmitted (number 3). While the TCP receiver gets the unnecessarily retransmitted segments, it invokes dupacks, one for each segment (number 4). Due to these, the TCP sender invokes a false fast retransmit, since the Baseline TCP implements the *Less Careful* variant of the NewReno bugfix. The *Careful* variant would not have done a false fast



(a) Baseline TCP, full connection



(b) Zoomed view

Figure 20: The 20th data segment is delayed for 3000 milliseconds.

retransmit, because the dupacks do not cover *more* than the highest sequence number sent before the RTO (number 5). The *Less Careful* variant, which was used, checks only that the dupacks *cover* the highest sequence number. As the TCP receiver gets the *new* segments that were transmitted after the RTO (number 6), it acknowledges them normally (number 7). Because the TCP sender is still doing a (false) fast recovery, those ACKs are interpreted as *partial acks*. Therefore, numerous unnecessary retransmissions are sent. Each of the retransmissions trigger one dupack later in the connection (number 8), but they do not cause any further unnecessary retransmissions. While the TCP sender has received an ACK for each of the segments transmitted before the dupacks, the connection continues normally by using congestion avoidance and a reduced cwnd. Since the full window is always retransmitted, the larger the cwnd is, the more unnecessary retransmissions are sent.

If the excess delay causes two consecutive RTOs for a segment, the same scenario occurs. Only the slow start phase is shorter, as congestion avoidance is entered earlier, because the ssthresh is lowered twice. Therefore, the number of unnecessary retransmissions, due to the false fast recovery, is not as high, as the number of outstanding segments in the network (segments 6 in the figure) is smaller at the time of the false fast retransmit.

Limited receiver window Since the size of the cwnd determines the number of unnecessary retransmissions, a limited rwnd improves the situation, when a spurious timeout occurs. Figure 22(a) shows the effect of a smaller rwnd when the 20th data segment is delayed for 3300 milliseconds. The zoomed Figure 22(b) shows the beginning of the con-

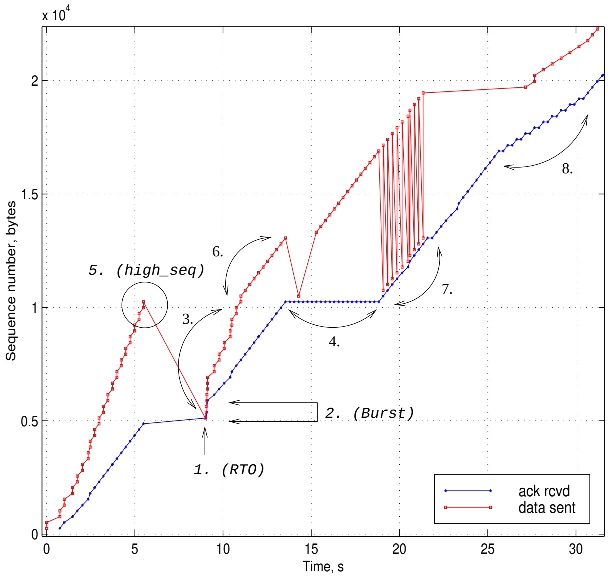
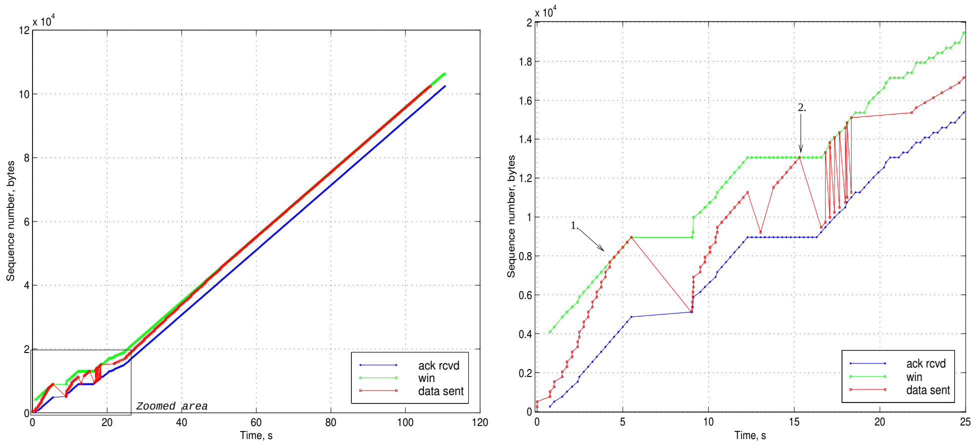


Figure 21: Baseline TCP: the 20th data segment is delayed for 3300 milliseconds. The delay causes multiple unnecessary retransmissions.

nection. The **rwnd** limits the number of segments transmitted by the TCP sender (numbers 1 and 2 in the figure). Naturally, the larger the **cwnd** would have been in absence of the limiting effect of the **rwnd**, the greater are the benefits of reducing the **cwnd** to 16 segments.



(a) Limited **rwnd** of 4 Kbytes, full connection

(b) Zoomed view

Figure 22: Limited **rwnd** of 4 Kbytes: the 20th data segment is delayed for 3300 milliseconds. A smaller **cwnd** reduces the number of unnecessary retransmissions

The use of a limited **rwnd** prevents the inflation of RTT and RTO calculations. As the number of outstanding segments does not grow beyond the **rwnd**, the RTT stays in reasonable limits. Under normal circumstances, it is favorable. However, if the `rwnd` is not reduced, the TCP sender tolerates longer delays, as the RTT estimates, and the equivalent RTOs, are inflated. For example, a delay that causes one spurious timeout for a segment when using regular Baseline TCP, may cause two consecutive RTOs for the segment if a limited `rwnd` is used[14].

Discussion The larger initial window of four segments was not tested, since it would not have changed the behavior upon a spurious timeout. The NewReno TCP modification would still have been used to recover from a spurious timeout, like in the Baseline tests. As the size of the `cwnd` determines the number of retransmissions, a larger IW would have reduced the performance, because the number of outstanding segments would have been higher.

The SACK option does not improve the performance, as no segments are dropped. The SACK blocks inform the TCP sender of the segments that are correctly received, but above the sequence number indicated by the corresponding `ACK`. Such scenarios occur, when some segments are dropped, but later segments are successfully transmitted to the receiver. The packet drop tests introduced in earlier chapters affirmed the benefits of SACK in an error-prone environment. In this current scenario, however, the TCP receiver *always* acknowledges the highest sequence number received, because no segments are dropped. Thus, SACK blocks cannot be transmitted to the TCP sender. The behavior would be exactly the same as with Baseline TCP. Preliminary tests were run, for both SACK and larger IW, that affirmed these conclusions.

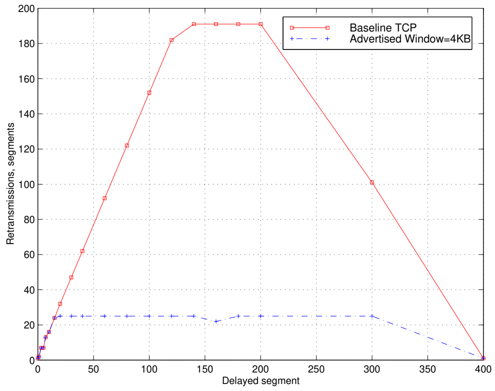
Since *all* retransmissions are unnecessary if a segment was only excessively delayed on the link, the behavior of the Baseline TCP is poor. Figure 23(a) shows the number of retransmissions when a segment was delayed. The extra time spent in retransmitting can be as long as 47 seconds, if the number of retransmissions is 191 segments, as it is in the worst case[15]. Obviously, the time lost is proportional to the number of transmitted segments, since there were no dropped segments. Figure 23(b) shows the achieved throughput for the Baseline TCP and limited receiver window of 4 Kbytes. Figure 23(c) shows the length of the excess delay causing an RTO[16]. It can be seen that once the `cwnd` is stabilized, and

[14] This can be seen in Figure 23(a) (discussed later) as a slight reduction in the retransmitted segments, when the 160th segment was delayed and a `rwnd` of 4 Kbytes was used. The `ssthresh` is lowered twice upon the two RTOs, and, thus, the number of unnecessary retransmits becomes smaller.

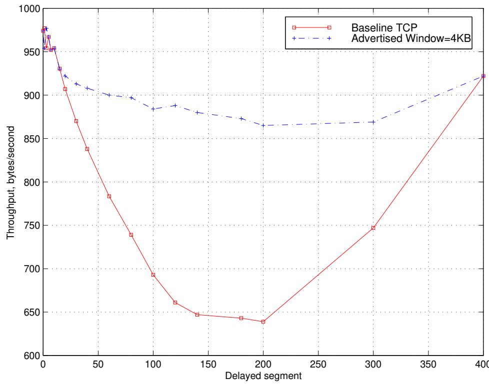
[15] The lost time is calculated by subtracting the length of the delay and the ideal connection time (102.05 seconds) from the actual connection time. For example, if the RTO occurred for the 200th data segment, the lost time is achieved by calculating $160.32s - 10.6s - 102.05s = 47.67s$

[16] The reader should refer to the Appendix C.1.3 to be aware of the RTO calculations used in Baseline TCP.

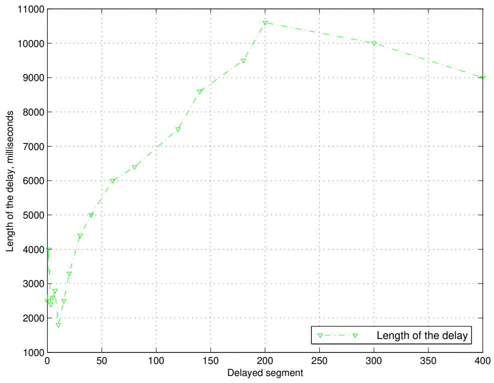
limited by the **rwnd** (after 200th segment), the RTO values begin to lower, as the RTT variation becomes smaller.



(a) The number of retransmissions



(b) The throughputs of the connections



(c) The length of the delay needed to cause a RTO

Figure 23: The statistics of the delay tests

The understanding is that the *Careful* variant of the NewReno bugfix would cause less retransmissions, as the *Less Careful* variant implemented by the Baseline TCP. However, it would not solve the retransmission ambiguity problem of TCP, as it changes only the requirements for invoking a fast retransmit. Thus, a full window of data would be retransmitted. Further research is needed before making final conclusions about the *Careful* variant, because it was not tested for this thesis. The extension to the SACK option, called D-SACK [FMMP00], possibly could avoid this problem to some extent, as it helps the TCP sender recognize the unnecessary retransmissions.

Summary of the delay tests

Because the full window of data is always retransmitted, the limited advertised window significantly reduces the number of unnecessary retransmissions as it prevents the congestion window from growing. By setting the receiver window to an arbitrary small size, the problem of unnecessary retransmits can be avoided, but it will cause great problems on an erroneous link as the TCP sender may not recover from a packet loss in any other way than with a retransmission timeout. The SACK option does not help in recovering from excess delays, if no packets are dropped. Larger initial window was not fully tested, but it is expected that it will not improve the performance in case of an RTO, but rather the opposite.

5.4 Combined effects of excess delays and packet drops

These tests were run over a link that contained both packet drops and excess delays. The probability of a delay of 6 seconds was 1%. In addition, the data packet drop rate of the link was 2%, 5% or 10%. The tests were repeated using Baseline, SACK, IW of four segments and a limited receiver window of 4 Kbytes (16 segments).

Baseline TCP Usually, the problems observed in these test cases are the same as outlined in packet drop tests (Section 5.2) and delay tests (Section 5.3). Since the link contains both packet drops and excess delays, the aggregate effects lead to a new problem, when studying the behavior of the Baseline TCP. In the previously described delay tests and packet drop tests, the *Careful* variant of the NewReno bugfix was considered to be better than the *Less Careful* variant, in case of a spurious timeout. The timeout was caused by an excess delay on the link, or by a large `cwnd` that prevented a new `ACK` to be received in time. The requirements for invoking a fast retransmit are more conservative, thus reducing the possibility of a false fast retransmit. However, it was observed that even the *Less Careful* variant may be too conservative, if there are packet drops on the same window, when an RTO occurs, and the same packet is dropped again, while retransmitting in slow start after the RTO. The bugfix eliminates the possibility of a fast retransmit in a situation, where it would be useful. Figure 24 shows one example.

When an RTO occurs due to an excess delay, the TCP sender retransmits the segment (number 1 in the figure), invokes slow start, and stores the highest sequence number sent (number 2). Due to segments stored at the receiver's link buffers, a burst of three `ACK`s eventually arrive, and `cwnd` is grown accordingly (number 3). One segment is dropped from the same window, when RTO occurred, and the same segment is dropped again,

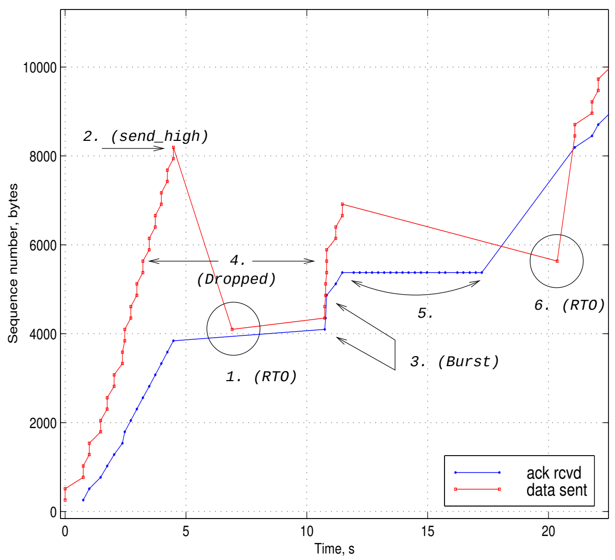


Figure 24: Baseline TCP: the effect of the NewReno bugfix

while in slow start after the RTO (segments number 4). The TCP receiver sends dupacks to the TCP sender (number 5) to report of a missing segment. Because of the NewReno bugfix, the TCP sender does not invoke a fast retransmit upon the third dupack; the dupacks do *not* cover the highest sequence number sent, before the RTO occurred. This causes a second RTO (number 6), which is twice as long as the previous, because of the *exponential backoff* that was invoked due to the first RTO. It must be noticed, however that this is only one scenario, where the NewReno bugfix does not work well. It is not a common situation, but possible. If there was an intermediate router, that suffered from buffer exhaustion, the scenario would be more likely; the router cannot reduce its queue because of the excess delay, and, as new packets arrive, the router has to discard them. That environment is studied in [Sar01], for example. Inferring, that the bugfix reduces the performance on a common level in our environment is an overstatement.

The number of retransmissions and actual packet drops are shown in Figure 25. Under good link conditions (i.e. the packet drop rate is 2%), the number of retransmissions is relatively high compared to the number of dropped segments. This is due to spurious timeouts, as the full window is retransmitted even if no segments were lost. Because only a few segments are dropped within a connection, the `cwnd` may grow without being halved upon a lost segment. Again, the large `cwnd` leads to numerous unnecessary retransmissions in case of a spurious retransmission timeout, as the whole window is retransmitted. The median connection time was 143.74 seconds.

Under mediocre link conditions (i.e. the packet drop rate is 5%), the number of unnecessary retransmissions drops heavily, when combining the number of retransmissions with actual packet drops. Although the spurious timeouts occur as often as they did in the former situation, all the retransmissions are not unnecessary, as more segments are dropped more likely. Therefore, invoking the slow start after an RTO does not retransmit all segments unnecessarily. Secondly, the `cwnd` is usually smaller due to dropped segments. Thus, the number of retransmissions is smaller after a spurious RTO. The total number of retransmissions under mediocre link conditions is even smaller, than under good link conditions. The median connection time was 163.79 seconds (+13.9% compared to good link conditions).

When the number of dropped segments increases to 10%, the number of retransmissions increases. The size of the `cwnd` stays low throughout the connection, as the dropped segments often halve the `cwnd`. Therefore, upon a spurious retransmission timeout, the number of unnecessary retransmissions is small. The median connection time was 195.13 seconds. The relative increases are 35.7% and 19.1% compared to good and mediocre link conditions, respectively.

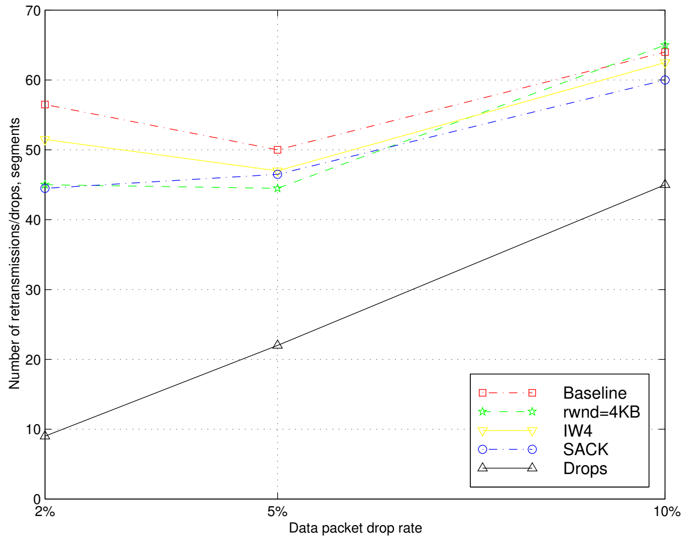


Figure 25: The number of retransmissions and packet drops over a link that also contains excess delays

Larger initial window Since a larger initial window does not change the behavior of the TCP sender, as it still uses the NewReno TCP modification to recover from the packet drops, the differences in the performance are minimal. The problems due to the NewReno bugfix still exist. As reported earlier in this thesis, the initially larger `cwnd` may prevent an RTO, if a packet drop occurs at the beginning of the connection. The higher number of outstanding segments allows the TCP receiver to send dupacks that may trigger a fast retransmit. Also, the capacity of the link is in full use from the very beginning of the connection, as the four initially sent segments fill the pipe. In case of a spurious retransmission timeout, however, the larger `cwnd` results in more unnecessary retransmissions, as the window of data is retransmitted. The overall performance is then dependent of the proportion of the excess delays and dropped segments over the connection. As the probability of the delay was only 1%, and packet drops occurred more often, the larger initial window gave minor advantage for the end-to-end performance. Larger IW of four segments gave 2.5% (3.6 seconds) and 3.3% (5.4 seconds) faster median connection times, than the Baseline TCP did, over good and mediocre link conditions, respectively. The results under bad link conditions are considered to be similar for both Baseline TCP and a larger initial window.

Limited receiver window As reported in Section 5.3, a limited `rwnd` reduces the number of unnecessary retransmissions in case of excess delays. A `cwnd` of 16 segments (i.e `rwnd`=4Kbytes), is big enough, if the link does not suffer from high packet drop rates; the `cwnd` is not often lowered to a value less than four segments which would lead to a retransmission timeout upon a lost segment. As the number of unnecessary retransmissions can be decreased by reducing the `cwnd`, the connection times for the limited receiver window of 4 Kbytes are 4.3% (6.2 seconds) and 6.6% (10.9 seconds) better than Baseline TCP's under good and mediocre link conditions, respectively. Under bad link conditions the connection time was 4.7% longer (9.2 seconds) than when using Baseline TCP. The difference was interpreted to be caused by the reduced `rwnd`, as it may occasionally prevent sending new segments. Although the `cwnd` is usually at a lower value than 16 segments, in some situations the `cwnd` could have inflated to be bigger. As a larger `cwnd` can prevent an RTO due to RTT inflation, the effect of excess delays is more harmful over a connection that has a small `cwnd`. It should be noticed that in plain random packet drop tests there were no differences between the Baseline and limited `rwnd`. The performance implications of a limited `rwnd` over an erroneous link that has excess delays need further study.

SACK SACK blocks provide extra information about the segments received by the TCP receiver. Thus, the TCP sender can retransmit more effectively the missing segments. The counterproductive effects of the NewReno bugfix are not as bad as they were with the Baseline TCP, because the SACK TCP sender can transmit new segments upon the dupacks, as well (see number 5 in Figure 24). Once a spurious RTO occurs, the SACK TCP sender cannot recover any better than the regular Baseline TCP sender. However, as there are packet drops during the transfer, the use of SACK option benefits the median connection times by 6% (8.58 seconds), 12.7% (20.79 seconds) and 20.9% (40.79 seconds) over good, mediocre and bad link conditions, respectively.

Discussion A conclusion of the effects of delays can be made while comparing the connection times of these tests to the results of random packet drop tests. The same packet drop rates were used, but the latter tests did not suffer from excess delays. Table 9 summarizes the median connection times for random packet drop tests and combined packet drop and delay tests. "Δt" is the difference in the connection times. This table confirms the interpretation that the excess delays are more harmful in an environment where the packet drops occurs less frequently. If the link conditions are bad, the consequences on retransmitting the whole window upon a spurious RTO are not that severe, as the packets would have been retransmitted anyway. Hence the difference between the two connections is only 16.3%, as it was 37% over mediocre link conditions.

Table 9: A comparison between erroneous connections with and without delays, Baseline TCP.

Drop rate	**Delay Prob**	**Mod**	**time (sec)**	**rexmt (Pkts)**	**Δt**
2%	-	Baseline	106.04	9.50	
2%	1%	Baseline	143.74	56.50	+35.6%
5%	-	Baseline	119.55	23.50	
5%	1%	Baseline	163.79	50.00	+37.0%
10%	-	Baseline	167.73	49.00	
10%	1%	Baseline	195.13	64.00	+16.3%

Table 10 shows the same statistics for SACK TCP. The effect of excess delays on the link is not as dependent of the erroneous link conditions. The median connection time is increased in the same order of magnitude throughout the different packet drop rates.

Table 10: A comparison between erroneous connections with and without delays, SACK.

Drop rate	**Delay Prob**	**Mod**	**time (sec)**	**rexmt (Pkts)**	**Δt**
2%	-	SACK	105.52	13.00	
2%	1%	SACK	135.16	44.50	+28.1%
5%	-	SACK	111.49	22.50	
5%	1%	SACK	143.00	46.50	+28.2%
10%	-	SACK	124.11	47.00	
10%	1%	SACK	154.35	60.00	+24.4%

The use of a limited `rwnd` cannot be recommended without better knowledge about the link characteristics, as the length of the delay is a significant factor, when comparing the performance of the TCP enhancements. If the connection suffers from excess delays, that usually cause RTOs, the limited `rwnd` reduces the number of unnecessary retransmissions. But, since the number of outstanding segments is smaller, the RTT estimates, and retransmission timer values, are much shorter, than in a regular scenario, where the `rwnd` does *not* limit the `cwnd`. Therefore, an excess delay that causes an RTO when using a limited `rwnd` may not cause an RTO on a regular scenario, as the RTT estimates are inflated due to large `cwnd`. As an RTO causes a whole window to be retransmitted, the RTOs *should* be avoided by all possible means. If the excess delay occurs at the beginning of the connection, the RTO is more likely to happen in both scenarios, because the RTT estimates are not yet inflated by the large `cwnd`. If the delay occurs in later phases of the connection, where the `cwnd` has possibly grown and RTT estimations have inflated, a limited `rwnd` may cause an RTO, that would not happen if it would not be in use.

Figure 26 summarizes the throughputs of different TCP enhancements. Baseline TCP was the worst performing TCP over good and mediocre link conditions. The higher the packet drop rate is, the more the SACK option benefits the connection. The number of retranmissions presented in Figure 25 correlates to the achieved throughput presented in Figure 26. Although the number of retransmitted segments is nearly the same with SACK and other TCP enhancements, SACK is able to retransmit more effectively due to SACK blocks that provide more information to the TCP sender.

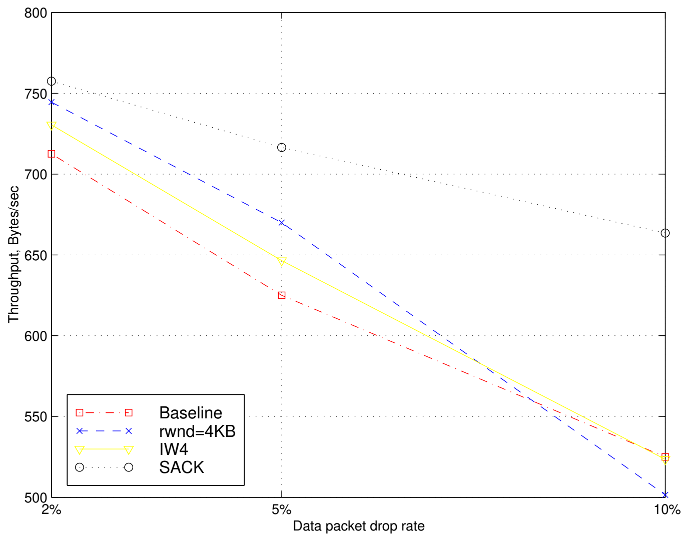


Figure 26: The throughput of different TCP modifications

Summary of the combined effects of excess delays and errors

The SACK option suits well in an error-prone environment even if it has excess delays. Other optimizations cannot reach that level of performance. Because the TCP cannot distinguish between a packet loss due to corruption or a packet loss due to congestion, the `cwnd` is halved upon the three dupacks. On the plain wireless link there was no congestion and the transmission speed was unnecessarily slowed down which led to suboptimal throughput. The larger initial window and the limited `rwnd` of 4 Kbytes may have positive effects over erroneous links that has excess delays.

5.5 Effect of different bandwidths

This section provides the results of the bandwidth tests. The tests included two different scenarios. First, different static bandwidths were tested (including bandwidth asymmetry). The bandwidth towards the mobile host (i.e downlink) was set to higher than in the opposite direction. Second, the bandwidth was subject to change during the slow start phase, or steady state. These tests are called *variable bandwidth tests.* The change was always from a higher bandwidth to a lower bandwidth. The results and analysis are given in this order, by first explaining the behavior of the Baseline TCP.

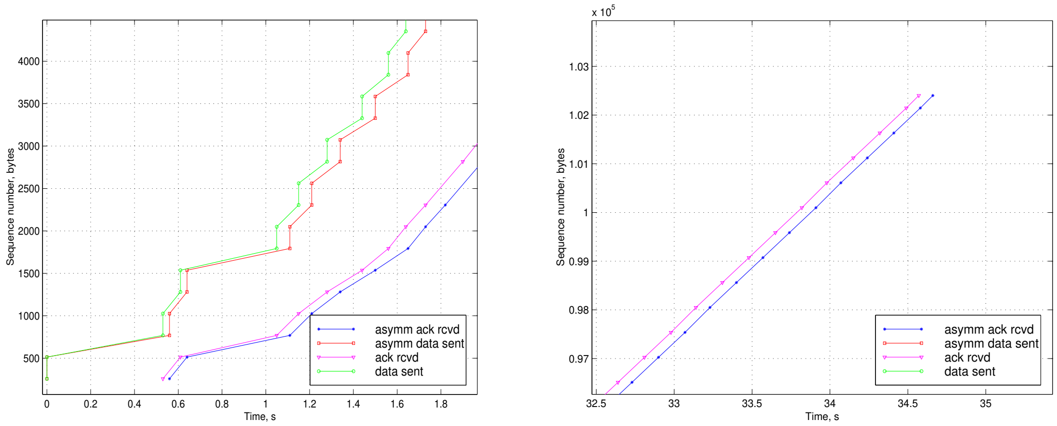
Static (and asymmetric) bandwidth

The exact bandwidth ratios that were tested are described in Section 4.5, page 26. The tests were repeated by using a limited receiver window of 2 and 4 Kbytes, and a larger initial window of four segments. In addition, a limited set of tests were run to gain knowledge about the effects of disabling the delayed acknowledgments.

Baseline TCP The *normalized asymmetry ratio, k*, was introduced in Section 3.4. Since in the tested environments, the pure bandwidth ratio is at most three, as the greatest difference in the tested bandwidths is 9600bps for the uplink and 3*9600=28800bps on the downlink, and the ratio of the packet sizes is 296bytes/40bytes=7.4[17], the normalized asymmetry ratio, k, is then approximately 3/7.4 = 0.4. As $k < 1$, the threshold for saturating the `ACK` path is not exceeded in any of the test cases. For that reason, no actual problems were found in the tested scenarios. The median connection times for all the tested bandwidths were proportional to the bandwidth of the data path, when comparing

[17] For the data segment: 256 bytes of TCP payload + 40 bytes of TCP/IP headers=296 bytes, and for the `ACK` segment: TCP/IP headers=40 bytes.

them to the regular 9600bps symmetric connection. A lower bandwidth in the uplink direction did not deteriorate the overall throughput. However, marginal differences were observed, that were due to bandwidth asymmetry. This can be seen when comparing the median connection times of the transfers over a 9600bps/28800bps (uplink/downlink) link versus a 28800bps/28800bps link. The difference is 0.15 seconds in favor of the symmetric bandwidth scenario. The interpretation is that the lost time is due to the first few `ACK`s that propagate slower to the TCP sender than in case if the bandwidth was symmetric. As the pipe is *not* full at the beginning, the time spent in waiting for the `ACK`s to arrive is lost, as new segments are not injected to the network to make use of the full capacity of the link. At later phases of the connection, the awaiting of the `ACK`s does not increase the connection time, as the pipe is already full. Figure 27 shows the traces of the symmetric and the asymmetric connections interlaced in the same graph. The bandwidth of the symmetric connection was 28800bps, and the bandwidth of the asymmetric connection was 9600bps for the uplink (i.e `ACK` path) and 28800bps for the downlink. Figure 27(a) is a microscopic view taken from the beginning of the connection. The TCP sender of the asymmetric connection cannot send new segments to the network as fast as the TCP sender that receives the `ACK`s via a faster uplink. Figure 27(b) shows the differences between the arrival times of the last `ACK`s of the transfers. The time lost at the end of the connection is not longer, as it was already at the beginning. It should be noticed, that if the transmission delay would be higher than the propagation delay, the differences would be even more visible. In this environment, the transmission delay of an `ACK` was only 0.04 seconds, when the uplink bandwidth was 9600bps, while the propagation delay was 0.2 seconds.



(a) Early slow start: two connections

(b) The end of the transfers: the last ACKs of the connections

Figure 27: The differences in early slow start and end of the transfer when using symmetric and asymmetric bandwidth

Larger initial window The capacity of the link becomes higher, when the bandwidth is augmented. For example, if the bandwidth is doubled from 9600bps to 19200bps, the link capacity is then the sum of propagation delays and transmission delays of the data and `ACK` packets multiplied by the bandwidth. This leads to a delay bandwidth product of 1312 bytes[18], which means that five data segments fill the link. Therefore, a larger initial window improves the performance, as the capacity of the link is faster in full use.

The benefits of an initial window of four segments are greater than in the regular 9600bps tests. In the earlier tests, the capacity of the pipe was 840 bytes. An initial window of three segments would have filled the pipe, and the use of a the fourth initially sent segments did not give any benefits. As the capacity of the link is more than four segments, each initially sent segment, up to the fourth segment, benefit the connection, as the pipe becomes more full. Therefore, the throughput increased up to 1.8% when comparing to the Baseline TCP. In regular 9600bps tests, a larger initial window of four segments gave only 0.2% higher throughput than the Baseline TCP. If the size of the initial window has grown over the delay-bandwidth product of the link, no further benefits are to be expected. This conclusion holds for large initial windows only. As reported earlier, an initial window of four segments helps the TCP sender better to recover from a lost segment at the beginning of a connection, because the incoming dupacks may trigger a fast retransmit, instead of falling to an inefficient retransmission timeout.

Limited receiver window By limiting the `rwnd` to 4 Kbytes, the performance is identical to the Baseline TCP's over ideal link conditions. However, the RTT and RTO calculations are different from when using the Baseline TCP. This may cause problems, if the link contains packet drops or excess delays. The effects of packet drops is discussed in Section 5.6.

If the `rwnd` is set to 2 Kbytes, the median connection time increases by 11.3% (3.0 seconds), if the bandwidth of the link is 14400bps/38400bps. The capacity of the link is 2352 bytes[19], so the maximum `cwnd` of eight segments should be enough. This means that, every time the TCP sender gets a new `ACK`, it should immediately send a new segment to keep the link fully utilized. However, this is not enough, as it takes a short while before a new `ACK` is processed at the receiver side due to delayed acknowledgments.

[18] $(2 * 200ms + 304B/2400Bps + 48B/2400Bps) * 2400Bps = 1312B$

[19] The capacity of the link is: $[0.4s + (304B/4800B) + (48B/1800B)] * 4800B = 2352B$. Thus, $2352B/304B \approx 7.7$ segments fill the pipe.

Table 11: The benefits of disabled delayed acknowledgments over a 14400bps/38400bps connection

Dis. delayed ACKs up to n segments	elapsed time (sec) or $\Delta(T)$ 2KB	5KB	10KB	20KB	50KB	100KB
$n = 2$	2.13	3.07	4.33	6.85	14.41	27.04
$n = 4$ (Baseline)	-0.07	-0.06	-0.07	-0.06	-0.06	-0.11**
$n = 6$	-0.08	-0.2	-0.21	-0.2	-0.2	-0.2
$n = 8$	-0.04**	-0.25	-0.26	-0.25	-0.25	-0.25
$n =$ all segments	-0.08	-0.26	-0.27	-0.26	-0.26	-0.26

Disabled delayed acknowledgments This TCP modification was tested only over an asymmetric connection of 14400bps/38400bps (uplink/downlink). The use of disabled delayed acknowledgments improves to the start-up phase of the early slow start. The `cwnd` grows faster, as each new `ACK` increases the `cwnd` by one SMSS. Therefore, as the `cwnd` grows faster to a level that fills the pipe, the connection time becomes shorter. However, the benefits are marginal, when transferring 100Kbytes. When comparing to a connection that disables the delayed acknowledgments up to the second segment, the improvement is at most 0.26 seconds ($\approx 1\%$). Table 11 summarizes the connection times for 2, 5, 10, 20, 50 and 100Kbytes. The absolute connection time is marked only for the TCP modification, which disabled the delayed acknowledgments up to the second data segment. This scenario is used as a reference. For other disabled delayed acknowledgment modifications, the differences in connection times compared to the reference connection are provided. The time saved remains the same after the first 5 Kbytes for all the modifications. The capacity of the link is fully in use after that, so no further benefits are observed. The connection time can be decreased by 8.5%, if only five Kbytes are transferred. The number of segments the TCP receiver should acknowledge immediately depends on the delay-bandwidth product of the link. As the capacity of the studied link is 2356 bytes[20], eight segments fill the pipe. Therefore, by disabling the delayed acknowledgments more than up to the eighth segment gives no benefits to the connection. The test results in Table 11 confirm this assumption, as the results are the same for disabling delayed acknowledgments up to the eighth segment ($n = 8$) and totally disabled delayed acknowledgments ($n =$ all segments)[21]

[20] $[(48B/1800Bps) + (304B/4800Bps) + (2 * 0.2)] * 4800Bps = 2356B$

[21] The 0.01 second difference between ($n = 8$) and ($n =$all segments) is probably due to inaccuracy in the emulator. The exact reason is unknown and cannot therefore be provided.

**This result exhibits anomalous behavior, and cannot be interpreted as a correct value.

Discussion The tests corroborated the assumption that a slower `ACK` path does not substantially decrease the throughput, if the normalized asymmetry ratio, k, is less than one. Obviously, if the data path is slower, the pure bandwidth of the data path determines the throughput.

The disabled delayed acknowledgments was tested over an asymmetric connection of 14400bps/38400bps. It is assumed that the environment possibly contributes to the benefits of the modification. As the `ACK` path is slow, the arrival of `ACK`s is more crucial for the connection to exit the early slow start phase. However, this is only speculation as no tests have been executed over a symmetric 38400bps link.

The size of the initial window up to the delay-bandwidth product of the link was considered to benefit the performance. However, too large a `cwnd` may lead to severe problems over a link that has an intermediate router between the hosts. If the router drops packets due to buffer exhaustion during the slow start, the number of dropped segments is in the order of half of the `cwnd` [BP95][Sar01]. Hence, arbitrary large initial windows are not considered as a good TCP enhancement; the `cwnd` should be grown according to the slow start that probes the capacity of the link, not by increasing it by an artificial TCP enhancement, which does not take the link characteristics into account.

The use of the SACK option was not tested as there was no packet drops during the transfer.

Variable bandwidth tests

In these tests the bandwidth was lowered during the transfer. The change in the bandwidth was made on two different occasions: in slow start and in steady state. The full test results can be found in Appendix A, Table 31.

Baseline TCP The changes in the transmission rate did not trigger a retransmission timeout. As the emulated link did not contain any packet drops or excess delays, the change in the transmission rate had an effect only on the throughput. The slow start phase of the connection becomes longer, if the transmission rate is lowered, as the arrival rate of the `ACK`s slows down. This is not an issue in the tested environments, because the full capacity of the link is already in use.

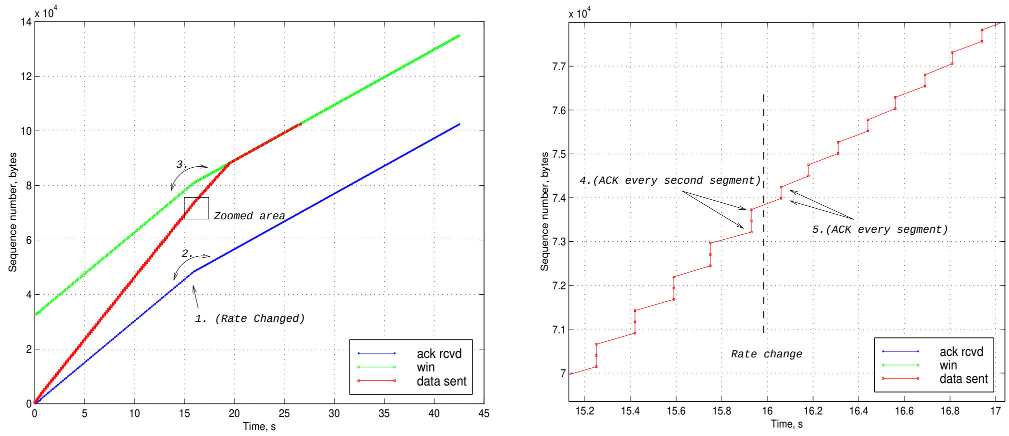
Larger initial window and limited receiver's window A larger initial window gave similar benefits, as in the static bandwidth tests. The four segments sent initially help the

TCP sender to fill the pipe faster than when using an initial window of two segments. A limited `rwnd` of 4 Kbytes did not affect the throughput of the connection, as no packets were dropped or delayed. However, it affects the RTT and RTO calculations similarly, as explained in Section 5.4, because the `cwnd` is smaller throughout the connection.

Discussion None of the changes in the bandwidths caused any problem. Immediately after the bandwidth was changed, the TCP sender will notice that the time interval at which the `ACK`s arrive, has grown. In the environment tested, there were no excess delays, but in the presence of additional delays, an RTO might occur, if the delay occurs along with the rate change. Such scenarios need to be further studied, as they were not included in these tests.

On some occasions, the TCP sender does not have to lower the transmission rate of new segments as much as the lowering of the bandwidth would imply. This is possible, if the TCP receiver changes its acknowledgment policy after the rate has been changed. An example is shown in Figure 28. The initial bandwidth of 28800bps was changed to 19200bps after 16 seconds (number 1 in the figure 28(a)). This can be seen as a slight adjustment in the arrival times of the received `ACK`s and `rwnd` (numbers 2 and 3). However, the transmission rate of the data segments does not become visibly lower. The Figure 28(b) shows a microscopic view of the data trace at the point where the bandwidth was altered. Before the change, the TCP receiver acknowledges every second segment due to delayed acknowledgments. As the connection is in slow start, the TCP sender transmits three new segments upon a new `ACK` (number 4). After the rate has changed, the TCP receiver does *not* acknowledge every second segment. Instead, it starts acknowledging every data segment, and the TCP sender gets a new `ACK` sooner than before the bandwidth was lowered. The transmission delay for two segments is 0.17 seconds (using a bandwidth of 28800bps), and for one segment 0.13 seconds, using the slower bandwidth of 19200bps. As the `ACK`s acknowledge only one segment, instead of two, the TCP sender may transmit two new segments upon a new `ACK` (number 5). Due to this behavior, the slow start phase does not take as long as it would have taken without the rate change.

There are at least two different consequences from this behavior. First, as the slow start is more aggressive, the intermediate routers may become faster overcongested. If the router overflow occurs in slow start phase, the number of expected packet drops is in the order of half the size of the `cwnd` [BP95][Sar01]. Second, if the capacity of the link is not in full use before the rate change, a more aggressive slow start after the change helps the TCP sender make full use of the link. Both of these scenarios are out of the scope of this thesis and they are not further discussed.



(a) A trace of a full connection

(b) A microscopic view of the data trace during the rate change: the effect of ack-every-segment

Figure 28: The effects of acknowledging every segment

Summary of the bandwidth tests

An initial widow of four segments gives more benefits if the capacity of the link is four segments or more; all initially sent segments are needed to fill the pipe. An advertised window of 4 Kbytes does not affect the performance in any way as there are no packet losses. By disabling the delayed acknowledgments, the connection time for a 100 Kbyte transfer decreases only by a fraction (less than 1%). However, when transferring smaller amounts of data, the connection time can be 8.5% shorter. The bandwidth asymmetry fractionally hinders the performance, as the slow ACK path prevents the TCP sender from filling the pipe as fast as it could when using a symmetric bandwidth.

In absence of packet losses and excess delays, the tested changes in transmission rate were not so big that they would have triggered an RTO. Combining the change in transmission rate and packet losses (or excess delays), the problem might become actual. The use of a larger initial window of four segments fractionally improves the performance.

5.6 Combined effects of bandwidths and packet drops

Random data packet drop rates of 2%, 5% and 10% was tested over different links. Several bandwidths was used to gain knowledge about the differences in performance. The bandwidths were the same as in static bandwidth tests. The full test results can be found in Appendix A, Tables 32-35.

Baseline TCP The bandwidth asymmetry hinders the performance if the packet drop rate is high. This is interpreted as a consequence of a slow ACK path, because the TCP sender is not able to get fast responses from the TCP receiver and make conclusions about dropped segments. Moreover, as the cwnd is small throughout the connection, the time lost in situations, where the capacity of the link is not in full use, cumulates over time. The reasoning for the time loss was given in Section 5.5, page 58. Figure 29 shows the median throughput for each studied bandwidth, when the Baseline TCP was used. The ideal link condition is added for reference[22]. As the normalized asymmetry ratio, k, is always less than one, the bandwidth of the data path correlates with the throughput when there are no or only a small amount of packet drops. The asymmetric bandwidth of 14400bps/38400bps gives the best throughput over good link conditions (i.e. the packet drop rate is 2%). The throughputs of a 9600bps/28800bps connection and a 28800bps/28800bps connection are equal, as $k < 1$. The worse the link conditions become, the more the asymmetric connections suffer from the slow ACK path. This can be seen in the graph, as the performance of all the tested asymmetric connections decrease more than the performance symmetric connections.

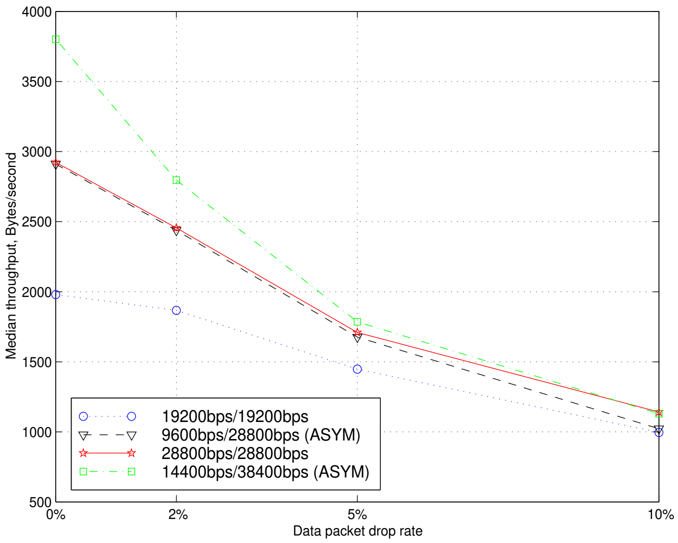


Figure 29: Baseline TCP: the median throughputs for different bandwidths over ideal, good, median and bad link conditions

The best achieved throughput (over a 28800bps/28800bps link) was 14.6% better than the worst throughput (over a 19200bps/19200bps link) over bad link conditions. The gain in throughput over bad link conditions is much less than the 50% higher bandwidth would have indicated. This can be explained by using an equation given by Mathis [MSMO97],

[22]The ideal link conditions are not widely discussed here, as they are already covered in static bandwidth analyses, in Section 5.5.

for example (Equation 1)[23]. It gives an estimate for the throughput (BW), while the packet drop rate is p. C is the constant of proportionality, and it is derived from the acknowledgment strategy (delayed acknowledgments vs. ack-every-segment), and loss mechanisms used by the TCP implementation. If delayed acknowledgments are used, C is set to $\sqrt{\frac{3}{4}}$.

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{1}$$

In the tests, the MSS and p are fixed, and only the RTT changes as the bandwidth is altered. Moreover, as the RTT constitutes significantly more on the fixed propagation delay (0.4 seconds) than on the transmission delay (less than 0.1 seconds over a 28800bps link), the benefits of a higher bandwidth become less important as the packet drop rate p gets higher. Therefore, the achieved throughputs are not highly related to the bandwidth of the link when the link conditions become worse.

In addition to these observations, no other problems were found that have not been discussed earlier in the regular random drop analyses.

Limited receiver's window As in earlier random drop tests, explained in Section 5.2, a limited `rwnd` of 2 Kbytes gave the worst performance. There are two main reasons for such behavior. Firstly, a `rwnd` of 2 Kbytes leads to a maximum `cwnd` of eight segments, which is too small over an erroneous link. As a packet drop halves the `cwnd`, the TCP sender has to eventually rely on the retransmission timer to be able to retransmit a lost segment, as the packet drop rate augments. Secondly, it is observed that each packet drop halves the `cwnd` to a value, which does not use the full capacity of the link; if the `cwnd` is limited by the `rwnd` when a lost segment is observed, the TCP sender cannot send any new segments during the recovery period. After receiving the new `ACK` that ends the fast recovery, the size of the `cwnd` is set to the minimum value of two segments, as earlier described in this thesis. An example of such a situation can be found in Figure 13, page 37. If the `cwnd` is *less* than eight segments upon a lost segment, the maximum value for the `cwnd` after halving it, is three segments.[24] Moreover, the capacity of the link is more

[23] This thesis does not explain the reasoning behind the formula, but simply uses it to comprehend the results of the tests. The equation is not taken as an exact derivation of the throughput, as it clearly gives false results if no packets are dropped. Also, the elapsed time consumed by RTOs is not modeled. The equation is used as a guideline to learn the basic factors that might determine the throughput. Another similar calculation has been provided in [PFTK98], for example.

[24] If `cwnd` is seven segments upon the third dupack, the new value is rounded to three segments due to the bitwise shift to the right, that is used for halving the `cwnd`.

than three segments over all the tested bandwidths[25], the pipe is underutilized every time a packet is dropped.

When using a limited `rwnd` of 4 Kbytes, the throughputs are almost equal with the Baseline TCP, if the bandwidth is symmetric. Over asymmetric links, the limited `rwnd` of 4 Kbytes do not reach the throughput of the Baseline TCP, when tested over good link conditions. The median connection time is 5.7% (2.4 seconds) longer, when the bandwidth of the link is 9600bps/28800bps. Similarly, over a 14400bps/38400bps link, the use of a limited `rwnd` leads to a 5.2% (1.9 seconds) longer connection time. The reasoning for the suboptimal performance is the following.

After a packet loss, the maximum size of the `cwnd` is seven segments, following the same reasoning as given in the previous chapter. This scenario was outlined also in regular random packet drop tests, Section 5.2. However, the consequences were not as noticeable as they are in these scenarios. The reason for the difference is that the pipe capacity is higher in the current tests. The TCP sender has to send seven segments to fill the pipe over a 9600bps/28800bps link[26]. Because the `rwnd` limits the `cwnd`, each packet drop –except the ones that occur when the `cwnd` is 14 or 15 segments– lead to underutilization of the link, as the `cwnd` is halved upon a lost segment. When using the Baseline TCP, the link is not underutilized as often, since the `cwnd` may grow beyond 16 segments, as there are only a small amount of packet drops. While the `ACK` path is slow, the consequences are more noticeable, as the `cwnd` is increased slower than over a symmetric link. The effects of a slow `ACK` path were introduced in Section 5.5. Limited `rwnd` does not change the fact that the deterioration of the performance is greater over asymmetric links, as the packet drop rate gets higher.

Larger initial window A larger initial window of four segments gave an advantage when tested over good and mediocre link conditions. The reasons are the same as previously described in this thesis; the `cwnd` grows faster to a safe size, which allows the TCP receiver to send three dupacks. The benefit is at most 8.7% (3.17 seconds), if the bandwidth is 14400bps/38400bps, and the packet drop rate is 2%. Usually, however, the advantages are not as big. The benefits are independent of the bandwidth symmetry.

[25] A symmetric 19200bps (2400Bps) link has the smallest capacity out of the studied bandwidths. The capacity is:$[0.4s + (304B/2400Bps) + (48B/2400Bps)] * 2400Bps = 1312B$. Therefore, the TCP sender has to send $1312B/304B = 4.3$ segments to fill the link.

[26] The capacity of the link is $[0.4s + (304B/3600B) + (48B/1200B)] * 3600Bps = 1888B$. Therefore, the TCP sender has to send $1888B/304B = 6.2$ segments to fill the link.

SACK As observed earlier, the SACK implementation retransmits aggressively, as it receives new information in the SACK blocks sent by the TCP receiver upon packet losses. The deterioration in the throughput is not as big over asymmetric connections, as it is when using other TCP enhancements. However, the bounding effects of bandwidth asymmetry are evident. The use of the SACK TCP option benefits the connection more, if the bandwidth of the link is symmetric.

Discussion The deployment of new W-WAN achitectures, such as GPRS, may lead to even greater bandwidth asymmetry. The bandwidth per direction may vary from zero throughput up to over 100 Kbps. GPRS also differentiates flows into four priority classes [BW97]. When the load in the system increases and the flow differentiation is taken into account, the bandwidth may become highly variable and therefore will affect the Quality-of-Service (QoS) of flows. Further research should be addressed for such an environment.

Summary of the combined effects

The advertised window of 2 Kbytes was the worst in all of the cases and SACK gave the best throughput. The advantages of SACK became explicit as the packet drop rate was grown. By limiting the advertised window to 4Kbytes, the performance was usually a little weaker than with the Baseline TCP. The greatest advantage the 4 Kbyte window gave was 2.7% in the connection time when the packet drop rate of 5% and bandwidth of 19200bps was used. On the other hand, Baseline TCP was at most 5.4% quicker than the 4KB window when the asymmetric bandwidth was 9600bps for uplink and 28800bps for downlink and the packet drop rate was 2%. The initial window of four segments gave better throughput over symmetric bandwidth when the packet drop rate was 5% when compared to Baseline TCP. Using the packet drop rate of 10%, the larger initial window did not give any benefits. Overall, the initial window of 4 segments may narrowly give better performance than Baseline TCP.

The asymmetry in the bandwidth does not alter the results when there are packet drops due to corruption compared to the basic GSM data link (9600bps). SACK TCP is the most useful TCP modification and the benefits or the disadvantages of a limited receiver window of 4 Kbytes are negligible and debatable.

5.7 Conclusion

As the evaluation of the TCP modifications has been provided in previous chapters, some results are gathered from different test scenarios. The focus of this conclusion is to help the reader to better understand and observe the effects of the link characteristics to the TCP performance. In addition, the differences between the TCP modifications are easier to discern, as the results are shown side by side.

The effect of a single event

Three different single events were studied: a packet loss, an excess delay, and a sudden change in the link's bandwidth.

The deteriorating effects of a single RTO and a single packet drop are combined in Figure 30. The graph shows the median throughputs for the tested TCP modifications, when a single event –a packet drop or an excess delay that causes an RTO– has occurred for a certain segment. The graph clearly affirms the conclusion that a spurious timeout is far more detrimental to a connection than a packet drop. Upon an RTO, the whole window of data is unnecessarily retransmitted. A single packet drop may cause suboptimal performance, if the `cwnd` is grown to a large value, thus leading the TCP sender to an RTO, as the next new `ACK` does not reach the TCP sender on time. Also, if the number of outstanding segments on the link is less than four, the TCP sender has to rely on the retransmission timer to be able to retransmit the missing segment.

The SACK option was not included in the tests, as it does not give any benefit to a connection. Also, larger initial window four segments actually worsens the performance, if a spurious RTO occurs, as the `cwnd` is bigger, thus leading to more unnecessary retransmissions. Preliminary tests were run that validated these assumptions, for both SACK and larger initial window. By limiting the `rwnd`, the pitfalls of too large a `cwnd` can be avoided for both RTO and packet drop.

The tested variations to the bandwidth were not so big that they would have cause any harmful effects. Especially, an RTO did not occur upon the rate change. However, as the RTT and RTO calculations are affected by the altered bandwidth, it is expected to lead to retransmission, if an excess delay occurs at the same time. These scenarios are left under further study, as they were not included in the tests. The acknowledging policy implemented by the TCP receiver may contribute to the performance. It was noticed that by disabling the delayed acknowledgments at the time the rate change may help the TCP sender to faster fill the link. Additionally, it may lead to a faster buffer overflow at the

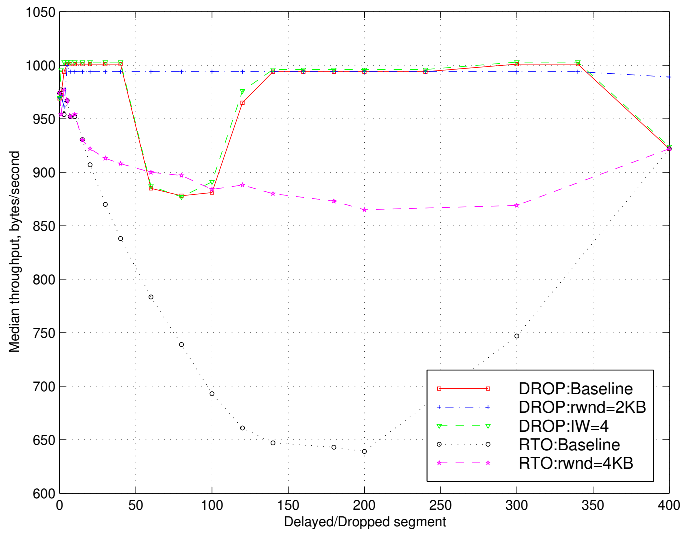


Figure 30: The effects of a single packet drop or a retransmission timeout

intermediate routers.

The analyses of single packet drops, excess delays and variable bandwidth was was given in Section 5.2, Section 5.3 and Section 5.5, respectively. The full test results can be found in Appendix A, Tables 14-17, Tables 20-22 and Table 31.

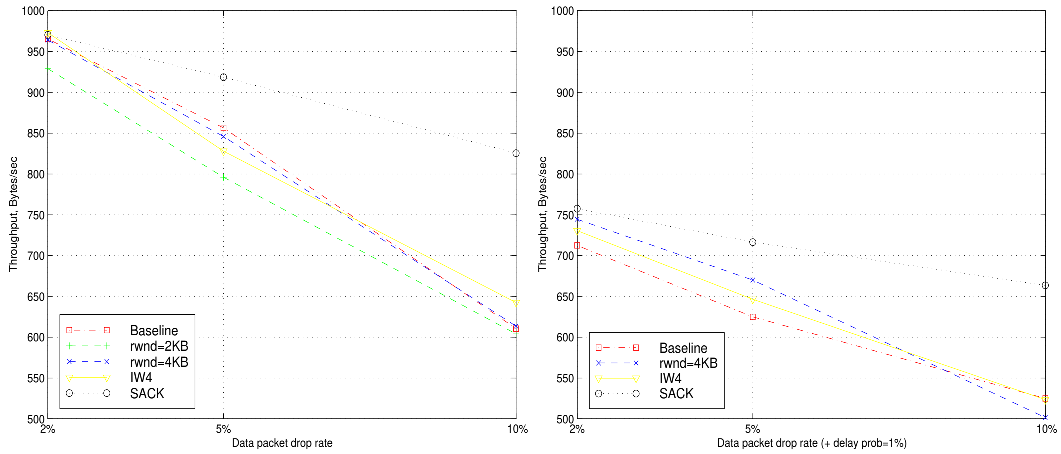
The combined effects of delays and errors

As a spurious RTO leads to numerous unnecessary retransmissions, the combining effects of packet losses and excess delays have great impact on the performance. Figure 31(a) shows the achieved throughputs over connections, where the packet drop rate was 2%, 5% or 10%. Figure 31(b) shows the similar statistics that were achieved over a link that suffers from excess delays, as well. The probability of a delay of six seconds is 1%. For better comparison, the graphs are provided in the same scale. In the presence of excess delays, the median connection time for the Baseline TCP increases by 35.6% (37.7 seconds), 37% (44.24 seconds) and 16.3% (27.14 seconds) over good, mediocre and bad link conditions, respectively.

The implications of spurious RTOs becomes less severe when the packet drop rate reaches a certain threshold. The test results indicate that the threshold limit would be a packet drop rate of 5%. However, the tests were not sufficient to make exact conclusions.

The SACK option gives more benefits to the performance the more packet drops there are on the link. The Baseline TCP performs well in an environment that has no excess delays. In the presence of excess delays, the limited `rwnd` efficiently reduces the number of unnecessary retransmissions, if the packet drop rate is not high. A larger initial window may benefit the connection if there are excess delays.

The analyses of the combined effects of packet drops and excess delays was given in Section 5.4 and the full test results can be found in Appendix A, Tables 23.



(a) A link that suffers from corruption losses

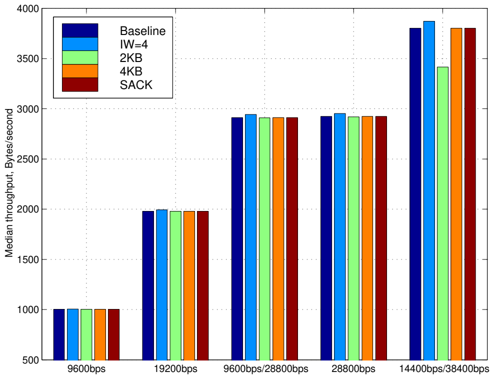
(b) A link that suffers from corruption losses and excess delays

Figure 31: The deteriorating effects of excess delays and packet drops

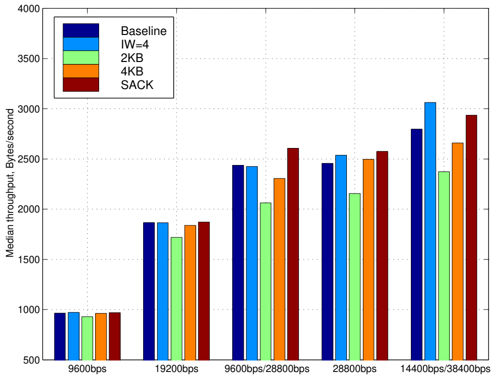
The effects of bandwidth and packet losses

For easier comparison, the throughputs of all the tested TCP enhancements over different bandwidths and packet drop rates are collected in Figure 32. The graphs are categorized by the packet drop rate. This allows the reader to better compare the effects of the bandwidth over a certain packet drop rate. All the graphs are provided in the same scale. Another version of the same statistics can be found in Appendix A, Figure 33, page 102, where the graphs are categorized by the bandwidth of the link. The effects of different packet drop rates over a certain bandwidth are easier to compare in that graph.

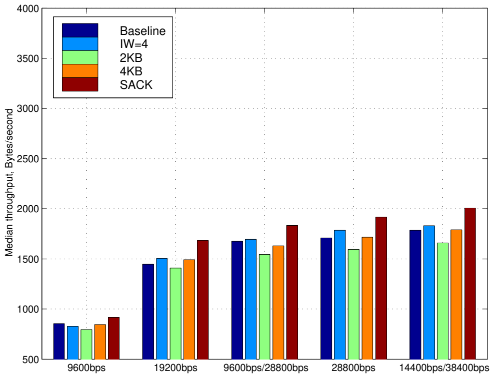
When looking at Figure 32(a), where no packet are dropped, it can be observed that the larger initial widow of four segments is the only TCP enhancement, that benefits the connection. As no packets are dropped, the capacity of the link is faster in full use. If the delay-bandwidth product of the link is not four segments, or more, the full benefits of a larger initial window of four segments are not achieved. This can be seen on the graph, as the advantages of the connections that used an initial window of four segments augment along with the bandwidth. This conclusion was earlier made in Section 5.5. The limited `rwnd` of 2 Kbytes hinders the connection, if the capacity of the link is approximately



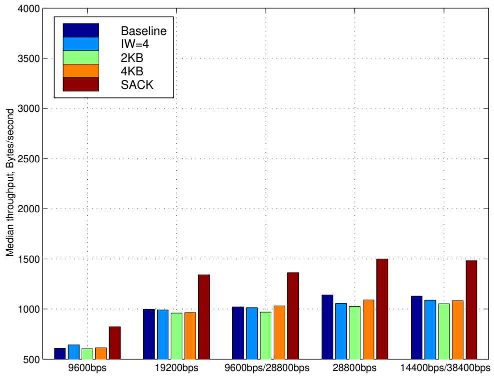
(a) Ideal link conditions, no packets are dropped



(b) Packet drop rate of 2%



(c) Packet drop rate of 5%



(d) Packet drop rate of 10%

Figure 32: The throughputs of every tested TCP enhancement over links with various characteristics

eight segments. Even the `cwnd` is eight segments, the time consumed by the delayed acknowledgments and processing lead to underutilization of the link, as a segment cannot be transmitted immediately after an `ACK` has been received.

As the packet drop rate rises, the bandwidth of the link does not correlate to the achieved throughput. As explained in Section 5.6, the RTT determines the performance of the connection. The RTT consists of a fixed propagation delay and the transmission delay of the segments. As the fixed propagation delay ($2 * 200ms = 400ms$) is a more significant factor than the transmission delay (less than 100ms over a 28800bps link), the effects of the bandwidth are minor compared to the propagation delay. This can be seen throughout all Figures 32(a-d), as the differences in the throughputs get smaller as the packet drop rate augments.

The bandwidth asymmetry exhibits minor performance losses over erroneous link connections, even if the *normalized asymmetry ratio, k*, is less than one. However, the observed differences are small, and cannot be easily seen in the graphs. When the capacity of the link is not in full use, the slow `ACK` path hinders the growth of the `cwnd`. If the link suffers from numerous packet losses, the `cwnd` is usually below the value that would make full use of the link. A more detailed description of the problem was given in Section 5.6.

The analyses of the effects of packet drops over different bandwidths was given in Section 5.6 and the full test results can be found in Appendix A, Tables 32-35. The analyses of static bandwidths over ideal link conditions was given in Sections 5.1 and 5.5, and the statistics of the tests can be found in Appendix A, Tables 13 and 24.

Evaluation of the tested TCP enhancement

To give an overview of the suitability of a TCP enhancement over a certain link environment, the results are gathered in Table 12. It should be remembered that the results given are heavily generalized, as the behavior of a TCP modification can vary extremely within the categorized environments. However, some general guidelines can be given. Before making any conclusions about the suitability of a TCP modification, one should (at least) read the equivalent chapters, where the behavior of the modification is further explained.

Table 12: The benefits of the TCP enhancement compared to the Baseline TCP

Environment	**Baseline TCP**	**2KB**	**4KB**	**IW=4**	**SACK**	**Dis. delacks**
Single drop	Varies	Good	Good	Varies	Varies	-
Random drop	Good	Bad	Good	Average	Excellent	-
Single delay	Bad	Good	Good	Bad	Bad	-
Drops and Delays	Bad	-	Good	Average	Excellent	-
Asymmetric bw	Good	Bad	Good	Good	Good	Good
Variable bw	Good	-	Good	Good	-	-
Static bw and Drops	Good	Bad	Average	Good	Excellent	-

6 Summary

The performance of different TCP modifications was tested over a slow wireless link. The empirical tests were run in five major categories. First, the emulated link suffered from corruption-related packet losses. In the absence of an intermediate router, which could have dropped packets due to buffer exhaustion, the congestion-related packet drops were not considered. Both single packet drops and random packet drops were studied. Second, a persistently reliable link, which caused excess delays due to link level retransmissions, was studied. Third, the performance implications of a link containing excess delay along with corruption losses were analyzed. Fourth, the bandwidth of the link was altered. Bandwidths up to 38400bps were studied, including bandwidth asymmetry and variance during a connection. Finally, the combined effects of different packet drop rates over several bandwidths were analyzed. By combining different link characteristics, the possible problems were observed more comprehensively. The large number of different test scenarios assisted to expansively understand and observe the behavior of the TCP under various link conditions; it is the main contribution of this study.

The experiments were made by using a real-time emulator over a dedicated network. The link characteristics were emulated by the software. Some of the link properties were selected to be GSM-like. However, the link characteristics are common for most of the Wireless Wide-Area-Networks. Hence, the results of this thesis can be exploited for all slow wireless links that suffer from excess delays and corruption losses.

The Baseline TCP was used as a reference TCP. It is based on the Linux TCP implementation, which was modified to confront the given specifications. The tests were repeated using several TCP enhancements. The workload used was a bulk data transfer of 100 Kbytes.

Several problems in the behavior of the TCP was found. A large congestion window, which has grown well beyond the delay-bandwidth product of the link, was observed to cause suboptimal performance upon a spurious retransmission timeout, in the absence of packet drops. As the TCP suffers from retransmission ambiguity, the whole window of data is unnecessarily retransmitted, even if no packets are lost. Moreover, the *Less Careful* variant of the NewReno "bugfix" leads to additional unnecessary retransmissions. Surprisingly, even a single packet drop may cause numerous unnecessary retransmissions, if the congestion window is too big. The multiple segments that are in the queue to be transmitted before the gap-filling `ACK` lead the TCP sender to a retransmission timeout. As a consequence, several packets are unnecessarily retransmitted. The consequences of too large a congestion window were efficiently avoided by limiting the receiver's advertised window.

In the case of random packet losses, the TCP sender eventually has to rely on the retransmission timer, if a retransmitted segment is dropped. Several consecutive retransmission timeouts were often observed over links with a high packet drop rate. The use of the SACK option was clearly the best out of the tested TCP enhancements in such an environment.

The combined effects of excess delays and random packet drops introduced a new problem. The NewReno "bugfix" may prevent a retransmission in a situation where it would be useful, and it forces the TCP sender to rely on the retransmission timer to be able to send the missing segment. The event was not very frequent, as it heavily depends on the occurrence of the excess delay and packet drops. However, it is quite usual in environments, where an intermediate router may drop packets due to buffer exhaustion, as the packet drops occur more regular basis [Sar01].

The bandwidth asymmetry was noticed to have effects on the performance, in some situations. Although the *normalized asymmetry ratio* was less than one, a slow `ACK` path prevented the TCP sender to efficiently fill the link at the beginning of the connection. The delay-bandwidth product of the link was observed to have effects on the benefits of a larger initial window, limited receiver's window, and disabled delayed acknowledgments.

The benefits of the SACK option was observed to be unquestionable over error-prone links. However, as it does not resolve the retransmission ambiguity problem of TCP, the deteriorating effects of a spurious retransmission timeout are still existing. Future research has to be addressed to D-SACK, as it helps the TCP sender recognize unnecessary retransmissions.

References

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

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

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

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

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

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

[BB95] A. Bakre and B. R. Badrinath. I-TCP: Indirect tcp for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 136–143, May 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.

[BP95] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected areas in Communications*, 13(8):1465–1480, October 1995.

[BP00] H. Balakrishnan and V. N. Padmanabhan. TCP performance implications of network asymmetry. IETF Internet Draft "draft-ietf-pilc-asym-01.txt", March 2000. Work in progress.

[BPK97] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *ACM MOBICOM*, September 1997.

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

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

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

[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.

[CI95] R. Caceres and L. Iftode. Improving the performance of reliable transport protocols in mobile computing environment. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[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.

[ETS95] ETSI. Radio link protocol for data and telematic services on the mobile station - base station system (MS-BSS) interface and the base station system - mobile switching center (BSS-MSC) interface, December 1995. GSM Specification 04.22, Version 5.0.0.

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

[FH99] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. IETF RFC 2582, April 1999.

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

[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.

[Goo97] D. J. Goodman. *Wireless Personal Communications Systems*. Addison-Wesley, 1997.

[Gur00] A. Gurtov. TCP performance in presence of congestion and corruption losses. Master's thesis, Department of Computer Science, University of Helsinki, December 2000. Available at: http://www.cs.Helsinki.FI/group/iwtcp/papers/.

[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.

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

[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.

[KRA96] M. Kojo, K. Raatikainen, and T. Alanko. Connecting mobile workstations to the internet over a digital cellular telephone network. In *Workshop on Mobile and Wireless Information Systems MOBIDATA*, November 1996. Rutgers University, NJ.

[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.

[Lee91] W.C.Y. Lee. Overview of cellular CDMA. *IEEE Transactions on Vehicular Technology*, 40(2):291–302, May 1991.

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

[LK00] R. Ludwig and R. H. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, 30(1), January 2000. Available at: http://www.acm.org/sigcomm/ccr/archive/2000/jan00/ccr-200001-ludwig.html.

[LMS97] T.V. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgment channel: a study of TCP/IP performance. In *INFOCOM 97*, volume 3, pages 1199–1209, 1997.

[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.

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

[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.

[Pax97] V. Paxson. End-to-end internet packet dynamics. In *ACM SIGCOMM '97*, pages 139–152, September 1997. Cannes, France.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: a simple model and its empirical validation. In *ACM SIGCOMM '98*, pages 303–314, August/September 1998. Vancouver, Canada.

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

[Pos81a] J. Postel. Internet protocol. IETF RFC 791, September 1981.

[Pos81b] J. Postel. Transmission control protocol. IETF RFC 793, 1981. Standard.

[PS97] C. Partridge and T. J. Shepard. TCP/IP performance over satellite links. *IEEE Network*, 1997.

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

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

[Sal99] A. K. Salkintzis. Packet data over cellular networks: the CDPD approach. *IEEE Communications Magazine*, 37(6):152–159, June 1999.

[Sar01] P. Sarolahti. Performance analysis of TCP improvements for congested reliable wireless links. Master's thesis, Department of Computer Science, University of Helsinki, February 2001. Available at: http://www.cs.Helsinki.FI/group/iwtcp/papers/.

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

[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.

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

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

A Full Test Results

This appendix describes the summarized test results for the tests that were run over a plain wireless link. Thus, all the packet losses were due to corruption. Congestion related packet drops did not exist. We give three different percentiles, first quartile (25% percentile), median (50% percentile) and third quartile (75% percentile) for the connection time (in seconds) for each test we ran. If the link contained random packet drops, the median value of dropped packets is given. Abbreviations are used in the tables to keep it more simple. *Baseline* means Baseline TCP outlined in Appendix C, *iw4* stands for larger initial window of four segments, Qn is the modification that disables the delayed acknowledgments up to nth segment, *Dis.delacks* demotes that the disabled delayed acks are applied throughout the connection, *2KB* and *4KB* mean the size of the limited receiver's advertised window, and *SACK* is used for the selective acknowledgment option. *tput* (throughput) and *rexmt* (retransmitted packets) are also the median values. *drops* (the number of dropped packets on the link) is derived from the log file of the *Seawind* emulator.

Table 13: The statistics for a 100 Kbytes transfer under ideal link conditions

Link	**Mod**	**elapsed time (sec)** 25 %	50 %	75 %	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
Ideal link	Baseline	102.05	102.06	102.06	1003.00	0.00	0.00
Ideal link	4kb	102.05	102.06	102.07	1003.00	0.00	0.00
Ideal link	iw4	101.85	101.85	101.85	1005.00	0.00	0.00

Table 14: Single packet drop tests: Baseline TCP, 20 replications

Packet dropped	Setting	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
402nd	Baseline	111.07	111.08	111.09	922.00	1.00	1.00
342nd	Baseline	102.31	102.32	102.34	1001.00	1.00	1.00
302nd	Baseline	102.31	102.31	102.32	1001.00	1.00	1.00
242nd	Baseline	102.98	102.99	103.02	994.00	1.00	1.00
202nd	Baseline	102.98	102.98	103.00	994.00	1.00	1.00
182nd	Baseline	102.98	102.99	103.00	994.00	1.00	1.00
162nd	Baseline	102.97	102.98	103.00	994.00	1.00	1.00
142nd	Baseline	102.98	102.99	103.00	994.00	2.00	1.00
122nd	Baseline	106.09	106.10	106.13	965.00	16.00	1.00
102nd	Baseline	116.18	116.19	116.20	881.00	56.00	1.00
82nd	Baseline	116.68	116.69	116.70	878.00	58.00	1.00
62nd	Baseline	115.68	115.68	115.69	885.00	54.00	1.00
42nd	Baseline	102.31	102.31	102.32	1001.00	1.00	1.00
32nd	Baseline	102.32	102.33	102.71	1001.00	1.00	1.00
22nd	Baseline	102.31	102.32	102.33	1001.00	1.00	1.00
17th	Baseline	102.31	102.32	102.33	1001.00	1.00	1.00
12th	Baseline	102.31	102.32	102.33	1001.00	1.00	1.00
9th	Baseline	102.31	102.32	102.33	1001.00	1.00	1.00
7th	Baseline	102.34	102.35	102.36	1001.00	1.00	1.00
5th	Baseline	102.98	102.99	102.99	994.00	1.00	1.00
4th	Baseline	107.44	107.44	107.46	953.00	1.00	1.00
3rd	Baseline	105.34	105.34	105.36	972.00	1.00	1.00
1st	Baseline	105.62	105.63	105.65	969.00	1.00	1.00

Table 15: Single packet drop tests: limited receiver's window of two Kbytes, 20 replications

Packet dropped	**Setting**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
402nd	2kb	103.56	103.56	103.58	989.00	1.00	1.00
342nd	2kb	102.98	102.99	103.01	994.00	1.00	1.00
302nd	2kb	102.98	102.98	102.99	994.00	1.00	1.00
242nd	2kb	102.98	102.99	103.00	994.00	1.00	1.00
202nd	2kb	102.97	102.98	103.00	994.00	1.00	1.00
182nd	2kb	102.97	102.97	103.00	994.00	1.00	1.00
162nd	2kb	102.97	102.99	103.02	994.00	1.00	1.00
142nd	2kb	102.97	102.98	103.00	994.00	1.00	1.00
122nd	2kb	102.97	102.98	102.99	994.00	1.00	1.00
102nd	2kb	102.97	102.98	103.00	994.00	1.00	1.00
82nd	2kb	102.97	102.99	102.99	994.00	1.00	1.00
62nd	2kb	102.97	102.99	102.99	994.00	1.00	1.00
42nd	2kb	102.97	102.98	103.02	994.00	1.00	1.00
32nd	2kb	102.98	102.98	102.99	994.00	1.00	1.00
22nd	2kb	102.98	102.99	103.01	994.00	1.00	1.00
17th	2kb	102.98	102.98	102.99	994.00	1.00	1.00
12th	2kb	102.97	102.98	102.99	994.00	1.00	1.00
9th	2kb	102.97	102.98	102.99	994.00	1.00	1.00
7th	2kb	102.34	102.35	102.36	1000.00	1.00	1.00
5th**	2kb	106.58	106.60	106.61	961.00	1.00	1.00
3rd	2kb	105.32	105.33	105.35	972.00	1.00	1.00
1st	2kb	105.61	105.62	105.63	970.00	1.00	1.00

**These results were due to a implementation bug, and the TCP sender was not able to do a fast retansmit. Thus, a RTO occured.

Table 16: Single packet drop tests using IW of four segments, 20 replications

Packet dropped	Setting	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
402nd	iw4	110.84	110.85	110.86	924.00	1.00	1.00
342nd	iw4	102.10	102.10	102.11	1003.00	1.00	1.00
302nd	iw4	102.09	102.10	102.10	1003.00	1.00	1.00
242nd	iw4	102.77	102.77	102.78	996.00	1.00	1.00
202nd	iw4	102.77	102.78	102.79	996.00	1.00	1.00
182nd	iw4	102.76	102.77	102.78	996.00	1.00	1.00
162nd	iw4	102.76	102.77	102.78	996.00	1.00	1.00
142nd	iw4	102.77	102.78	102.79	996.00	2.00	1.00
122nd	iw4	104.87	104.88	104.89	976.00	12.00	1.00
102nd	iw4	114.96	114.97	114.97	891.00	52.00	1.00
82nd	iw4	116.47	116.75	116.98	877.00	59.00	1.00
62nd	iw4	115.46	115.47	115.47	887.00	54.00	1.00
42nd	iw4	102.10	102.10	102.11	1003.00	1.00	1.00
32nd	iw4	102.10	102.11	102.11	1003.00	1.00	1.00
22nd	iw4	102.10	102.10	102.11	1003.00	1.00	1.00
17th	iw4	102.10	102.10	102.11	1003.00	1.00	1.00
12th	iw4	102.10	102.11	102.23	1003.00	1.00	1.00
9th	iw4	102.09	102.10	102.11	1003.00	1.00	1.00
7th	iw4	102.09	102.10	102.10	1003.00	1.00	1.00
5th	iw4	102.10	102.10	102.11	1003.00	1.00	1.00
4th	iw4	102.63	102.64	102.65	998.00	1.00	1.00
3rd	iw4	102.76	102.77	102.78	996.00	1.00	1.00
1st	iw4	105.61	105.62	105.64	969.50	1.00	1.00

Table 17: Single packet drop tests: SACK, 20 replications

Packet dropped	**Setting**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
402nd	SACK	111.07	111.08	111.09	922.00	1.00	1.00
342nd	SACK	102.32	102.32	102.33	1001.00	1.00	1.00
302nd	SACK	102.32	102.32	102.34	1001.00	1.00	1.00
242nd	SACK	102.98	103.00	103.01	994.00	1.00	1.00
202nd	SACK	102.99	103.00	103.00	994.00	1.00	1.00
182nd	SACK	102.99	103.00	103.01	994.00	1.00	1.00
162nd	SACK	102.98	103.00	103.02	994.00	1.00	1.00
142nd	SACK	102.99	102.99	103.01	994.00	2.00	1.00
122nd	SACK	104.59	104.60	104.63	979.00	10.00	1.00
102nd	SACK	114.67	114.68	114.69	893.00	50.00	1.00
82nd	SACK	114.92	114.93	114.94	891.00	51.00	1.00
62nd	SACK	113.15	113.17	113.19	905.00	44.00	1.00
42nd	SACK	111.14	111.15	111.15	921.00	36.00	1.00
32nd	SACK	102.32	102.33	102.34	1001.00	1.00	1.00
22nd	SACK	102.33	102.34	102.38	1001.00	1.00	1.00
17th	SACK	102.32	102.33	102.34	1001.00	1.00	1.00
12th	SACK	102.33	102.33	102.34	1001.00	1.00	1.00
9th	SACK	102.32	102.32	102.33	1001.00	1.00	1.00
7th	SACK	102.42	102.43	102.44	1000.00	1.00	1.00
5th	SACK	102.42	102.43	102.44	1000.00	1.00	1.00
4th	SACK	102.64	102.65	102.68	997.50	1.00	1.00
3rd	SACK	102.91	102.92	102.93	995.00	1.00	1.00
1st	SACK	105.62	105.64	105.65	969.00	1.00	1.00

Table 18: Random drop tests, 50 replications

Drop Rate	Settings	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
2%	Baseline	104.51	106.04	108.89	965.50	9.50	9.00
2%	2kb	107.11	110.21	113.27	929.00	8.00	8.00
2%	4kb	104.79	106.19	108.93	964.00	8.00	8.00
2%	iw4	103.64	105.23	107.96	973.00	9.00	8.00
2%	sack	104.23	105.52	110.69	970.50	11.00	8.00
5%	Baseline	115.17	119.55	127.01	856.50	23.50	21.50
5%	2kb	120.66	128.68	136.36	796.00	23.50	22.50
5%	4kb	114.71	121.03	126.21	846.00	22.00	21.00
5%	iw4	116.57	123.71	131.70	828.00	24.00	22.00
5%	sack	109.44	111.49	114.07	918.50	23.50	22.00
10%	Baseline	158.45	167.73	183.88	610.50	49.00	46.00
10%	2kb	159.37	169.62	185.26	604.00	48.00	45.00
10%	4kb	156.46	166.89	191.51	613.50	50.50	46.00
10%	iw4	146.14	159.38	178.60	642.50	47.00	43.50
10%	sack	119.99	124.11	131.60	825.50	47.00	44.00
2%	mss512	95.47	96.41	99.74	1062.00	4.50	3.50
2%	mss512, 2kb	96.85	98.04	101.96	1044.50	4.00	4.00
2%	mss512, 4kb	96.43	97.38	98.84	1051.50	4.00	4.00
2%	mss512, iw4	96.04	96.62	99.12	1059.50	5.00	5.00
2%	mss512, sack	95.93	96.86	100.98	1057.00	5.50	4.00
5%	mss512	100.37	104.98	112.06	975.50	11.00	11.00
5%	mss512, 2kb	107.37	110.22	117.39	929.00	11.00	10.50
5%	mss512, 4kb	102.07	107.22	115.22	954.00	10.00	10.00
5%	mss512, iw4	101.87	105.02	119.04	975.50	12.00	10.50
5%	mss512, sack	98.89	101.27	104.08	1011.00	11.50	10.50
10%	mss512	124.75	138.61	149.48	739.00	25.00	22.00
10%	mss512, 2kb	135.44	147.51	170.31	694.00	25.50	24.00
10%	mss512, 4kb	129.52	145.29	173.16	705.00	26.00	24.00
10%	mss512, iw4	118.75	137.53	164.03	744.50	24.50	22.50
10%	mss512, sack	105.89	109.20	113.39	937.50	23.50	22.00

Table 19: Random drops: packet drops occur on both up- and downlink, 50 replications

Drop Rate	**Settings**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
2%	Baseline	104.07	106.29	112.00	963.00	9.00	17.00
2%	4kb	105.06	107.25	110.04	954.50	8.00	16.00
2%	iw4	104.15	105.97	109.76	966.50	11.00	18.00
2%	sack	104.92	107.26	111.11	955.00	13.00	15.50
5%	Baseline	120.13	125.01	135.20	819.00	23.00	41.00
5%	4kb	117.56	124.82	136.00	820.50	21.00	42.00
5%	iw4	116.11	124.44	136.38	822.50	22.50	41.00
5%	sack	109.97	112.65	115.23	909.00	22.50	43.00
10%	Baseline	196.12	217.38	249.20	471.00	54.00	89.50
10%	4kb	180.75	208.26	234.35	491.50	52.00	88.00
10%	iw4	178.06	210.22	248.58	487.00	55.50	85.00
10%	sack	128.16	137.42	146.00	745.00	47.00	84.00
2%	mss512	95.50	96.71	99.58	1058.50	4.00	7.50
2%	mss512, 4kb	96.04	98.07	103.47	1044.00	3.00	8.00
2%	mss512, iw4	96.38	96.85	98.39	1057.00	5.00	8.00
2%	mss512, sack	95.47	96.40	99.18	1062.00	5.00	8.00
5%	mss512	105.01	111.15	124.11	921.00	11.50	21.00
5%	mss512, 4kb	104.82	114.26	126.34	896.50	11.00	21.00
5%	mss512, iw4	101.05	108.59	121.86	943.00	11.50	21.00
5%	mss512, sack	99.18	100.91	103.05	1014.50	11.00	20.50
10%	mss512	151.55	181.01	223.77	566.00	26.00	43.00
10%	mss512, 4kb	145.81	164.22	199.57	623.50	25.50	45.00
10%	mss512, iw4	156.60	186.36	206.37	549.50	26.50	44.00
10%	mss512, sack	108.66	116.70	124.05	877.50	24.00	43.00

Table 20: Delay tests - Baseline TCP, 20 replications. Three different delays are tested:one that does not lead to a RTO, one that triggers a RTO, and one that triggers two RTO consecutive RTOs: Table A

		elapsed time (sec)				
Packet delayed	**Length (ms)**	**25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**
402nd	90000	192.13	192.13	192.14	533.00	2.00
402nd	9000	111.09	111.10	111.11	922.00	1.00
402nd	8000	110.06	110.06	110.07	930.00	0.00
302nd	89000	216.32	216.34	216.35	473.00	102.00
302nd	10000	137.08	137.09	137.10	747.00	101.00
302nd	7500	109.00	109.00	109.01	939.00	0.00
202nd	82000	219.58	219.58	219.59	466.00	143.00
202nd	10600	160.31	160.32	160.34	639.00	191.00
202nd	10000	111.49	111.50	111.51	918.00	0.00
182nd	75000	212.58	212.58	212.59	482.00	143.00
182nd	9500	159.21	159.22	159.23	643.00	191.00
182nd	9000	110.50	110.51	110.52	927.00	0.00
162nd**	139000	276.57	276.58	276.59	370.00	143.00
162nd**	35000	184.71	184.72	184.74	554.00	191.00
162nd**	31000	132.50	132.51	132.51	773.00	0.00
142nd	61000	198.50	198.51	198.52	516.00	143.00
142nd	8600	158.32	158.33	158.34	647.00	191.00
142nd	8000	109.49	109.50	109.50	935.00	0.00
122nd	55000	191.03	191.04	191.05	536.00	137.00
122nd	7500	143.62	154.91	154.91	661.00	182.00
122nd	7000	108.50	108.50	108.51	944.00	0.00
102nd	50000	180.48	180.49	180.49	567.00	115.00
102nd	8000	147.81	147.81	147.82	693.00	152.00
102nd	7500	109.00	109.01	109.01	939.00	0.00
82nd	40000	165.19	165.20	165.21	620.00	94.00
82nd	6400	138.65	138.65	138.66	739.00	122.00
82nd	6000	107.50	107.50	107.51	953.00	0.00
62nd	33000	152.64	152.65	152.66	671.00	72.00
62nd	6000	130.69	130.70	130.71	783.50	92.00
62nd	5600	107.10	107.10	112.96	956.00	0.00

**These values are due to a implentation problem in the RTO calculations. Please, refer to the Appendix C.1.3 for the details.

Table 21: Delay tests - Baseline TCP, 20 replications: table B

Packet delayed	**Length (ms)**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**
42nd	27000	141.10	141.11	141.11	726.00	50.00
42nd	5000	122.13	122.13	122.14	838.00	62.00
42nd	4500	106.00	106.00	106.01	966.00	0.00
32nd	24000	135.34	135.34	135.76	757.00	39.00
32nd	4400	117.75	117.75	117.77	870.00	47.00
32nd	4000	105.50	105.50	105.51	971.00	0.00
22nd	18000	126.56	126.57	126.58	809.00	28.00
22nd	3300	112.87	112.88	112.89	907.00	32.00
22nd	3000	104.50	104.50	104.51	980.00	0.00
17th	11000	118.04	118.05	118.05	867.00	22.00
17th	2500	110.04	110.05	110.06	930.50	24.00
17th	2000	103.50	103.51	103.52	989.00	0.00
12th	7700	113.23	113.24	113.25	904.00	16.00
12th	2000	107.53	107.54	107.54	952.00	16.00
12th	1800	107.33	107.34	107.35	954.00	16.00
9th	9000	113.78	113.78	113.79	900.00	13.00
9th	2800	107.58	107.59	107.60	952.00	13.00
9th	2000	103.50	103.51	103.52	989.00	0.00
7th	7000	109.81	109.82	109.82	932.00	10.00
7th	2600	105.86	105.86	105.87	967.00	7.00
7th	2200	103.70	103.70	105.46	987.00	0.00
5th	13000	116.52	116.52	116.53	879.00	8.00
5th	4000	107.32	107.32	107.33	954.00	7.00
5th	3700	105.46	105.47	105.49	971.00	0.00
3rd	10000	113.06	113.07	113.08	906.00	6.00
3rd	2400	104.83	104.84	104.86	977.00	2.00
3rd	2200	104.25	104.26	104.28	982.00	0.00
1st	10000	112.62	112.62	112.63	909.00	2.00
1st	2500	105.15	105.16	105.18	974.00	1.00
1st	2000	104.05	104.06	104.07	984.00	0.00

Table 22: Delay tests - Limited advertised window of 4KB, 20 replications. The test were run only for the cases where one RTO occurred.

Packet delayed	Length (msec)	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)
402nd	9000	111.09	111.10	111.11	922.00	1.00
302nd	10000	117.80	117.81	117.81	869.00	25.00
202nd	10600	118.40	118.42	118.45	865.00	25.00
182nd	9500	117.30	117.30	117.32	873.00	25.00
162nd**	35000	142.05	142.05	142.06	721.00	22.00
142nd	8600	116.40	116.41	116.42	880.00	25.00
122nd	7500	115.30	115.31	115.33	888.00	25.00
102nd	8000	115.80	115.80	115.81	884.00	25.00
82nd	6400	114.20	114.21	114.22	897.00	25.00
62nd	6000	113.80	113.80	113.81	900.00	25.00
42nd	5000	112.80	112.81	112.82	908.00	25.00
32nd	4400	112.20	112.21	112.21	913.00	25.00
22nd	3300	111.10	111.11	111.11	922.00	25.00
17th	2500	110.05	110.06	110.18	930.00	24.00
12th	1800	107.33	107.33	107.35	954.00	16.00
9th	2800	107.58	107.59	107.60	952.00	13.00
7th	2600	105.86	105.87	105.88	967.00	7.00
5th	4000	107.31	107.32	107.33	954.00	7.00
3rd	2400	104.83	104.84	104.85	977.00	2.00
1st	2500	105.15	105.16	105.18	974.00	1.00

**These values are due to a implentation problem in the RTO calculations. Please, refer to the Appendix C.1.3 for the details.

Table 23: Tests with combined errors and delays on the link, 50 replications. If the delay occurs, its length is 6 seconds.

Drop rate	**Delay Prob**	**Mods**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
2%	1%	Baseline	136.08	143.74	155.67	712.50	56.50	9.00
2%	1%	4kb	125.35	137.52	157.05	744.50	45.00	9.00
2%	1%	iw4	131.84	140.13	153.66	730.50	51.50	9.00
2%	1%	sack	126.12	135.16	147.90	757.50	44.50	9.00
5%	1%	Baseline	150.91	163.79	178.57	625.00	50.00	23.50
5%	1%	4kb	141.83	152.94	167.61	670.00	44.50	21.00
5%	1%	iw4	143.36	158.35	167.52	646.50	47.00	22.50
5%	1%	sack	134.73	143.00	152.55	716.50	46.50	21.00
10%	1%	Baseline	177.91	195.13	216.10	525.00	64.00	45.00
10%	1%	4kb	187.95	204.30	221.76	501.50	65.00	45.00
10%	1%	iw4	179.73	195.50	216.61	523.50	62.50	46.00
10%	1%	sack	142.21	154.35	165.70	663.50	60.00	45.00

Table 24: The results of the static bandwidth tests, 20 replications.

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
1-1-96	Baseline	102.05	102.06	102.06	1003.00	0.00	0.00
1-1-96	iw4	101.85	101.85	101.85	1005.00	0.00	0.00
1-1-96	2kb	102.05	102.07	102.08	1003.00	0.00	0.00
1-1-96	4kb	102.05	102.06	102.07	1003.00	0.00	0.00
1-3-96	Baseline	35.17	35.18	35.19	2911.00	0.00	0.00
1-3-96	iw4	34.78	34.78	34.80	2944.00	0.00	0.00
1-3-96	2kb	35.18	35.19	35.19	2910.00	0.00	0.00
1-3-96	4kb	35.16	35.18	35.18	2911.00	0.00	0.00
2-2-96	Baseline	51.70	51.71	51.72	1980.00	0.00	0.00
2-2-96	iw4	51.38	51.39	51.39	1993.00	0.00	0.00
2-2-96	2kb	51.70	51.71	51.72	1980.00	0.00	0.00
2-2-96	4kb	51.71	51.71	51.72	1980.00	0.00	0.00
2-2-144	Baseline	35.02	35.03	35.04	2923.00	0.00	0.00
2-2-144	iw4	34.67	34.67	34.68	2953.00	0.00	0.00
2-2-144	2kb	35.06	35.07	35.08	2920.00	0.00	0.00
2-2-144	4kb	35.03	35.04	35.04	2923.00	0.00	0.00
144-38	Baseline	26.92	26.93	26.94	3802.00	0.00	0.00
144-38	iw4	26.45	26.45	26.46	3871.00	0.00	0.00
144-38	2kb	29.97	29.98	30.00	3415.50	0.00	0.00
144-38	4kb	26.92	26.93	26.94	3802.50	0.00	0.00

Table 25: The disabled delayed acknowledgments tests, first 2KB. qn stands for disabled delayed acks up to the nth data segment. *Dis.*ACK*s* means, that every segment is separately acknowledged through the connection.

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	2.06	2.06	2.08	1276.53	0.00	0.00
14400/38400	q2	2.12	2.13	2.13	1226.96	0.00	0.00
14400/38400	q6	2.04	2.05	2.06	1288.58	0.00	0.00
14400/38400	q8	2.08	2.09	2.10	1260.74	0.00	0.00
14400/38400	Dis.delacks	2.04	2.05	2.06	1296.63	0.00	0.00

Table 26: The disabled delayed acknowledgments tests. First 5KB

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	2.99	3.01	3.02	2007.71	0.00	0.00
14400/38400	q2	3.06	3.07	3.08	1962.17	0.00	0.00
14400/38400	q6	2.85	2.87	2.87	2125.07	0.00	0.00
14400/38400	q8	2.81	2.82	2.83	2165.46	0.00	0.00
14400/38400	Dis.delacks	2.80	2.81	2.82	2179.37	0.00	0.00

Table 27: The disabled delayed acknowledgments tests. First 10KB

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	4.25	4.27	4.28	2688.12	0.00	0.00
14400/38400	q2	4.32	4.33	4.34	2646.45	0.00	0.00
14400/38400	q6	4.11	4.13	4.13	2790.70	0.00	0.00
14400/38400	q8	4.07	4.08	4.09	2825.27	0.00	0.00
14400/38400	Dis.delacks	4.06	4.07	4.08	2837.09	0.00	0.00

Table 28: The disabled delayed acknowledgments tests. First 20KB

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	6.77	6.79	6.80	3235.74	0.00	0.00
14400/38400	q2	6.84	6.85	6.86	3205.39	0.00	0.00
14400/38400	q6	6.63	6.65	6.65	3308.95	0.00	0.00
14400/38400	q8	6.59	6.60	6.61	3333.01	0.00	0.00
14400/38400	Dis.delacks	6.58	6.59	6.59	3341.33	0.00	0.00

Table 29: The disabled delayed acknowledgments tests. First 50KB

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	14.33	14.35	14.36	3686.34	0.00	0.00
14400/38400	q2	14.40	14.41	14.42	3670.50	0.00	0.00
14400/38400	q6	14.19	14.21	14.21	3723.87	0.00	0.00
14400/38400	q8	14.15	14.16	14.17	3736.10	0.00	0.00
14400/38400	Dis.delacks	14.14	14.15	14.15	3740.18	0.00	0.00

Table 30: The disabled delayed acknowledgments tests: full connection (100KB), 20 replications

Bw (up/down)	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
14400/38400	Baseline (q4)	26.92	26.93	26.94	3802.00	0.00	0.00
14400/38400	q2	27.03	27.04	27.05	3788.00	0.00	0.00
14400/38400	q6	26.82	26.84	26.84	3815.00	0.00	0.00
14400/38400	q8	26.78	26.79	26.80	3822.00	0.00	0.00
14400/38400	Dis.delacks	26.77	26.78	26.78	3823.00	0.00	0.00

Table 31: The results of variable bandwidth tests: 20 replications. Notice that the time of the rate change was measured from the beginning of the *emulation*. As the emulation begins before the connection is eshtablished, the actual time is a little shorter when measuring from the beginning of the connection.

Bw1	**Bw2**	**Rate changed (sec)**	**Mod**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
2_96	1_96	10s	Baseline	97.00	97.01	97.14	1056.00	0.00	0.00
2_96	1_96	10s	4kb	96.99	97.01	97.14	1056.00	0.00	0.00
2_96	1_96	10s	iw4	96.42	96.43	96.43	1062.00	0.00	0.00
2_96	1_96	25s	Baseline	81.99	82.00	82.03	1249.00	0.00	0.00
2_96	1_96	25s	4kb	81.99	82.00	82.02	1249.00	0.00	0.00
2_96	1_96	25s	iw4	81.42	81.42	81.44	1258.00	0.00	0.00
1_144	1_96	10s	Baseline	99.47	99.48	99.49	1029.00	0.00	0.00
1_144	1_96	10s	4kb	99.47	99.47	99.49	1029.00	0.00	0.00
1_144	1_96	10s	iw4	99.03	99.04	99.04	1034.00	0.00	0.00
1_144	1_96	30s	Baseline	89.47	89.47	89.48	1144.50	0.00	0.00
1_144	1_96	30s	4kb	89.47	89.48	89.49	1144.00	0.00	0.00
1_144	1_96	30s	iw4	89.03	89.04	89.05	1150.00	0.00	0.00
1_3-96	1_1-96	10s	Baseline	92.64	92.65	92.69	1105.00	0.00	0.00
1_3-96	1_1-96	10s	4kb	92.64	92.65	92.66	1105.00	0.00	0.00
1_3-96	1_1-96	10s	iw4	91.41	91.42	91.60	1120.00	0.00	0.00
1_3_96	1_1-96	25s	Baseline	62.57	62.74	62.76	1632.00	0.00	0.00
1_3_96	1_1-96	25s	4kb	62.57	62.74	62.75	1632.00	0.00	0.00
1_3_96	1_1-96	25s	iw4	61.50	61.51	61.52	1665.00	0.00	0.00
2_144	1_144	10s	Baseline	63.70	63.71	63.73	1607.00	0.00	0.00
2_144	1_144	10s	4kb	63.71	63.72	63.72	1607.00	0.00	0.00
2_144	1_144	10s	iw4	63.01	63.01	63.03	1625.00	0.00	0.00
2_144	1_144	27s	Baseline	46.74	46.74	46.74	2191.00	0.00	0.00
2_144	1_144	27s	4kb	46.74	46.74	46.75	2191.00	0.00	0.00
2_144	1_144	27s	iw4	45.96	45.96	45.99	2228.00	0.00	0.00
2_144	2_96	10s	Baseline	49.36	49.37	49.38	2074.00	0.00	0.00
2_144	2_96	10s	4kb	49.36	49.37	49.38	2074.00	0.00	0.00
2_144	2_96	10s	iw4	48.83	48.84	48.85	2097.00	0.00	0.00
2_144	2_96	27s	Baseline	40.83	40.89	40.90	2504.00	0.00	0.00
2_144	2_96	27s	4kb	40.87	40.88	40.89	2505.00	0.00	0.00
2_144	2-96	27s	iw4	40.31	40.31	40.38	2540.00	0.00	0.00

Table 32: Tests with different bandwidths and random packet drops, 50 replications: table A

Bw (up/down)	**Drop Rate**	**Mod**	**elapsed time (sec) 25 %**	**50 %**	**75 %**	**tput (Bytes/s)**	**rexmt (Pkts)**	**drops (Pkts)**
1-3-96	2%	iw4	39.27	42.20	44.35	2426.50	8.00	8.00
1-3-96	2%	2kb	47.10	49.66	52.50	2062.00	9.00	9.00
1-3-96	2%	4kb	41.80	44.40	47.30	2306.00	8.00	8.00
1-3-96	2%	sack	37.43	39.30	42.43	2605.50	7.00	7.00
1-3-96	2%	Baseline	37.22	42.00	45.70	2438.00	8.00	8.00
1-3-96	5%	iw4	54.69	60.43	65.26	1694.50	21.50	20.00
1-3-96	5%	2kb	61.60	66.25	71.11	1545.50	23.00	22.00
1-3-96	5%	4kb	57.49	62.83	72.23	1630.00	22.00	22.00
1-3-96	5%	sack	52.09	55.84	59.99	1833.50	22.00	21.00
1-3-96	5%	Baseline	54.89	61.11	68.15	1676.00	22.00	21.00
1-3-96	10%	iw4	93.06	100.94	112.19	1014.50	50.00	46.50
1-3-96	10%	2kb	97.04	105.55	116.69	970.50	50.00	46.00
1-3-96	10%	4kb	92.02	99.23	108.75	1032.50	47.00	43.50
1-3-96	10%	sack	70.01	75.10	79.89	1363.50	47.00	45.00
1-3-96	10%	Baseline	91.78	100.15	110.58	1022.50	47.00	44.00

Table 33: Tests with different bandwidths and random packet drops, 50 replications: table B

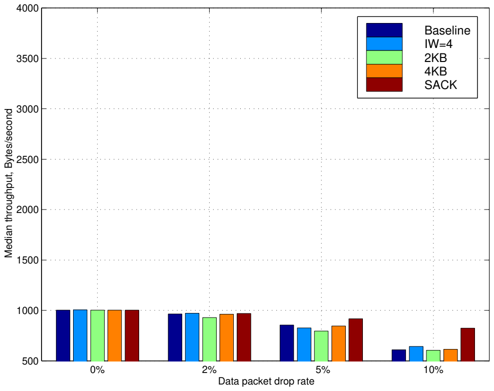
Bw (up/down)	Drop Rate	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
144-38	2%	iw4	31.71	33.45	38.47	3061.50	8.00	8.00
144-38	2%	2kb	40.27	43.15	46.72	2373.50	7.50	7.50
144-38	2%	4kb	35.36	38.52	41.82	2659.00	8.00	8.00
144-38	2%	sack	30.78	34.87	38.35	2936.50	8.00	8.00
144-38	2%	Baseline	32.46	36.62	39.27	2796.50	8.00	8.00
144-38	5%	iw4	49.26	55.92	60.29	1831.50	22.00	22.00
144-38	5%	2kb	55.53	61.73	66.67	1659.00	23.00	22.00
144-38	5%	4kb	52.27	57.22	62.19	1790.00	20.50	20.50
144-38	5%	sack	47.68	51.00	54.56	2008.00	22.00	21.50
144-38	5%	Baseline	49.84	57.37	63.09	1785.00	22.00	21.00
144-38	10%	iw4	87.32	94.07	103.17	1088.50	49.00	46.00
144-38	10%	2kb	85.72	97.24	105.88	1053.00	46.00	43.50
144-38	10%	4kb	88.13	94.52	104.20	1083.50	49.00	45.00
144-38	10%	sack	65.49	69.08	74.15	1482.00	47.00	46.00
144-38	10%	Baseline	82.57	90.66	103.00	1129.50	46.00	42.00

Table 34: Tests with different bandwidths and random packet drops, 50 replications: table C

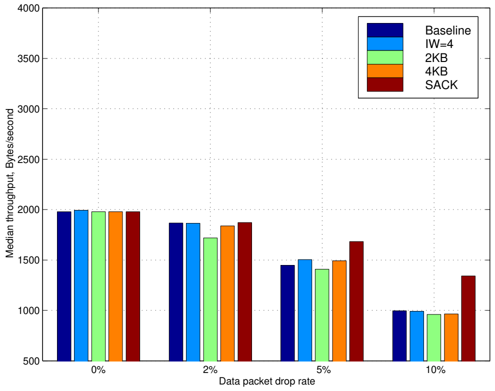
Bw (up/down)	Drop Rate	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
2-2-144	2%	iw4	37.43	40.35	43.34	2538.00	9.00	9.00
2-2-144	2%	2kb	44.19	47.48	51.32	2156.50	9.00	9.00
2-2-144	2%	4kb	39.72	41.02	44.97	2496.50	7.00	7.00
2-2-144	2%	sack	37.13	39.74	41.29	2576.50	9.00	8.50
2-2-144	2%	Baseline	38.88	41.70	45.05	2455.50	8.50	8.00
2-2-144	5%	iw4	53.29	57.31	62.57	1786.50	22.00	21.50
2-2-144	5%	2kb	60.03	64.18	67.70	1595.50	23.00	22.00
2-2-144	5%	4kb	53.85	59.66	64.91	1716.50	22.00	21.00
2-2-144	5%	sack	48.58	53.40	56.00	1917.50	24.00	22.50
2-2-144	5%	Baseline	55.78	59.91	66.52	1709.50	23.00	22.00
2-2-144	10%	iw4	88.31	96.97	102.98	1056.00	49.00	46.00
2-2-144	10%	2kb	88.62	99.75	107.74	1026.50	51.00	48.00
2-2-144	10%	4kb	86.24	93.88	100.26	1090.50	47.00	42.00
2-2-144	10%	sack	63.77	68.26	74.24	1500.00	48.50	46.00
2-2-144	10%	Baseline	82.92	89.75	100.27	1141.00	46.50	42.00

Table 35: Tests with different bandwidths and random packet drops, 50 replications: table D

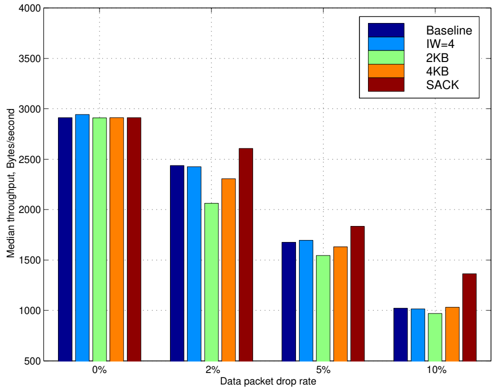
Bw (up/down)	Drop Rate	Mod	elapsed time (sec) 25 %	50 %	75 %	tput (Bytes/s)	rexmt (Pkts)	drops (Pkts)
2-2-96	2%	iw4	53.71	54.89	56.76	1865.50	10.00	9.00
2-2-96	2%	2kb	57.06	59.59	62.59	1718.50	8.00	8.00
2-2-96	2%	4kb	54.43	55.71	57.47	1838.00	8.00	8.00
2-2-96	2%	sack	53.37	54.69	56.05	1872.50	11.00	8.00
2-2-96	2%	Baseline	53.19	54.82	57.86	1867.50	10.00	8.00
2-2-96	5%	iw4	62.39	68.06	72.75	1504.50	21.00	20.50
2-2-96	5%	2kb	68.57	72.65	76.81	1409.50	21.00	20.00
2-2-96	5%	4kb	64.57	68.61	74.33	1492.50	22.00	20.50
2-2-96	5%	sack	59.37	60.86	63.99	1682.50	23.00	21.00
2-2-96	5%	Baseline	64.71	70.74	76.84	1448.00	23.50	22.00
2-2-96	10%	iw4	97.19	103.21	116.37	992.00	48.00	44.00
2-2-96	10%	2kb	97.32	106.69	116.37	960.00	50.00	46.00
2-2-96	10%	4kb	97.66	106.17	111.93	964.50	48.50	45.50
2-2-96	10%	sack	72.52	76.23	81.11	1343.00	48.00	44.50
2-2-96	10%	Baseline	95.91	102.87	111.43	995.50	47.00	43.00



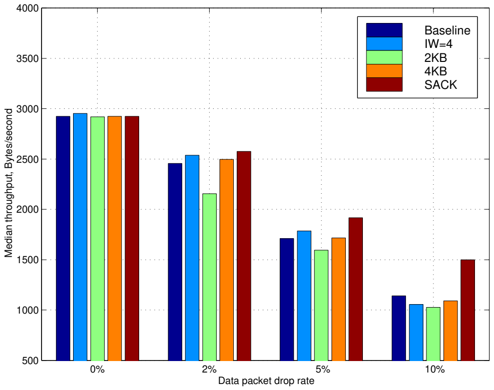
(a) 9600bps (1200Bps)



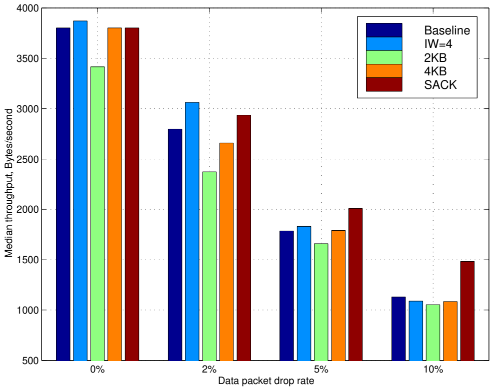
(b) 19200bps (2400Bps)



(c) 9600bps/28800bps(1200Bps/3600Bps)



(d) 28800bps (3600Bps)



(e) 14400bps/38400bps(1800Bps/4800Bps)

Figure 33: The throughputs of every tested TCP enhancement over links with various characteristics

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 [Pos81b]. 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 34 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 34 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.

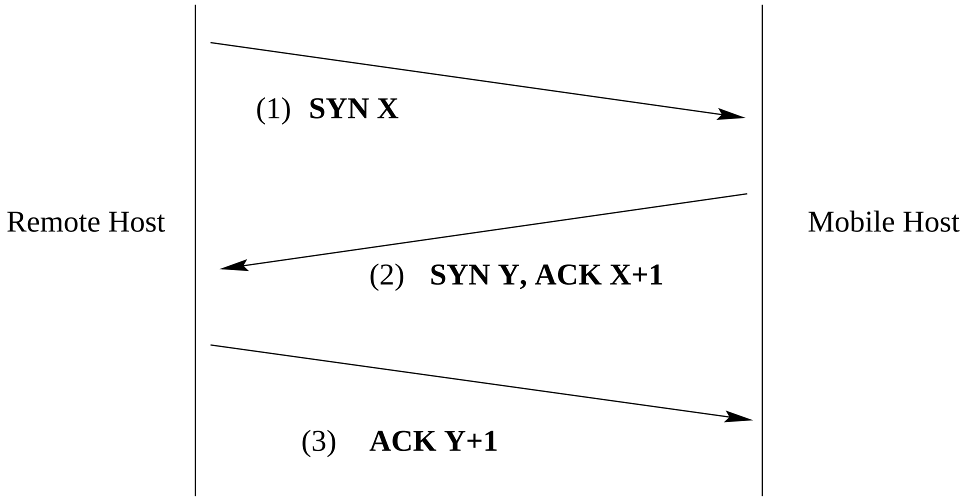


Figure 34: The three-way handshake at the beginning of the connection

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 34). 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 [Pos81b] 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[27]. 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.

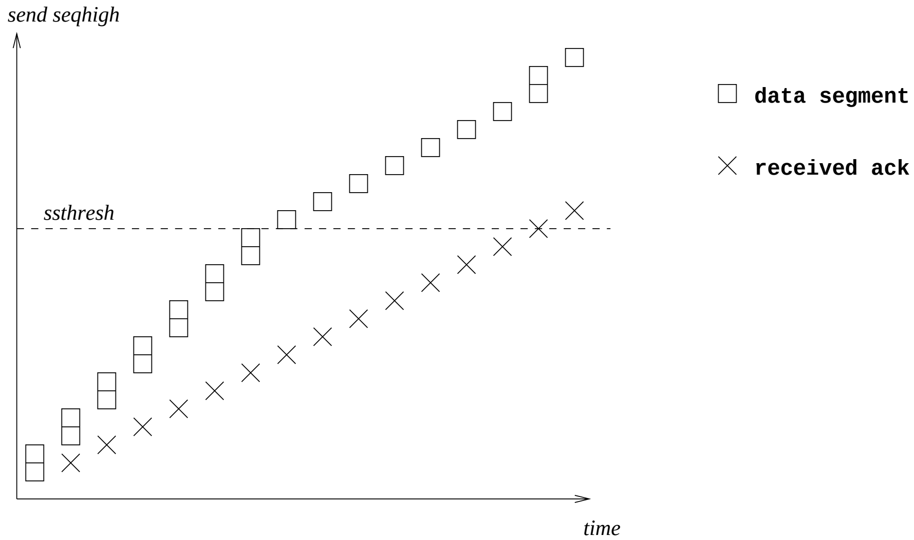


Figure 35: A trace of slow start with the IW size of two segments, without delayed acknowledgments.

In Figure 36 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

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

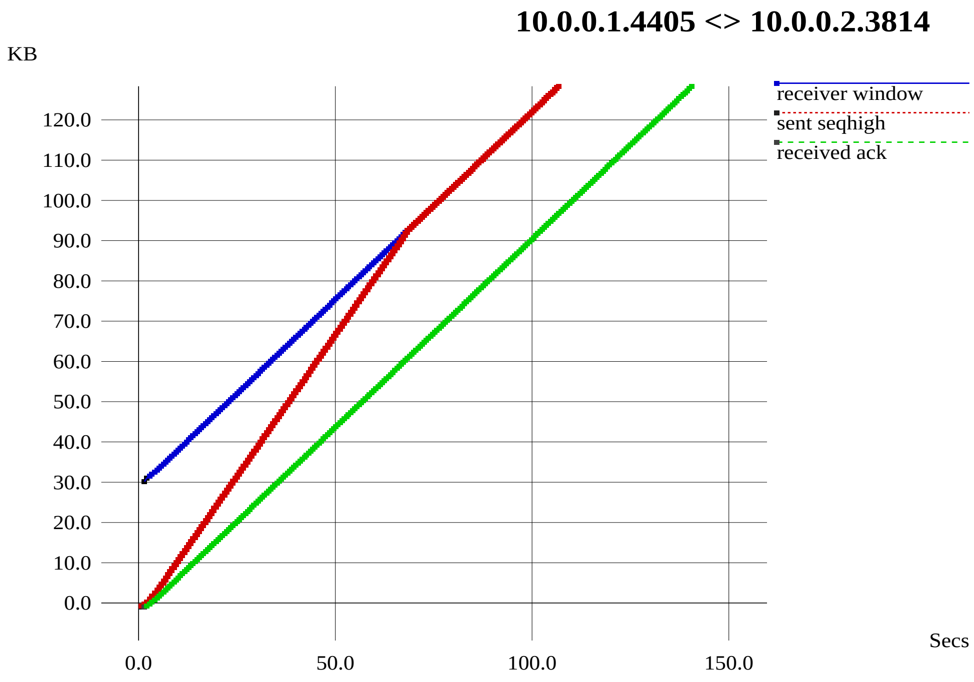


Figure 36: A 130KB transfer

congestion is detected by RTO. A commonly used formula to update `cwnd` is [APS99]:

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

Figure 35 shows this situation as the TCP sender sends two new segments the the networks after receiving an `ACK`. If *delayed acknowledgments* 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 37 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.

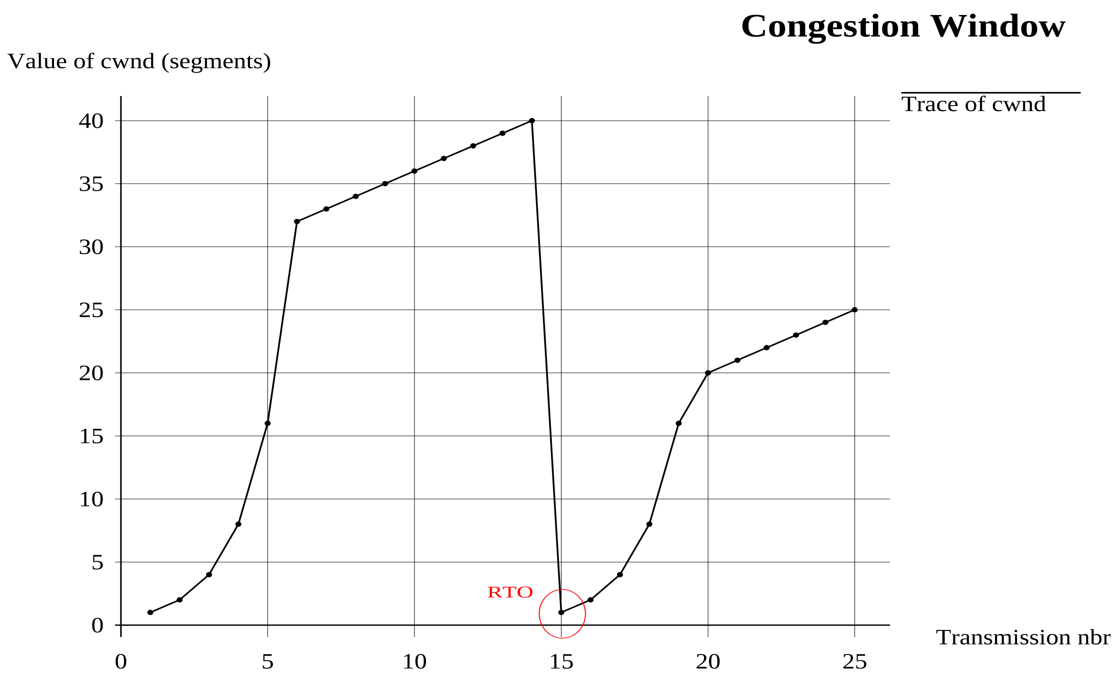


Figure 37: The trace of cwnd after a retransmission timeout

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 [Pax97]. 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)[28]. 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.

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

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

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 38 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 ACKs, are printed. The figure is drawn from the sender's TCP dump[29] 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*.

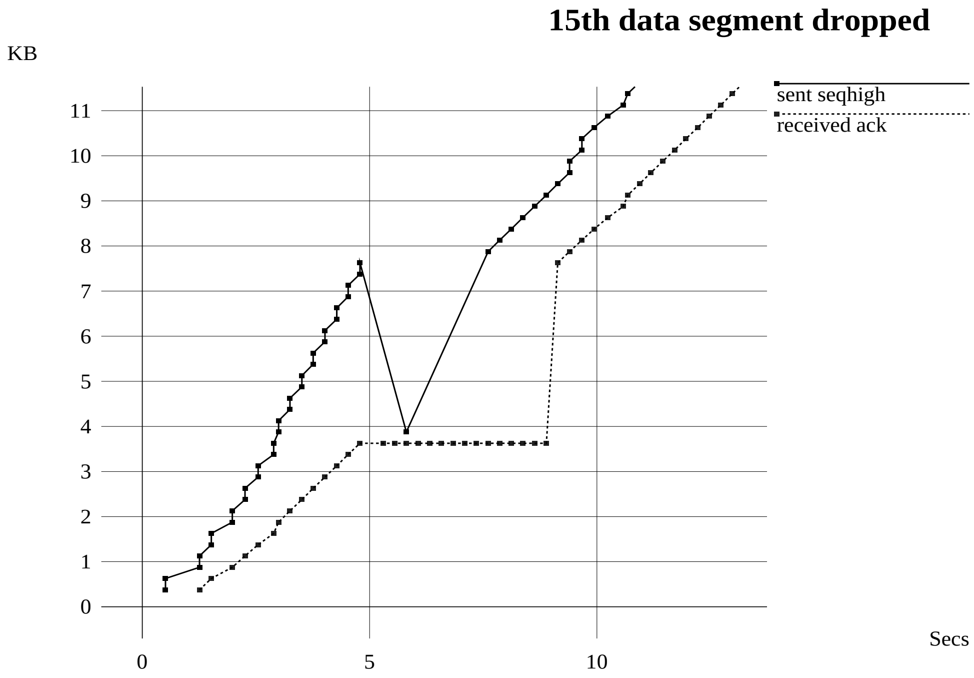


Figure 38: Recovery from a missing segment using the fast retransmit/fast recovery algorithm

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

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.*

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 111. 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* (the value in step 5) + 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 [Pos81b], [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[30].

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 `ACK`s 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[31]. The exact behavior is explained in Section B.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 consecutive data segments of size 296 bytes is more than 200ms. Therefore, in most of

[30] 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.

[31] We call it BSD style, because Baseline TCP imitates the behavior that was used in the 4.4BSD-Lite version, checked from [WS95].

our tests each data segment is acknowledged separately. When using higher bandwidths, two segments are "quickacked" [32] 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 36 summarizes the algorithms, parameters and their values used in Baseline TCP.

Table 36: 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

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

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.

- **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 [Pos81b]. By advertising a smaller window the receiver can control the number of segments that the sender is allowed to transmit. The basic analysis of the effect of the receiver window on a protocol performance can be found e.g. in [Sta00].

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

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

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 are

```
for each packet arrival
   calculate the moving average of the queue size avg
   if min_th ≤ avg < max_th
      calculate probability p_a
      with probability p_a:
         drop the arriving packet
   else if max_th ≤ avg
      drop the arriving packet
```

Figure 39: The general algorithm of the Random Early Detection (RED).

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 40. 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.

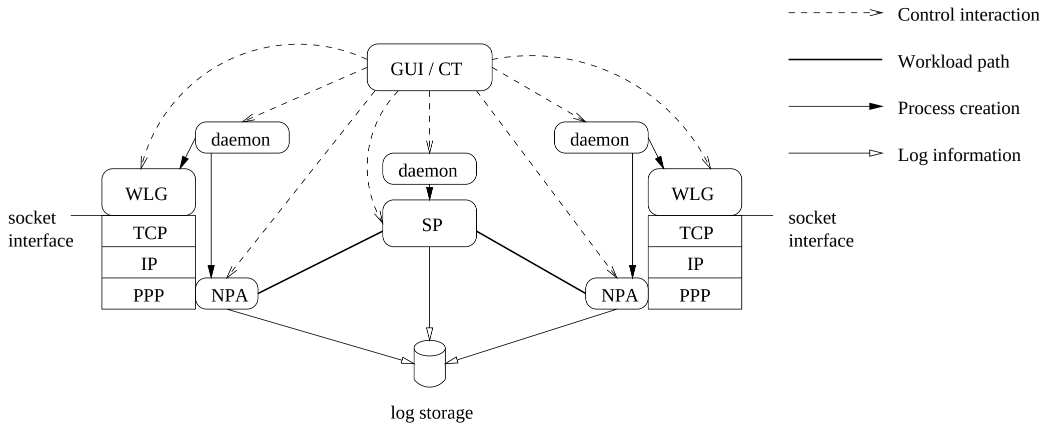


Figure 40: Architecture of the Seawind emulator.

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 41 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.

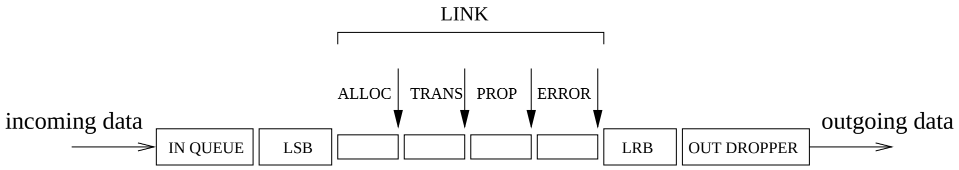


Figure 41: Ordering of events for a packet in a simulation process.

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 41). 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 41). 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 41 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 37. 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.

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

Table 37: 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

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.