Faculty of Economic Sciences, Communication and IT
Computer Science

Per Hurtig, Johan Garcia and Anna Brunstrom

Loss Recovery in Short TCP/SCTP Flows

Karlstad University Studies
2006:71

Per Hurtig, Johan Garcia and Anna Brunstrom

Loss Recovery in Short TCP/SCTP Flows

Karlstad University Studies
2006:71

Per Hurtig, Johan Garcia and Anna Brunstrom.
Loss Recovery in Short TCP/SCTP Flows

Research Report

Karlstad University Studies 2006:71
ISSN 1403-8099
ISBN 91-7063-101-8

Distribution:
Karlstad University
Faculty of Economic Sciences, Communication and IT
Computer Science
SE-651 88 KARLSTAD
SWEDEN
+46 54-700 10 00

www.kau.se

Printed at: Universitetstryckeriet, Karlstad 2006

Loss Recovery in Short TCP/SCTP Flows

Per Hurtig, Johan Garcia and Anna Brunstrom

December 8, 2006

Abstract

The Transmission Control Protocol (TCP) has been the dominant transport protocol within IP-based networks for many years, mainly due to the reliability it provide to its users and the congestion control it employs. However, as the amount of signaling traffic within IP-based networks have increased significantly in recent years, it has become clear that TCP is not suited for this kind of traffic. In order to meet the requirements of signaling traffic the Stream Control Transmission Protocol (SCTP) was developed by the Internet Engineering Task Force (IETF). SCTP is heavily influenced by TCP and is therefore similar to TCP in many ways. One example is the SCTP loss recovery and congestion control mechanisms which are almost identical to those of TCP. The primary purpose of this work is to study the performance and behavior of the TCP/SCTP loss recovery mechanisms for short flows. Using a simple client/server model, we evaluate the performance of these mechanism over a wide range of bandwidths, link delays and packet loss patterns. The experiments evaluate one TCP implementation and two SCTP implementations, and were conducted using network emulation. The experimental results show that there exist strong dependencies between the position of packet loss and the actual transmission time of the corresponding flow. In addition to these dependencies, we also found a number of implementation mistakes in the examined protocol implementations.

Contents

1 Introduction

The Transmission Control Protocol (TCP) [15] has for many years been the predominant transport protocol within the Internet. It is currently used by a vast range of applications including web browsers, file sharing applications, and online computer games. Because of the massive growth of the Internet and the evolution of networking techniques such as mobile communication, the demands that a transport protocol like TCP is facing have become more and more complex. In order to meet these demands a lot of research and refinements have been done on TCP, making it one of the most complex transport protocols of today. In addition to these refinements, new transport protocols have also been designed and implemented to better meet the new demands that different network applications have. One such protocol is the Stream Control Transmission Protocol (SCTP) [21] which was designed to better support the transport of PSTN signaling traffic. SCTP is based on TCP and is therefore, of course, related to TCP in many different ways.

One commonality between TCP and SCTP is that they both try to prevent that the network they operate in becomes congested. Network congestion is a phenomenon that was discovered in the late 80's when the Internet began to grow large. Basically, the problem of congestion occurs when parts of the network infrastructure become over-utilized by large amounts of traffic and, therefore, are unable to deliver any data to its users. In both TCP and SCTP the mechanisms that try to prevent congestion are coupled with the mechanisms that are used to ensure that no data is lost in the communication process, the so called reliability mechanisms. This is done because packet loss historically has been seen as a sign of network congestion, and it is therefore considered natural that these mechanisms work together. One interesting aspect of the loss recovery mechanisms employed by both TCP and SCTP is that the transmission time of a flow is affected differently depending on which packet that is lost. For example, a loss of the first packet in a TCP connection (the `SYN` segment) will add three seconds to the transmission time of that flow, which very well may be a large portion of the total transmission time if the flow that we consider is short-lived (e.g. a HTTP request).

The primary goal of the work that is described in this technical report was to investigate what effect the existence and position of a single packet loss, within a short TCP or SCTP flow, has on the total transmission time of the flow. This was done by performing a large series of experiments. To be able to conduct these experiments an experimental environment consisting of a client and a server was used. The communication between these two hosts was routed over a network emulator that was equipped with an extended version of the Dummynet network emulation software. This extension, called KauNet, provides deterministic packet loss capabilities, i.e. the ability to lose packets according to their position in a flow.

The rest of this report is structured as follows. Section 2 provides some background information related to TCP and SCTP. Furthermore, it also contains a basic description of network emulation, the Dummynet network emulation soft-

ware, and the KauNet extension that was used for the experiments described in this report. Section 3 describes shortly how the total transmission time of a TCP or SCTP flow can depend on the actual position of a single packet loss. In Section 4 the experiments that were conducted are described more closely and a description of the experimental environment that was used is also provided. In Sections 5 and 6, a representative selection of the experimental results for TCP and SCTP are presented and explained. Section 7 contains discussions of results that are of particular interest. Finally, the report is concluded with a section that presents possible future work, in Section 8, followed by a summary in Section 9.

In addition to this, a number of appendices are also available. In Appendix A we describe the basics of the congestion control mechanisms that both TCP and SCTP incorporates. Appendix B contains a full listing of all the experiments that have been conducted during the work with this report. Finally, in appendices C and D we describe some important aspects of the TCP and SCTP implementations that were used for the experiments.

2 Background

2.1 TCP & SCTP

The Transmission Control Protocol (TCP) [15] is the most used transport protocol in the Internet today. It is part of the TCP/IP protocol suite which allows computers, regardless of operating system and hardware, to communicate with each other. One of the major properties of TCP is that it is able to provide a connection-oriented data transfer service that also provides reliability to applications who require that no data is lost and/or damaged in the communication process. TCP is used in conjunction with the Internet Protocol (IP) [14] which only provides an unreliable connection-less data transfer service between different hosts. To be able to provide connection-oriented reliable communication, TCP implements a number of mechanisms on top of the simple datagram service that IP offers. These mechanisms are designed to overcome the shortcomings of the simple IP service by retransmitting lost packets, order segments that have been reordered by the network, and regulate the data transfer rate in order to prevent network congestion.

The Stream Control Transmission Protocol [21] (SCTP), on the other hand, is a relatively new protocol that was designed in order to support the transmission of PSTN signaling messages over IP better than TCP manages. Like TCP, SCTP also provides a reliable connection-oriented service to its users, but in order to give better support for PSTN signaling, SCTP has been designed to ameliorate the following shortcomings of TCP:

- Vulnerability to denial of service attacks.
- The byte-oriented transfer service that forces TCP applications to implement their own, possibly complex, record markings.
- The requirement that data must always be delivered in a strictly ordered fashion.
- The problem of availability; in order to increase the chance of connectivity, SCTP incorporates multi-homing features.

Despite the differences mentioned above, TCP and SCTP have very much in common. One example of this is the mechanisms for loss recovery, which are almost identical. Both protocols employ two different mechanisms to detect and recover from packet loss:

1. If an acknowledgment for a transmitted packet is not received within a certain time frame, a Retransmission Time-Out (RTO) occurs and the lost packet is retransmitted.
2. If a certain number of duplicate acknowledgments arrives at the sender, the corresponding packet is retransmitted. For TCP, retransmission is

done after receiving three duplicate acknowledgments, and for SCTP this threshold is set to four[1].

Of these two mechanisms the latter, called fast retransmit, is the preferred one for two different reasons. Firstly, the loss detection is faster as the sender does not have to wait for the RTO timer to expire. Secondly, the congestion control mechanism that is invoked upon fast retransmit allows for higher throughput. The advantage of fast retransmit is especially beneficial for short-lived flows, as the time needed for the RTO timer to expire can be a relatively large portion of the total transmission time of such a flow.

The reader of this report is assumed to have basic knowledge about TCP and SCTP. However, in Appendix A we provide a short introduction to the congestion control mechanisms that are common for both protocols.

2.2 Network Emulation

Because of the increase in complexity of todays networks, especially the Internet, the transport protocols that are in use have also become more and more complex. For example, the SCTP specification [21, 23] which is standardized by the Internet Engineering Task Force (IETF) consists of a total of 151 pages and a SCTP Errata and Issues document [20] of 109 pages. Furthermore, one of the most popular implementations of this SCTP specification, LKSCTP [5], consists of 28612 lines of source code[2]. Thus, the complexity of such transport protocols makes it hard, if possible at all, to fully grasp the behavior and performance issues without evaluating them under controlled conditions. Fortunately, there exist several evaluation methods that are able to provide control of the conditions that a protocol is working under, making it easier to evaluate the protocol. These methods include mathematical analysis, simulation experiments, and experiments with real protocol implementations. Even though analysis and simulation can be used with advantage, due to the fact that a high level of control is provided, evaluation with real protocol implementations is often desired. This is desired because real protocol implementations more accurately reflect how transport protocols that are used in real operational networks behave. In order to perform evaluations in this fashion, and still have control of the environment, network emulation can be used.

Network emulation can be seen as a cross between simulation and reality; in simulation all entities are based on models, and in reality all entities are real. When using network emulation as the evaluation method, however, the network and its entities is often simulated while the end hosts are using real protocol implementations, and real network applications. By employing network emulation and, thus, simulating the network it is possible to control network

[1] Actually, the SCTP Errata and Issues document [20] suggests that this threshold should be lowered to three for SCTP as well. But unfortunately this specification is ambiguous as it at another place in the text clearly states that a threshold of four should be used. As this ambiguity exists we will consider the "correct" threshold to be four throughout this report.

[2] As of 2006-10-13

parameters such as end-to-end delays, queue sizes, bandwidths, and packet loss probabilities. This is something that is rather hard, if possible at all, if one uses a real operational network as a test-bed. Furthermore, because these network parameters are controllable it is much easier to do repeatable evaluations. It is just a matter of using identical parameter sets. Figure 1 shows an example of network emulation; two "real" hosts communicate with each other over a machine that emulates the behavior of a complete network.

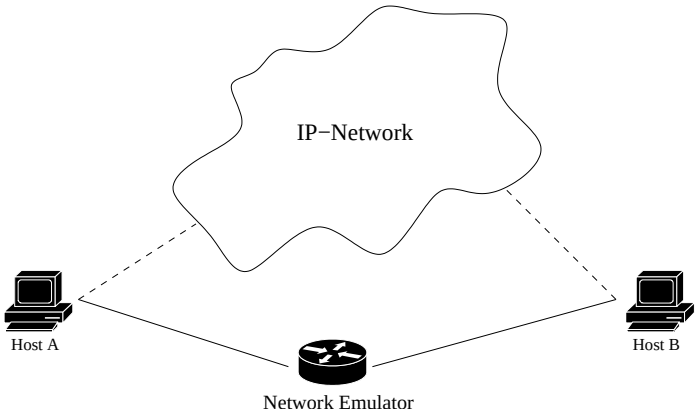


Figure 1: Network Emulation

There are some drawbacks of using network emulation though. Firstly, it is very hard, or at least expensive, to make large scale experiments that require many hosts. Secondly, since experiments are conducted using a specific implementation of a protocol, the results from these experiments may lack in terms of generality to some extent. This occurs because different implementations may incorporate different, non-standardized, functionalities into their implementations in order to do various enhancements/optimizations.

The next section describes the network emulator, called KauNet, that was used for the work described in this technical report.

2.3 KauNet: A Dummynet Extension

Dummynet [18, 19] is a network emulation software that is included in the FreeBSD kernel [7]. Dummynet works by intercepting the network communication within the FreeBSD protocol stack and applying user defined emulation effects on selected traffic. The possible emulation effects that a user can use include different bandwidth and queue limitations, delays, packet loss probabilities, and reordering effects.

To be able to configure Dummynet for network emulation, the FreeBSD firewall is used. By creating a so called Dummynet pipe with the firewall configuration program, a user can force certain types of traffic into that pipe and apply emulation effects on it. An example of such a pipe is shown in Figure 2.

This example pipe is configured to contain all TCP traffic from IP address 10.0.1.1. Furthermore, this pipe is configured to apply 10ms of delay to each packet passing through the pipe as well as a 5% packet loss probability. In order to create this pipe the FreeBSD firewall configuration program, `ipfw`, was used as shown:

```
ipfw add 1 pipe 100 tcp from 10.0.1.1 in
ipfw pipe 100 config delay 10ms plr 0.05
```

Figure 2: Example of a Dummynet pipe

KauNet is an extension to Dummynet that adds better control of emulation effects. The main contribution of KauNet is that it is able to provide facilities for deterministic packet loss, bit errors, and delay/bandwidth management by the use of user specified patterns. Let us say that we want to transmit a SCTP flow from one host to another. In addition to this we also want the third packet in this flow to be discarded by the network emulator for some reason. If our network emulator had been equipped with an ordinary version of Dummynet this task would have been hard, if possible at all. As mentioned earlier, Dummynet is only able to perform packet drops according to a user specified packet loss probability. With the KauNet extension, however, this task is very simple. By using the pattern generation tool that is bundled with KauNet one can create patterns that specify exactly which packets should be lost. In Figure 3 a packet loss pattern is shown. When using such a pattern KauNet will match incoming packets against the values in the pattern. In this example the pattern defines that the third incoming packet should be discarded by KauNet.

0	0	1	0	0	0	. . .	0

Figure 3: Packet loss pattern

By providing this extra functionality, a user can gain a high level of control over his/her emulation scenarios. In addition to this, all scenarios are repeatable. It is only a matter of saving the different patterns. As mentioned previously, KauNet does not only provide deterministic packet loss. It is also able to handle deterministic bit error patterns, delay change patterns, and bandwidth change patterns.

3 Packet Loss and Positional Dependencies

As mentioned previously both TCP and SCTP use two different loss recovery mechanisms. Of these two the fast retransmit mechanism is the preferred one, as it allows for faster loss detection and higher throughput. To be able to trigger this mechanism a TCP sender is required to receive three consecutive duplicate acknowledgments, and a SCTP sender four. However, while the use of this threshold helps to avoid spurious retransmissions if network reordering occurs, it can also inhibit the fast retransmit mechanism in a number of situations. For example, when:

- the sender is limited by the network application. This situation occurs whenever the application does not produce enough data to trigger a sufficient amount of duplicate acknowledgments at the receiver.
- the sender is limited by the receivers advertised window.

In an earlier study [8] a particular case that is related to the first mentioned situation was investigated for TCP; in the end of a connection there are, by definition, never enough data in order to trigger the fast retransmit mechanism. It can therefore be costly, especially for short-lived flows, if packet loss occurs late in the flow and the sender is forced to rely solely on the expiration of the RTO timer. The performance degradation that occurred for flows that experienced packet loss late in the flow, as opposed to flows with packet loss earlier in the flow where fast retransmit could be employed, was therefore evaluated using the KauNet network emulation software. However, when this work was conducted several other phenomena were also discovered, phenomena that only occurred for flows with packet loss at certain positions within a flow. These phenomena were related to the performance as well as to the behavior of the TCP implementation that was used. Because of this dependency between packet loss position and protocol behavior it seemed natural to use deterministicly positioned packet loss as a transport protocol evaluation method. Because of the previously mentioned relationship between the loss recovery mechanisms of TCP and SCTP, this evaluation method seems to be appropriate to use on SCTP implementations as well. Due to the fact that SCTP is a relatively new transport protocol, and therefore not evaluated as extensively as TCP this evaluation method may help in finding implementation mistakes and comparing the effects of different design choices that are adopted in different implementations of the protocol. Thus, the intention of the study that is described in this report was to more thoroughly examine the dependencies that exists between the position of a single packet loss, within TCP and SCTP flows, and the transmission time of the flow.

4 Experimental Design & Setup

4.1 Experimental Overview

In order to determine what effect the position of a single packet loss within a flow, poses on the total transmission time of the flow, a simple experimental design was used. Figure 4 shows the design along with the actual setup of the experimental environment. The design, and setup, was used for the TCP as well as for the SCTP experiments. Let us first consider the design of the experiments. Firstly, the client initiated a connection with the server, which in turn sent a fixed number of data packets back to the client and then terminated the connection. All together 20 packets were sent from the server to the client, including packets used for connection establishment and termination[3]. The time that was required for the transmission of that flow was then logged by the client. In the first experiment run no packets were lost, this in order to get a reference on how fast the transmission was without any loss. In the second run the experimental steps ($1-3$ in Figure 4) were repeated, but this time the first packet from the server to the client was lost. This behavior ($1-3$) was then repeated until a packet had been lost, individually, on each of the 20 positions. This was done in order to reveal how the total transmission time was affected by the placement of the loss.

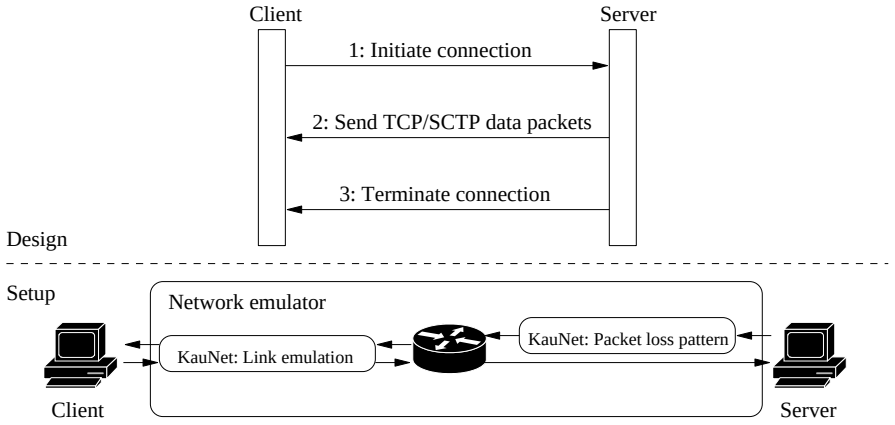


Figure 4: Experimental setup

To be able to conduct the actual experiments, the setup and configuration of the environment, shown in Figure 4, was done according to:

Client: The client host was equipped with both FreeBSD 6.0 and a Linux distribution running the 2.6.15 kernel. This was done to make it possible to experiment with different implementations of TCP and SCTP. The test

[3]Please note that 20 packets is the original length of the flow. If packets are lost during the transfer the actual length of the flow will be larger due to retransmissions.

application that ran on the client machine was constructed to initiate a connection with the server and then, after successfully receiving the data sent from the server, report the time that was required.

Server: Like the client, the server was also equipped with both FreeBSD and Linux. The test application that was on this machine was constructed to accept incoming connections, send a fixed amount of data to the connecting client and then terminate the connection.

Network Emulator: The emulator shown in the figure above is an ordinary host running FreeBSD 6.0. This host was equipped with the KauNet network emulator (see Section 2.3) in order to enable the deterministic packet loss capabilities that were required for the experiments.

As mentioned earlier we used a flow length of 20 packets for all experiments that we conducted. Furthermore, the experiments were run for a large set of different bandwidths and delays. Every experiment was also replicated three times in order to account for possible variance[4]. For a complete list of the different bandwidths and delays that were used, please see Appendix B. In addition to this, a number of TCP and SCTP specific parameters were also evaluated during the experiments. These parameters are discussed in more detail in the following two subsections.

4.2 TCP Experiments

As mentioned earlier, both FreeBSD and Linux were used in our experimental test-bed in order to give us the opportunity to compare different implementations of TCP and SCTP. However, as the results from the experiments with different TCP implementations showed to be very similar we have chosen to only present the results for the FreeBSD 6.0 TCP implementation [7]. However, a complete listing of all the TCP experiments that were conducted is available in Appendix B.

Below is a list of the different test cases that were considered for FreeBSD TCP. The first test case described in this list is the experimental baseline, which all subsequent test cases are compared against. The default parameterization of the TCP implementation in FreeBSD, and a number of interesting implementation details are described in Appendix C.1[5].

Experimental baseline: The FreeBSD TCP implementation was used with its original settings, with one exception. This exception was the FreeBSD TCP feature that is called "Inflight bandwidth limiting", which is enabled by default. When this feature is enabled TCP tries to calculate the bandwidth-delay product of the connection, and use this estimation

[4] As the experimental results did not indicate any significant variance between the different replications, we have chosen to only present the median values throughout this report.

[5] In addition to this, the default parameterization and some details regarding the Linux TCP implementation are available in Appendix C.2.

to optimize the congestion window size. This is done in order to prevent congestion in a pro-active manner rather than in a reactive manner, which TCP normally does. The algorithm that is used for this estimation is said to be very similar to TCP Vegas [4]. The reason why this feature was not included in the experimental baseline is that this algorithm is not a TCP standard, and an enabling could therefore give results which are not representative for TCP in general.

Inflight bandwidth limiting enabled: These experiments included the bandwidth limiting feature that was excluded from the experimental baseline. This was done in order to see what possible effects this experimental feature has on TCP.

Limited transmit disabled: For these experiments we evaluated the Limited Transmit extension [1], by studying which effects a disabling of this algorithm has on a TCP connection. The Limited Transmit extension is an extension that allows a TCP sender to transmit a previously unsent segment upon the reception of each of the first two duplicate acknowledgments. The intention of this strategy is to generate additional duplicate acknowledgments that can trigger the fast retransmit mechanism. For instance, let us consider a flow where three segments are traveling from a sender to a receiver. If the first of these segments are lost, then the following segments will generate two duplicate acknowledgments which are sent back. Upon reception of these two duplicate acknowledgments, the sender will transmit two previously unsent segments. These segments cause the receiver to generate two additional duplicate acknowledgments that will trigger the fast retransmit mechanism at the sender.

Minimum RTO: In order to see what effect the lower bound of the RTO timer has, we changed the default value of this lower bound (3ms in FreeBSD) to a value of one second which is the standardized [13] lower bound of the RTO timer. The calculation of the RTO timer is somewhat complicated in FreeBSD. Each time the RTO is (re)calculated the implementation adds a so called "slop" of 200ms to the RTO value and then checks whether the current value of the timer should be adjusted with respect to the lower bound. By default the lower bound of the RTO timer is 3ms, but the use of the slop will make this bound at least 200ms. Even though this slop exists, a change of the lower bound from 3ms to one second will not cause the effective lower bound to be 1.2 seconds in general (due to the slop). Consider a calculated RTO value of 600ms. After the slop of 200ms is added to this value it will be compared to the minimum bound, and because this value is less than one second, the RTO will be adjusted to one second. However, if the calculated RTO value is 900ms, the slop will cause the RTO timer to be 1.1s. Thus, depending on the calculated RTO value the slop can impact the enforcement of the lower bound.

Larger initial windows disabled: To be able to evaluate how important the use of the larger initial windows extension [2] is for TCP performance, we

included a test case in which it was disabled. The larger initial windows extension is a standardized extension that allows a TCP sender to use an initial congestion window that is up to three full-sized segments. Before this extension was adopted by TCP implementations, a sender normally used an initial congestion window of the size $1-2$ full-sized segments. This extension is considered to increase the performance of a connection in, possibly, three different ways. Firstly, since the incorporation of delayed acknowledgments [3] in TCP, an initial window of one full-sized segment is very suboptimal since the sender may be forced to wait as long as 500ms before the acknowledgment is sent from the receiver. Secondly, for connections that transmit a small amount of data (e.g. web pages) a larger initial window may allow all data to be sent in the first round-trip time, which can considerably lower the transmission time. Finally, for connections that have a large bandwidth-delay product the link will be utilized much quicker if the initial window is larger.

The results from the test cases that were described above are presented in Section 5. Please note, though, that the TCP experiments that are described in this report are replications from earlier work [8]. The reason to why these experiments, and the results from them, are described in this report, is that this report contains a more thorough analysis of the experimental results. Furthermore, a bug in the TCP implementation was found and corrected after the completion of that study. This bug allowed TCP to use a larger initial congestion window than the larger initial window extension [2] permits. For a description of the actual bug, and a more detailed description of how it was corrected, please see Appendix C.1.4.

4.3 SCTP Experiments

In addition to the experimental details that were specified for both TCP and SCTP, a number of SCTP specific parameterizations were also evaluated. In the list below we describe the three test cases that were used. The first test case is the experimental baseline, which all the other test cases are compared against when the results are presented. These test cases were performed for two different implementations of SCTP: The KAME SCTP implementation [16] which is available for FreeBSD [7], and LKSCTP [5] which is available for the Linux operating system [10]. The default parameterization of the two implementations, along with some implementation details, is available in Appendix D.

Experimental baseline: The two different SCTP implementations were used with their original settings.

Initial RTO: Both KAME and LKSCTP use an initial RTO that is equal to the standardized value [21] of three seconds. For these experiments the initial RTO value was set to one second in order to see what effect this timer has on the behavior and performance of a SCTP flow.

Minimum RTO: In order to see what effect the lower bound of the RTO timer had, we changed it from the standardized value of one second (which is used by both KAME and LKSCTP) to the much smaller value 200ms.

The results from the test cases described above are given in Section 6. In that section the two different implementations are compared against each other, for each test case. In addition to this, a performance comparison that covers all the different bandwidths and delays that were used is also given. This performance comparison only considers flows with no packet loss and baseline configuration. This was done in order to compare the two implementations under ideal conditions.

5 TCP Results

In this section the results for the TCP experiments are shown. As mentioned in the previous section these experiments are replications from earlier experiments, with the difference that this report contains a more thorough analysis of the results, and that a bug (in the TCP implementation) that caused the initial congestion window to be larger than allowed was found and corrected.

Due to the large number of experiments that was conducted we only present a representative subset of them. For each of the different test cases mentioned in the previous section, three different "display sets" are shown in detail. These different sets combine different bandwidths and delays in order to cover the protocol performance and behavior for different bandwidth-delay products. The display sets named *low* and *high* are meant to be extreme cases in which the bandwidth-delay product either is very low or very high, while the display set *medium* is considered as the "normal" case.

Low: This display set combines a low bandwidth of 40Kbit/s with link delay of 5ms.

Medium: In this set a bandwidth of 1000Kbit/s is combined with a link delay of 40ms.

High: This set is a combination of a relatively high bandwidth of 10000Kbit/s with a link delay of 300ms.

In subsequent sections the results for each of the different test cases are shown.

5.1 Experimental Baseline

Figure 5 shows the results for the three different display sets. Shown in these figures is the total transmission time of the flow when a loss occurs at a certain position within the flow. The y-axis of these graphs shows the total transmission time, in milliseconds, that was required to transmit a flow with a loss positioned according to the x-axis. Please note that position 0 on the x-axis means that no packet loss has been introduced within that particular flow.

If we first take a look at Figure 5(b) which shows the results with medium bandwidth-delay product, we can observe a few things. Firstly, a loss of the first packet, from server to client, adds about three seconds to the total transmission time. This is due to the conservative initial setting of the RTO timer, which is standardized to three seconds [13]. If we continue to look in this figure we can see that flows that experience packet loss in the middle of the flow only require a small amount of additional time to complete the transmission. The reason to this is that the fast retransmit mechanism, which allows retransmission of lost packets before RTO expiration, is invoked for these flows. Finally, if we take a look at flows that experience packet loss late in the flow, we can see that these require slightly more time in order to complete. This is due

to the fact that fast retransmit is inhibited for losses in the end of flows, a direct consequence of the limited amount of segments that can trigger duplicate acknowledgments at the receiver[6]. However, considering the fast retransmit threshold that TCP uses only three segments should be inhibited from fast retransmission, not four as suggested in Figure 5(b). The explanation to why four, and not three, segments are inhibited from using fast retransmit is that the use of window updates[7] sometimes causes four duplicate acknowledgments to be required for fast retransmit. The implications of window updates, considering fast retransmit, is discussed in more detail in Section 7.1.3.

In Figure 5(a), which shows the display set with low bandwidth-delay product, we can see a different behavior. The only similarity with the previously mentioned results is that a loss of the first packet will cause the flow to be delayed for three seconds. However, in this figure we can see two strange peak values for losses at positions 12 and 17. These peaks are a consequence of two different implementation details in FreeBSD. The first detail regards the calculation of the RTO timer, which is done in an opportunistic way in FreeBSD. The second implementation detail is a yet unknown bug. The RTO calculation problem is covered in Section 7.1.1, and the effects of the unknown bug is described in Section 7.1.2. These two bugs combined cause a number of spurious retransmissions, which in turn causes the flow to be longer than 20 packets. This is why fast retransmit is not inhibited for flows with packet loss introduced on positions $16-19$. The graph is simply dislocated. Another implication of the two bugs is that the transmission time is longer when no losses are introduced in the flow, as opposed to when a loss is placed at, for example, position eight. This is because packets $2-7$ are transmitted twice, and a loss of one of these duplicates does not result in any performance degradation. In fact, it seems to a have a positive impact on the performance, as FreeBSD performs worse the more spurious retransmissions it detects.

The results from the experiments with high bandwidth-delay product are shown in Figure 5(c). As we can see these results are rather similar to the results shown in Figure 5(b). There are some exceptions though. It is apparent that flows that suffer from packet loss early in the flow have a relatively long transmission time if the bandwidth-delay product is high. This is due to the fact that a loss early in the flow causes TCP to switch from slow start to congestion avoidance. For these flows that, compared to the previously mentioned flows, have a large bandwidth-delay product such a switch can be costly if it happens early in the flow, when the capacity of the link has not been utilized sufficiently yet. Another thing that we can observe is that fast retransmit seems to be inhibited for yet another position (15). Even though it may seem like this is the

[6] As can be seen in the figure, a loss of the last packet does not cause the transmission time to increase. This packet is the final acknowledgment in the connection termination process, and when that packet is sent by the server the client already knows that all data is correctly received, and has therefore already reported the time that was required for the transmission to complete.

[7] Window updates are used in order to inform a TCP sender about the receive buffer at the receiver.

case, fast retransmit is actually used here. The reason why the transmission time suggests the opposite is that the two last data packets traveling from the server to the client (packet 18 and 19) are sent one round-trip time later than packet 15. This implies that the server only receives two duplicate acknowledgments and then has to wait an extra round-trip time before receiving more, which considering the large link delay will take some time.

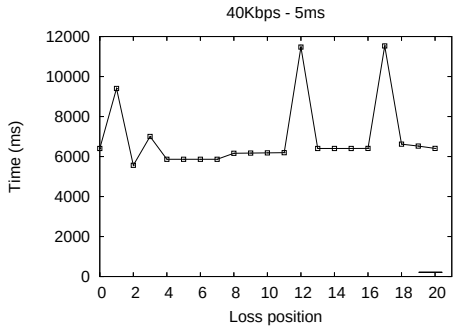
5.2 Inflight Bandwidth Limiting enabled

As mentioned earlier the FreeBSD TCP implementation incorporates a feature that tries to prevent congestion in a pro-active manner. This mechanism is enabled by default in FreeBSD 6.0, but was not included in our experimental baseline (see Section 4.2). In this section, however, we present the effects of having this mechanism enabled. If we take a look at Figure 6(b), which shows the results for a medium bandwidth-delay product, we can observe some differences compared to the experimental baseline. These differences are due to the fact that the pro-active algorithm limits the transfer rate, so that not enough packets are in flight for triggering fast retransmit. Even though it may seem strange that this mechanism does not improve the performance, but instead degrades it, we must keep in mind that we use an experimental setup that forces packets to be lost. Thus, making the task of preventing packet loss impossible for this mechanism.

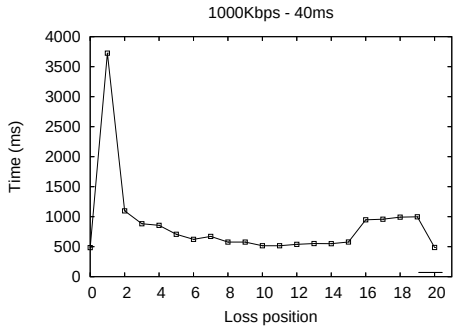
In figures 6(a) and 6(c), which show the results for low and high bandwidth-delay products, we can observe some performance gains though. For the results with low bandwidth-delay product it is, unfortunately, impossible to conclude why this gain appears. The interaction between the limiting mechanism and the previously mentioned implementation bug is too complex. The small performance gain we can observe for a loss of the first packet, in Figure 6(c), has nothing to do with the bandwidth limiting feature. Instead, it is a random variance that is a consequence of the initial RTO timer that TCP employs, and can happen regardless of the TCP parameterization. The first server segment is the `SYN_ACK` segment used in the connection establishment phase, and when this packet is lost the client will eventually retransmit the `SYN` segment (after the expiration of the initial RTO timer). However, if this new `SYN` segment is not received by the server before it times out on its own `SYN_ACK` segment, the `SYN_ACK` segment will be retransmitted as well. When the server then receives the `SYN` segment, it will once again retransmit the `SYN_ACK` segment, causing the connection establishment to be delayed by an extra round-trip time. This happened to be the case for the experimental baseline, but not for the bandwidth limiting experiments as the retransmitted `SYN` segment was received before the server had had the chance to do any retransmissions of the `SYN_ACK` segment.

5.3 Limited Transmit disabled

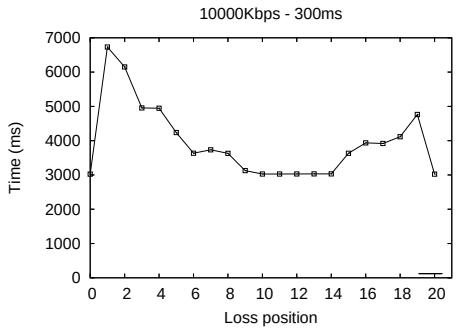
As described earlier, Limited Transmit [1] allows a TCP sender to inject a previously unsent segment upon the reception of each of the two first duplicate



(a) Low

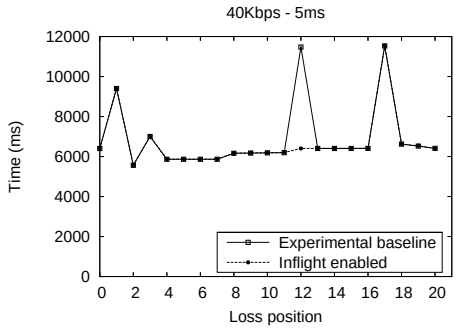


(b) Medium

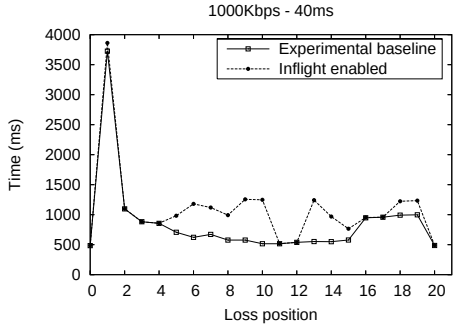


(c) High

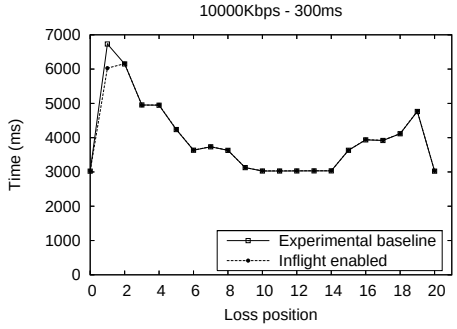
Figure 5: Experimental baseline



(a) Low



(b) Medium



(c) High

Figure 6: Inflight bandwidth limiting enabled

acknowledgments. If we take a look at Figure 7(b) we can see that a disabling of the Limited Transmit algorithm causes the transmission time to be larger for flows that experience packet loss early in the flow. This is even more evident for the results with high bandwidth-delay product, shown in Figure 7(c). The performance improvement that Limited Transmit gives, in these particular cases, is not because of the fact that additional duplicate acknowledgments are generated by the receiver. The explanation is rather that previously unsent segments are injected earlier in the network. The reason why no significant performance gains can be observed, when Limited Transmit is enabled, for losses that occur from position nine and later in the flow is that all the data has left the sender at that point in time, which means that the Limited Transmit algorithm is not triggered upon duplicate acknowledgment reception.

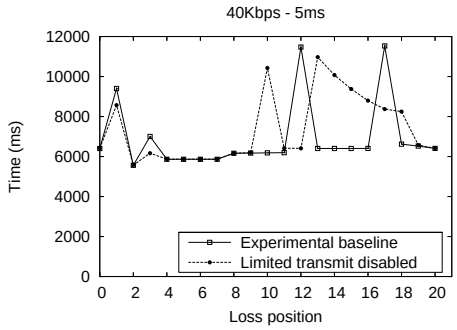
For the flows with low bandwidth-delay product, shown in Figure 7(a), we can see that a disabling of Limited Transmit has a number of different effects on the transmission time. Unfortunately, the reason to why these effects occur has not been found due to the complex interaction effects with the previously mentioned bug.

5.4 Minimum RTO

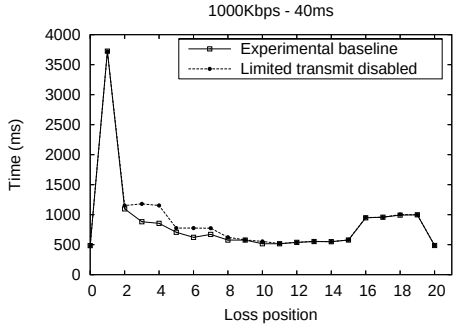
When the minimum allowed RTO was changed from approximately 200ms (which is used by FreeBSD) to the standard value of one second, as specified in [13], some interesting results were achieved. Looking at Figure 8(a), which contains the results with low bandwidth-delay product, we can see that the two peaks in the transmission time, at positions 12 and 17, has disappeared. Instead, the structure of the graph is similar to the one showing the medium results (Figure 8(b)). The reason to why the peaks existed in the first place is related to the opportunistic RTO calculation that FreeBSD performs. This opportunistic calculation is effectively disabled when the minimum RTO is set to the more conservative standard value of one second. More details about the opportunistic RTO calculation is available in Section 7.1.1.

Another interesting thing about the results shown in Figure 8(a), is that fast retransmit seems to work for a loss of packet number 16. This has not been the case for any of the other results shown so far. In theory fast retransmit should be triggered for a loss of this packet regardless of bandwidth and/or delay. However, due to the use of window updates TCP is sometimes unable to trigger the fast retransmit mechanism. The possible effects that window updates can have on the triggering of the fast retransmit mechanism was mentioned earlier, in Section 5.1, and is described in more detail in Section 7.1.3.

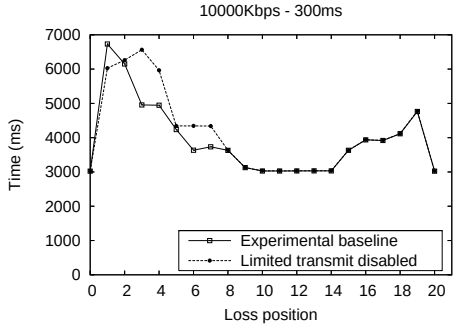
If we shift our focus to Figure 8(b) we can clearly see that the minimum RTO setting have a large impact on the performance for flows which experience packet loss late in the flow. This is natural as loss of one of the last packets can not be recovered via the fast retransmit mechanism, and therefore must rely solely on recovery via RTO expiration. This is also visible for the the results with a high bandwidth-delay product, shown in Figure 8(c).



(a) Low

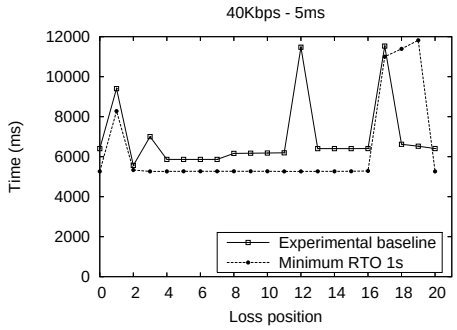


(b) Medium

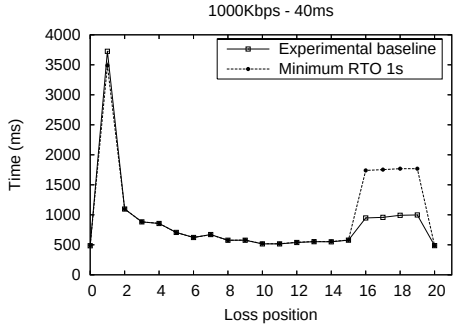


(c) High

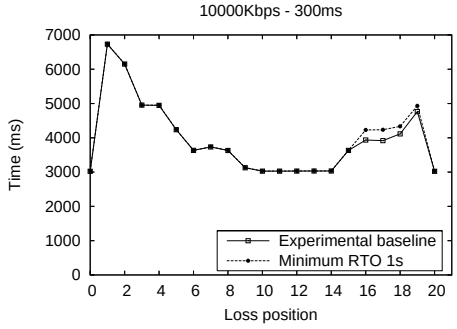
Figure 7: Limited transmit disabled



(a) Low



(b) Medium



(c) High

Figure 8: Minimum RTO = 1s

5.5 Larger Initial Windows disabled

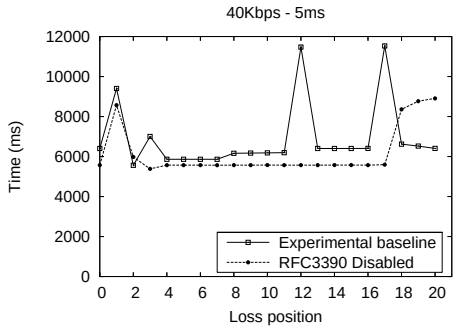
When the larger initial windows extension [2] is disabled we can see, for example in Figure 9(b) and 9(c), that the transmission time required to complete a flow increases significantly. In contrast to other TCP parameter settings that we have examined (e.g. disabling Limited Transmit, increasing the minimum RTO), a reduction of the initial window size gives a performance degradation for losses on nearly all different positions within a flow. Even for the case of no packet loss. This is due to the fact that the previously examined parameters all had in common that they were related to the TCP loss recovery mechanisms. The larger initial windows extension is not associated with these mechanisms, but merely advocates an increase in the initial window size from $1 - 2$ full-sized segments to[8]

$$\min(4 * \texttt{MSS}, \max(2 * \texttt{MSS}, 4380 \text{ bytes})) \text{ bytes},$$

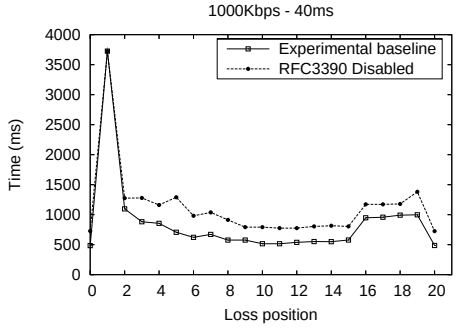
which will result in a faster utilization of the link, especially for flows with large bandwidth-delay products.

In addition to this we can see that the peaks visible for the experimental baseline in Figure 9(a) are gone, and that the performance is generally improved. The reasons for this are again due to the implementation bug and are discussed further in Section 7.1.1.

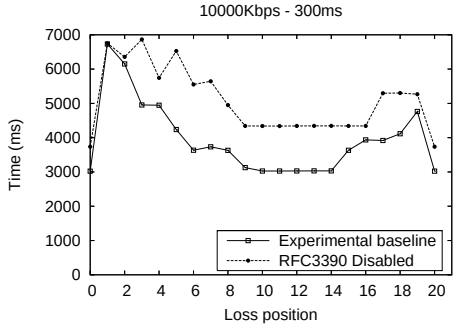
[8] In this particular implementation of TCP an initial window of one segment is used if the larger initial windows extension is disabled.



(a) Low



(b) Medium



(c) High

Figure 9: Larger initial windows disabled

6 SCTP Results

As for the TCP results (presented in Section 5), we introduce three different "display sets" which are shown in detail. These display sets are presented for each of the different test cases that were presented in Section 4. The different sets combine different bandwidths and delays in order to cover the protocol performance and behavior at different bandwidth-delay products. The display sets named *low* and *high* are meant to be extreme cases in which the bandwidth-delay product either is very low or very high, while the display set *medium* is considered as the "normal" case.

Low: This display set combines a low bandwidth of 40Kbit/s with link delay of 5ms.

Medium: In this set a bandwidth of 1000Kbit/s is combined with a link delay of 40ms.

High: This set is a combination of a relatively high bandwidth of 10000Kbit/s with a link delay of 300ms.

In the following subsections the results for each of the different test cases are described. For each test case the results for the different SCTP implementations are explained and compared to each other. But first of all, in the coming subsection, a performance comparison that covers all the different bandwidths and delays that were used is given. This performance comparison only considers flows with no packet loss and the baseline configuration. This was done in order to compare the implementations under ideal conditions.

6.1 Performance Comparison

In the graph shown in Figure 10 a performance comparison between KAME and LKSCTP is shown. The y-axis of this graph represents the additional transfer time (in percent) that KAME required, in comparison with LKSCTP, the x-axis shows the different link delays, and each one of the different lines in the graph represents one of the bandwidths that were used. Furthermore, the comparison shown in the graph only considers the baseline experiments for flows where no packet loss was introduced.

Looking at this graph, we can immediately see that LKSCTP outperforms KAME for all different combinations of bandwidth and delay. For combinations of high bandwidth and low delay the relative performance difference is very large. In fact, for flows with a bandwidth of 10000Kbit/s and a link delay of 5ms the additional time required by KAME showed to be as much as 400%.

One major explanation to the performance difference between the two implementations is that they employ different receive buffer techniques. When the last data packet, in an association, has reached the receiver, the acknowledgment will be delayed according to the 200ms SACK delay that is used by both

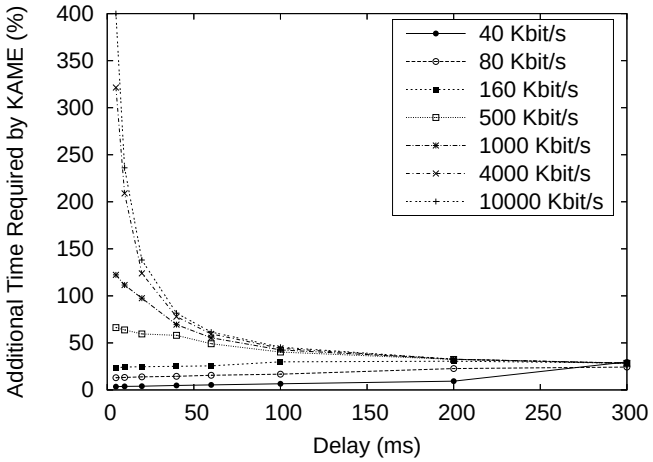


Figure 10: Performance Comparison

implementations[9]. This is done in KAME, but not in LKSCTP as it immediately sends SACK's for announcing that more receive buffer space is available[10]. By doing this, LKSCTP effectively disables the SACK delay for the last data packet, which will shorten the transmission time with approximately 200ms. For flows with high bandwidth and low delay 200ms will be a significant, if not dominating, part of the total transmission time.

Another explanation to why LKSCTP performs better than KAME is that LKSCTP uses a larger initial congestion window. KAME's initial congestion window is set to one full-sized segment, while LKSCTP uses a larger initial window of four full-sized segments.

6.2 Experimental Baseline

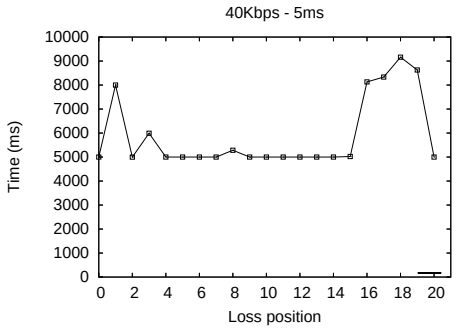
6.2.1 KAME

Figure 11 shows the results for the three different SCTP display sets using the KAME implementation. Shown in these figures is the total transmission time of a flow when a loss occurs at a certain position within the flow. The y-axis of these graphs shows the total transmission time, in milliseconds, that was required to transmit a flow with a loss positioned according to the x-axis. Note that position 0 on the x-axis corresponds to no introduced packet loss.

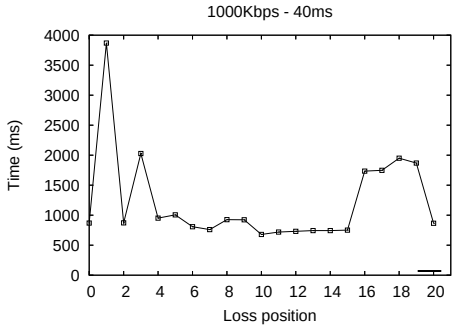
If we take a look at Figure 11(b) we can see that the SCTP results have the same basic structure as the TCP results presented in Section 5. For example, if a loss occurs in the end of the flow, fast retransmit is inhibited and loss recovery

[9]The reason why this SACK always is delayed is that the number of data packets sent by the server is odd.

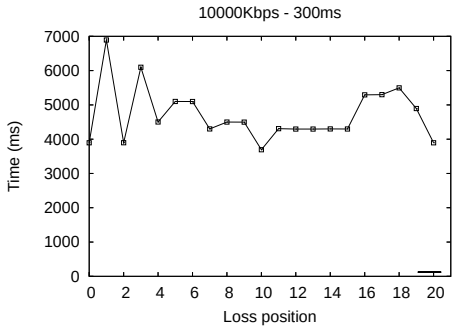
[10]This might be a consequence of the fact that LKSCTP uses a much smaller receive buffer (52736 bytes) than KAME (233016 bytes).



(a) Low



(b) Medium



(c) High

Figure 11: Experimental baseline (KAME)

can only be made possible via expiration of the RTO timer. However there are some interesting differences between the protocols. Firstly, for all results shown in Figure 11 we can observe a peak in the total transmission time if the third packet of the flow is lost. For KAME SCTP this third packet is the first packet that contains a data chunk and a loss of this packet will inevitably result in a retransmission due to timeout, because KAME only uses an initial congestion window of one segment. The reason why this was not observed for the corresponding TCP results is that the FreeBSD TCP implementation uses a larger initial window, which makes it possible to retransmit the first data packet via the fast retransmit mechanism[11]. One interesting observation that can be made, considering a loss of this data packet, is that the timeout occurs after roughly one second in Figures 11(a), 11(b) and about two seconds in Figure 11(c), even though the value of the initial RTO timer is set to three seconds. This implies that the KAME SCTP implementation uses packets sent during the association establishment phase for calculation of the RTO timer[12].

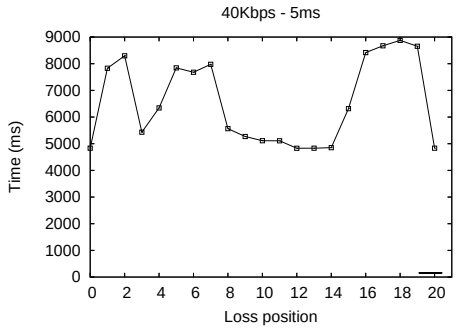
Furthermore, we can see that there exists some local peaks and dips in the transmission time for losses at other positions. For example, in Figure 11(c) we can observe that a flow that experiences packet loss on the tenth position has a very low transmission time compared to flows with packet loss on adjacent positions. To explain these deviations we must consider how a SCTP sender manages its data transmission. Like TCP, SCTP sends a number of data chunks and then waits for these to be acknowledged, after these acknowledgments are received the sender transmits a new burst of data. When losses are introduced in a flow, the total amount of data that the sender transmits will be larger, and in some cases this will require the sender to use more bursts in order to complete the transmission, which will in turn cause the transfer time to be larger. This phenomenon is described in more detail in Section 7.2.2.

6.2.2 LKSCTP

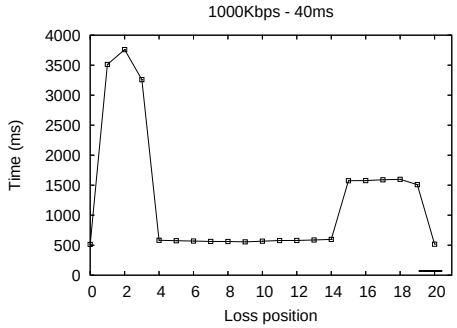
The other SCTP implementation that we consider is LKSCTP that is available for the Linux operating system. If we compare the results shown in Figure 12 with those that we got for the KAME implementation we can immediately see both commonalities and some interesting differences. One common thing that we can observe in Figure 12(c), which shows the results with high bandwidth-delay product, is the dip in the transmission time if a loss occurs at positions eight or nine in the flow, which is due to the bursty send behavior that SCTP employs. This behavior was mentioned in the previous section, and is described more thoroughly in Section 7.2.2. Another common thing is that both implementations are inhibited from using fast retransmit if a packet loss occurs late in

[11] One exception to this was the TCP experiments with the larger initial windows extension disabled, presented in Section 5.5. However, as the TCP implementation both conducts opportunistic RTO calculations and has a much lower bound on RTO timer, the transmission time was not affected significantly.

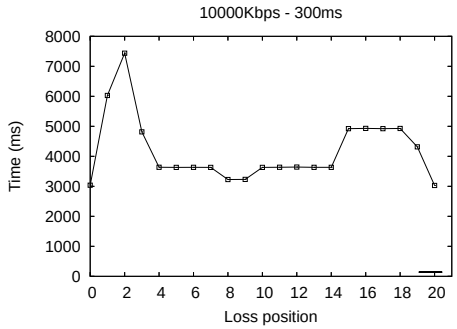
[12] The KAME RTO calculation is believed to be similar to the opportunistic RTO calculation that we have discovered that FreeBSD TCP conducts.



(a) Low



(b) Medium



(c) High

Figure 12: Experimental baseline (LKSCTP)

the flow. This is visible in all the graphs of Figure 12. However, the two implementations seem to employ different duplicate acknowledgment thresholds for the fast retransmit mechanism. The LKSCTP implementation uses a threshold of four duplicate acknowledgments, which is correct according to [21], while the KAME implementation uses a threshold of three[13]. This difference can easily be observed by comparing the flows which experience packet loss late in the transmission, shown in figures 11(b) and 12(b). In Figure 11(b), which contains results for the KAME implementation, four out of the last five packets are only possible to recover via timeout, which implies that the threshold must be set to three. If we instead take a look at the LKSCTP results, shown in Figure 12(b), we can see that RTO recovery occurs for five out of the last six packets, which indicates a threshold of four. For TCP we stated that only three packets normally are inhibited from using fast retransmit, and it may therefore seem natural that only four packets are impossible to recover via fast retransmit in SCTP. But, as no data is sent along with the `SHUTDOWN` chunk in SCTP (in contrast to the `FIN` segment in TCP) five packets will be impossible to recover via the fast retransmit mechanism, if the fast retransmit threshold is set to four.

A more significant difference between the two implementations can be observed by looking at Figure 12(a), which shows the results for the flows with low bandwidth-delay product. In this figure we can see a strange peak in the total transmission time when losses occur early in the flow (positions $5 - 8$). This interesting result is a consequence of a bug in the LKSCTP implementation which is described in more detail in Section 7.2.1.

In addition to this, some other interesting differences are visible. We can for example see that a loss of the second packet ,the `COOKIE_ACK` packet, adds a significant amount of time to the total transmission time. The LKSCTP implementation uses a timer with roughly the same value as the initial RTO timer for loss detection of this packet. KAME, on the other hand, does not care if the `COOKIE_ACK` packet is lost, instead it behaves like the association establishment is completed and starts the actual data transfer. This must, of course, be an implementation mistake in KAME. Another implementation mistake that is rather interesting and that is conducted by LKSCTP, is visible in Figure 12(b). Here we can see that a loss of the third packet, which is the first data packet from the server to the client, adds a significant amount of time to the total transmission time. The reason behind this is that LKSCTP seems to be unable to recover a loss of the first data packet with the fast retransmit mechanism, and therefore must wait for the RTO timer to expire. This is also the case for the other baseline results (Figures 12(a), 12(c)), but as they have a rather long transmission time this behavior is not as visible as in Figure 12(b).

[13] A threshold of three may be correct according to [20]. Unfortunately this specification is ambiguous about the correct value of this threshold.

6.3 Initial RTO

6.3.1 KAME

In Figure 13 we can see the results for the KAME experiments where the initial RTO has been set to the less conservative value of one second, as opposed to the default and standardized value of three seconds. As can be seen in the figure, no significant differences between these results and the experimental baseline results exist. This was expected as the opportunistic RTO calculation reduces the timer so quickly that the initial RTO timer does not have any effect. The only difference that is evident is that a loss of the `INIT_ACK` chunk (position 1 in the graph) gets timed out according to the new initial RTO value.

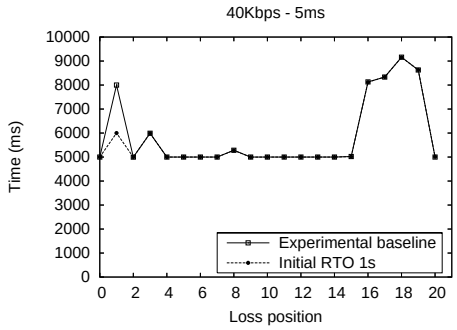
6.3.2 LKSCTP

Just like the results for KAME the initial RTO setting does not pose any major difference on the results for LKSCTP, as seen in Figure 14. This could be seen as rather surprising as LKSCTP does not conduct the same opportunistic RTO calculation as KAME. However, as LKSCTP uses a larger initial congestion window than KAME, it is able to use fast retransmit instead of RTO recovery if packet loss occurs early in the flow. Thus, the impact of the initial RTO value is not so important. The main difference compared to the baseline is, as for KAME, that a loss of the `INIT_ACK` chunk is retransmitted earlier due to the less conservative initial RTO value. There are also some variations in the results for the results with low bandwidth-delay product, Figure 14(a), but these variations are once again due to interaction effects with the retransmission bug that was previously mentioned. The most interesting thing, though, is that a loss of the first data packet, packet number three, does not seem to time out according to the new initial RTO value of one second. As mentioned previously, LKSCTP was unable to recover a loss of the first data packet with the fast retransmit mechanism, and the assumption was that it was retransmitted according to the initial RTO value. However, even though the initial RTO value is set to one second for these experiments, the retransmission of this packet seems to take longer (as seen for instance in Figure 14(b)).

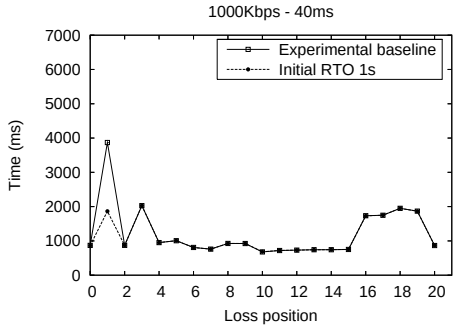
6.4 Minimum RTO

6.4.1 KAME

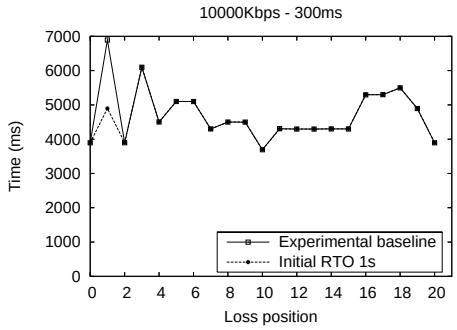
The experimental results for KAME that uses a lower limit, 200ms, for the minimum RTO differs rather significantly from the baseline results that use a lower bound of one second. If we start by looking at Figure 15(b), which shows the medium bandwidth-delay product results, we can see some expected differences. Firstly, a loss of the third packet, which is the first data packet, is retransmitted earlier due to the lower bound of the timer. Secondly, for flows that experience packet loss late in the flow, where fast retransmit is inhibited, we can see that the lower RTO bound causes the transmission time to be smaller.



(a) Low

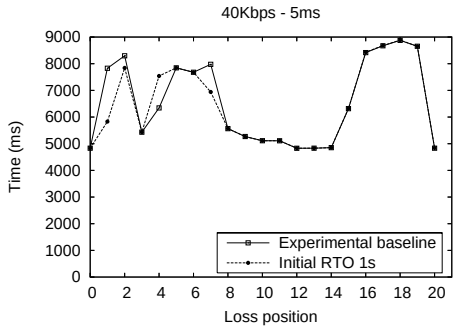


(b) Medium

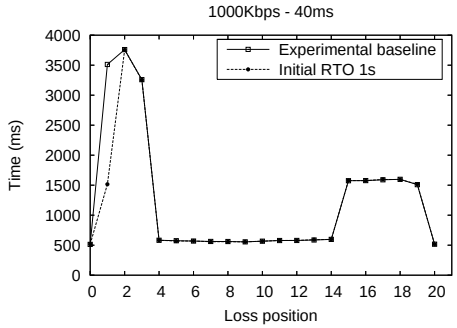


(c) High

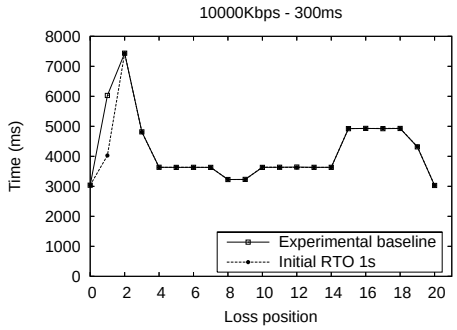
Figure 13: Initial RTO 1s (KAME)



(a) Low



(b) Medium



(c) High

Figure 14: Initial RTO 1s (LKSCTP)

However, for a loss of the last packet the transmission time is slightly higher than expected. This is a consequence of the lower RTO bound and the SACK delay. Because of the fact that nearly no SACKs are delayed during the transfer[14], the RTO timer will eventually have such a low value that it will spuriously retransmit segments for which the SACKs are delayed. In the end of the transfer, the last data packet will have its corresponding SACK delayed[15] and therefore a spurious retransmission of this packet will occur. Thus, the flows with a lower RTO bound, shown in Figure 15(b), are actually one packet longer and therefore the graph is somewhat dislocated[16].

In Figure 15(a) we can see some considerable differences between the experimental baseline results and the results with a lower bound on the RTO timer. First of all we can see that the transmission time is generally higher for the "Minimum RTO" experiments, even when no packet loss is introduced. This has to do with the opportunistic RTO calculation that we mentioned earlier. The effect of this calculation is that the three first packets will be lost before the RTO timer has "backed-off" enough in order to stop spuriously retransmitting packets. The reason why this was not observed for the baseline results is that the lower bound of the RTO timer, which we use for these experiments, accepts values that actually are so small that spurious retransmission will occur. The reason why these values are too small in the first place is a consequence of the low link delay in combination with the low bandwidth. This combination will cause the round-trip time to be much smaller for the small association initiation packets, than for the full-sized data packets that are sent later on. Thus, the RTO timer will be calculated too optimisticly, which also is allowed by our lower RTO bound of 200ms. These spurious retransmission will also cause the flow to be longer, which is the reason to why the graph is dislocated. Another interesting result is that a loss of the fifth packet causes the transmission time to be very large. Unfortunately, we have not been able to identify the reason for this. When examining the transmission logs, though, it was obvious that the RTO timer did not "back-off" accordingly (after the spurious retransmission that we mentioned earlier), but instead remained at such a low level that every single packet was spuriously retransmitted. This is rather strange as this phenomenon does not occur for packet loss placed at any other position.

Let us finally take a look at the results for a high bandwidth-delay product, shown in Figure 15(c). As for the previous results, these also exhibit some interesting anomalies. Looking at the graph we can see a general reduction of the transmission time, even when no packet loss is introduced. This may seem rather strange as a lower bound on the RTO timer should not have any effects if no packets are lost. However, the KAME implementation exhibits a very strange

[14] Because of the continuous data flow that we use, SACKs will also be generated continuously.

[15] This SACK is delayed because the number of data chunks traveling from the server to the client is uneven.

[16] All of the results shown in this figure is one packet longer but the effect is only visible for the last flow in the graph, as this flow usually has a transmission time that is approximately equal to flows with no loss.

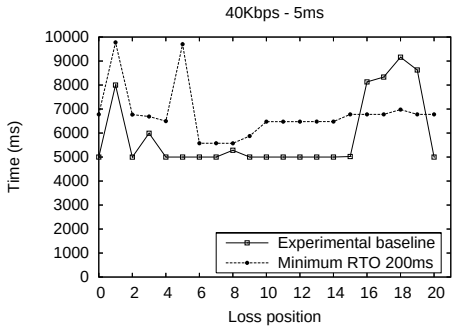
behavior. When the first data packet is sent[17] KAME waits for about 400ms and then sends three additional packets, even if no SACK has been received. This implies that some strange (and erroneous) coupling between the RTO timer and the data transmission exists in this implementation of SCTP. The effects of this behavior is that more data is initially sent, causing the transmission time to be generally lower.

6.4.2 LKSCTP

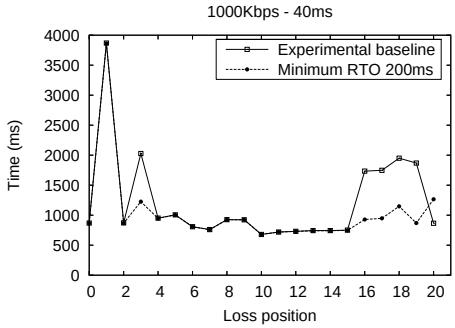
For the LKSCTP experiments with a lower RTO bound, we can observe some differences. As for the KAME results, we can see that the experiments with medium bandwidth-delay product, shown in Figure 16(b), has a lower transmission time when fast retransmit is inhibited. A small performance gain can also be observed for the low bandwidth-delay product results, Figure 16(a), when packet number two is lost. This gain is not due to the changed RTO bound though, rather due to simultaneous expirations of different timers that SCTP use, and is entirely random and can occur regardless of the SCTP parameterization. LKSCTP sends four data packets immediately after the `COOKIE_ACK` (packet number two) is sent. When a loss of this chunk occurs the other end (the client) will eventually retransmit the `COOKIE_ECHO` chunk. If this new `COOKIE_ECHO` chunk arrives at the server after the server has timed out and retransmitted the first data packet there will be a total of five retransmissions[18]. If the `COOKIE_ECHO` chunk instead arrives before the server times out, only four retransmission will be necessary (which will give a slight performance increase). For the experimental baseline the `COOKIE_ECHO` chunk happened to arrive after the server retransmitted the first data packet, and for the "Minimum RTO" experiments the situation was the opposite.

[17] Note that KAME uses an initial congestion window of one full-sized packet.

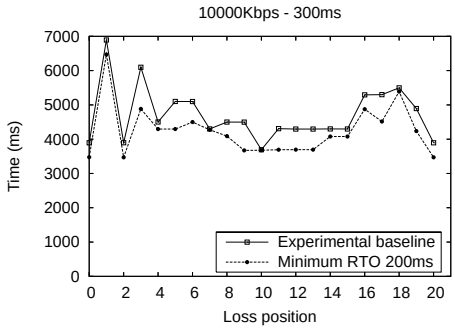
[18] No data is accepted by the other end before the COOKIE_ECHO chunk is acknowledged, therefore the server must retransmit these data packets if the COOKIE_ACK chunk is lost.



(a) Low

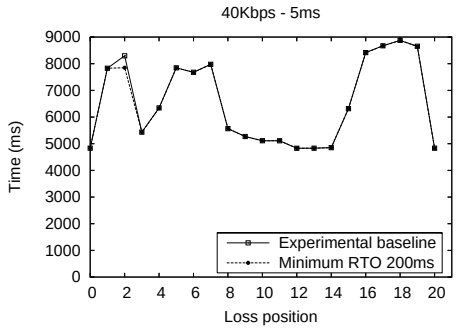


(b) Medium

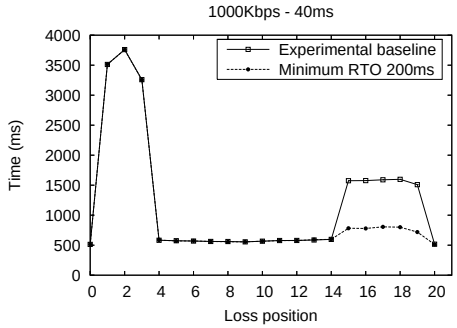


(c) High

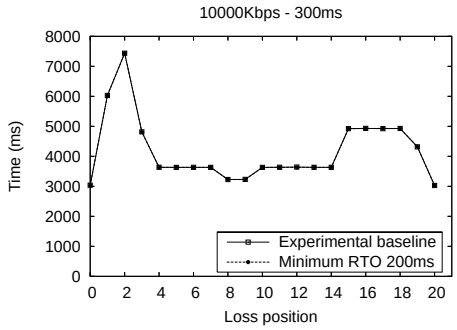
Figure 15: Minimum RTO 200ms (KAME)



(a) Low



(b) Medium



(c) High

Figure 16: Minimum RTO 200ms (LKSCTP)

7 Discussion

7.1 TCP

7.1.1 Opportunistic RTO Calculation

The TCP implementation in FreeBSD uses a RTO calculation technique that is rather opportunistic. Instead of using an initial RTO value of three seconds [13] for the data that is sent in the first window, FreeBSD uses the acknowledgments in the connection establishment phase to calculate a RTO value. This can be seen as a rather natural optimization of the RTO calculation, but there are some drawbacks with this strategy. As we could see in Section 5.1 the transmission times of the 40Kbit/s flows that had losses on positions 12 and 17 were extremely high. One of the reasons to this lies within the opportunistic RTO calculation that FreeBSD employs. Let us consider a client that connects to a server and the path that they communicate over has a bandwidth of 40Kbit/s. The round-trip time of the `SYN_ACK-ACK` segments that TCP uses while establishing the connection will be approximately $\frac{78*8}{40*10^3} + \frac{66*8}{40*10^3} = 0,0156 + 0,0132 = 0,0288$ s, if we omit the link delay[19]. If we calculate the RTO based on this round-trip time, according to the kernel sources in Appendix C.1.2, it will be approximately 287ms[20]. If we then consider the round-trip time of a full sized (1500 bytes) segment we get approximately $\frac{1500*8}{40*10^3} + \frac{66*8}{40*10^3} = 0.3132$ seconds. Thus, when the initial window of data is sent from the server to the client the packets will be timed out and spuriously retransmitted, causing a severe performance penalty on the connection. This scenario is shown in Figure 17 which shows the TCP communication from the servers point of view. In this figure we can see that the server starts by sending three full-sized segments (indicated by the three concatenated arrows in the figure). It then waits for incoming acknowledgments from the receiver, but times out and resends the first segment. When the original acknowledgment then arrives, indicated by the raised acknowledgment level in the figure, the server retransmits the next two segments. Figure 18, which shows the client, confirms this as we can see that all three segments are correctly received and that three spuriously resent segments arrive after that.

This behavior is problematic due to several reasons. Firstly, spurious retransmissions are never good as they waste bandwidth that could have been used for "real" traffic. Secondly, the RTO timer will back-off exponentially for each timeout that is experienced, thus making error recovery very expensive if a subsequent packet really is lost. However, this alone does not explain the strange peaks in the transmission time, as this problem will occur for all flows regardless of where a packet loss is placed. The reason to why losses at positions 12 and 17 of a flow give enormous transmission times is due to a possible implementation bug that is "activated" by these initial losses. The effects of this bug is more closely described in the next subsection.

[19] The SYN_ACK and ACK segments in the connection establishment phase have the sizes 78 and 66 bytes respectively.

[20] The FreeBSD TCP implementation allows RTOs that are less than the standard of one second.

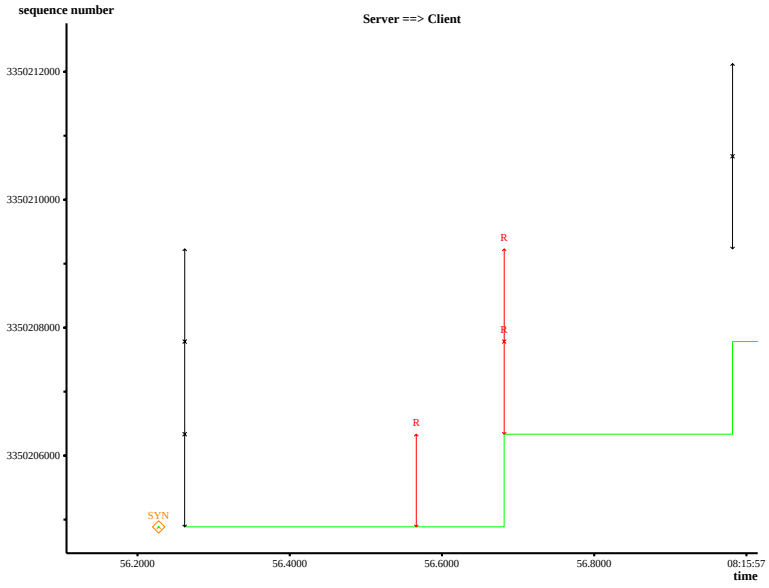


Figure 17: Opportunistic RTO calculation effects (server side)

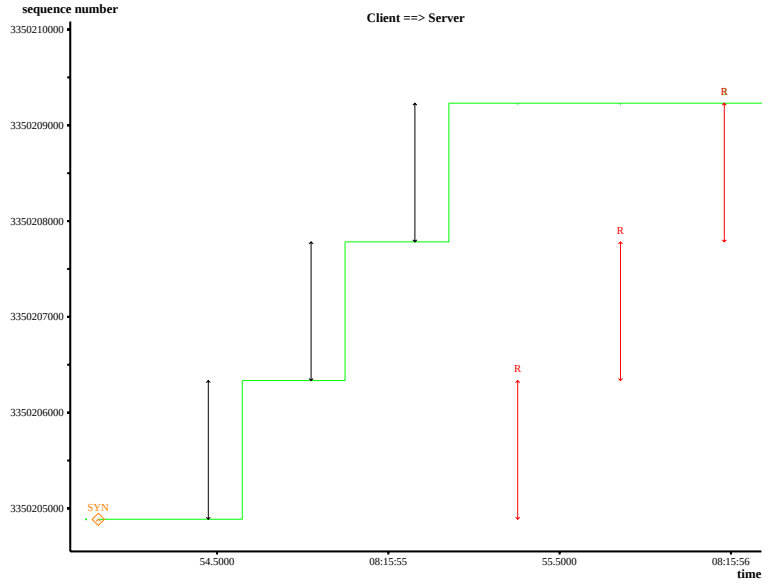


Figure 18: Opportunistic RTO calculation effects (client side)

7.1.2 Faulty Error Recovery

In addition to the RTO calculation that was described in the previous subsection, FreeBSD also has problems with its error recovery. In Figure 19, which shows the communication from the servers point of view, we can once again see the impact of the server side RTO calculation. However, compared to the flow that was shown previously this flow will experience a loss of the 12'th packet. What will happen is that when acknowledgments for new packets, packets sent after the retransmissions, are received by the server they are regarded as already received and duplicate acknowledgments are sent back to the client[21]. Because of this behavior there are not enough packets in flight to trigger the fast retransmit mechanism, instead the server waits for the expiration of the retransmission timer and then resends packet number 12. The reason why the timer expires after such a long time is because it has been exponentially backed off a couple of times, due to the spurious retransmissions, and that the implementation is unable to "undo" this exponential back-off when new acknowledgments are received.

As noted in sections 5.4 and 5.5, where we had a higher limit for the minimum RTO and disabled the larger initial windows extension, this problem dissapears sometimes. In the case of a higher limit for the minimum RTO the previously mentioned problem with spurious retransmissions will not occur and therefore this bug will not show either. The situation is similar when using a smaller initial window. Because of the fact that only one segment (instead of three) will be lost due to the low RTO, the retransmission timer has not had the chance to back-off so much.

Unfortunately the reason to why this strange bug occurs has not been found yet.

7.1.3 Duplicate Acknowledgment Ambiguity

Similar to the congestion window that a TCP sender uses to avoid flooding the network, the receiver also uses a window to protect itself from being flooded. This window, the receiver window, is used to ensure that a receiver does not receive more data than it can handle, and a sender must not send any more data than this window allows. To make it possible for a sender to know how large this window currently is, it is included in the TCP header of every segment that is sent. When an acknowledgment with updated receiver window size arrives, TCP considers the acknowledgment as a window update and adjusts its' transfer speed accordingly.

A problem that is associated with these window updates, and that was evident for a majority of the TCP results in Section 5, is that TCP is unable to differentiate between duplicate acknowledgments and window updates. This can cause more than three duplicate acknowledgments to be needed to trigger the fast retransmit mechanism. This problem occurs more often for connections

[21] Indicated by the small x's in the figure

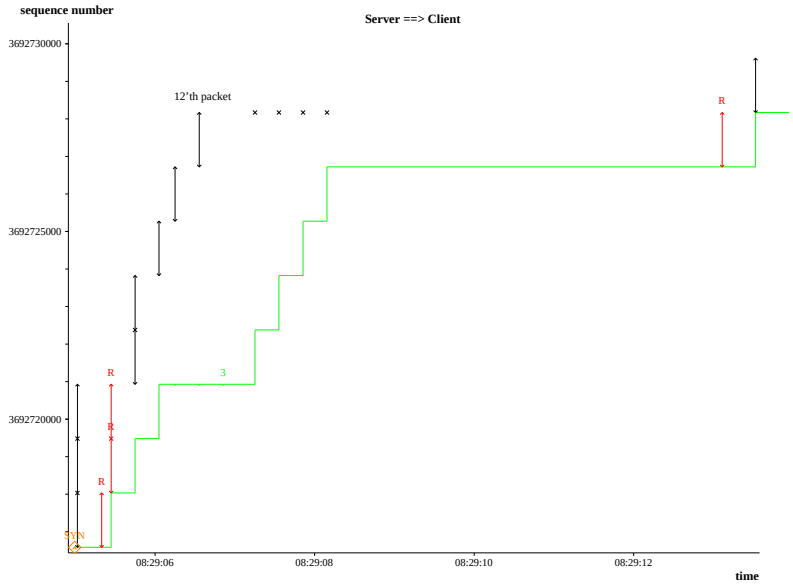


Figure 19: FreeBSD error recovery bug (server side)

that have a low link delay or a large bandwidth, as these connections are able to feed a receiver with data quickly. Thus, filling the receiver's buffer more rapidly.

7.2 SCTP

7.2.1 Retransmission bug

As shown in Figure 12(a) (section 6.2.2), LKSCTP performs very poorly when any one of the packets $4-7$ is lost. Figure 20 shows the communication from the servers perspective when the fifth packet is lost. In this figure each transmitted data packet is represented by a vertical line[22]. Furthermore, the cumulative SACK level is illustrated by the thick black line, and gap reports with dotted lines. As we can see in this figure LKSCTP starts to spuriously retransmit packets that are in flight when the retransmission of the fifth packet is conducted. This behavior then continues throughout the entire association, and has a severe effect on the performance. The reason behind this behavior is unfortunately unclear, but by comparing this scenario to that of losing the eight packet instead, where this dramatic performance degradation does not exist, we might come closer to the actual problem. If we lose the eight packet, shown in Figure 21, we can see that spurious retransmissions happens here too, but only for the last two packets (and the retransmission). One major difference between the different graphs is that the SACK flow from the client is less intense for the latter scenario, especially in the beginning of the association. This is natural

[22]Packets used for association initiation are not shown in this figure.

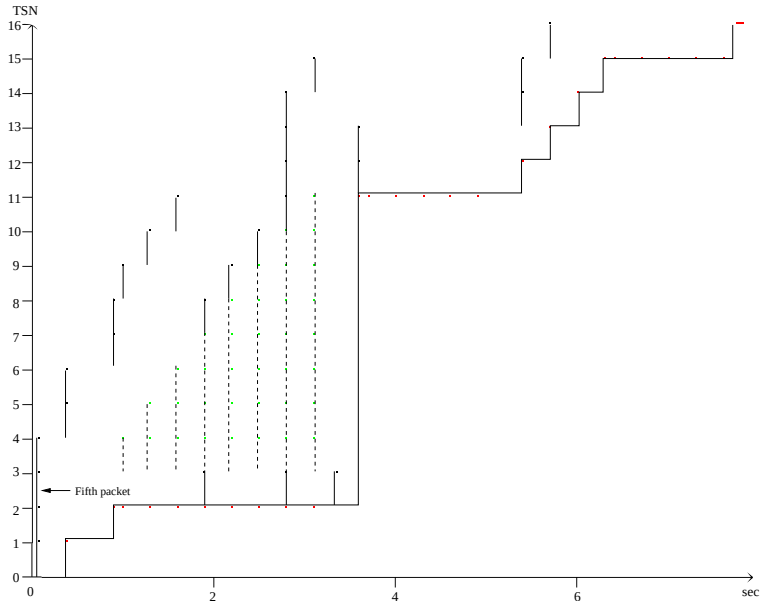


Figure 20: Retransmission bug, fifth packet lost (LKSCTP, server side)

as a receiver should delay SACKs as long as a gap in the incoming data flow is not detected. In this case the gap, due to the lost packet, will be detected much later than for the previous case (where we lost the fifth packet), which results in that the RTO estimation is conducted on delayed SACKs, in the beginning of this association. For a loss of the fifth packet hardly none of the SACKs are delayed, due to the early gap, and therefore the RTO calculation is conducted on SACKs that have not been delayed at all. Therefore it seems like the presence/absence of the SACK delay in the RTO calculation has something to do with this behavior. This could be due to the fact that an inclusion of the SACK delay will result in a more conservative RTO calculation, which results in less risk of spurious retransmissions. Important to note, though, is that this problem only seems to occur for flows that have a very small bandwidth-delay product.

7.2.2 Impact of bursts

The reason why we could observe dips in the total transmission time when packet loss occurred at certain positions in the SCTP flows, for example in Figure 11(c) position ten, is a direct consequence of the sending behavior of SCTP combined with the length of the flow. The transmission behavior of SCTP is to send a number of packets, wait for them to be acknowledged, and then send more packets. Figure 22 shows the communication from the servers point of view for the KAME baseline flow with packet loss on position ten,

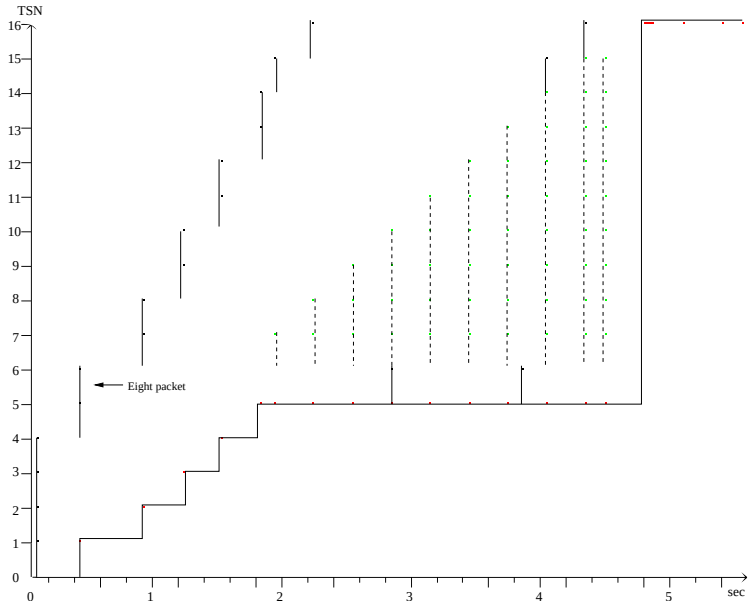


Figure 21: Retransmission bug, eight packet lost (LKSCTP, server side)

bandwidth of 10000Kbit/s, and a delay of 300ms. As we can see in this figure, the transmission is ordered into four distinct bursts. If we instead consider Figure 23, which shows the communication of a similar flow, but with packet loss on position nine, we can observe that one additional burst is required in order to complete the transmission. This extra burst adds one round-trip time to the total transmission time, which can be a significant amount of time if the delay is large. The reason to why an extra burst is required for the latter case is that the congestion window has not grown enough in order to allow enough new data to be sent along with the retransmission, in order to complete within four bursts. For flows that experience packet loss after the congestion window has grown enough in order to allow this, the problem can still occur. If we once again consider Figure 11(c), we can see that the transmission time for a flow with packet loss on position eleven also suggests that five bursts are required. This time it is not congestion window limitations that results in the extra burst, instead it is due to the fact that all the data destined for the client already have been sent when the loss is detected. Thus, an extra burst is needed in order to do the retransmission.

8 Future Work

The results that were achieved by our experiments did not only reveal the expected dependencies between packet loss position and protocol behavior, but they also revealed a number of implementation issues in all the different proto-

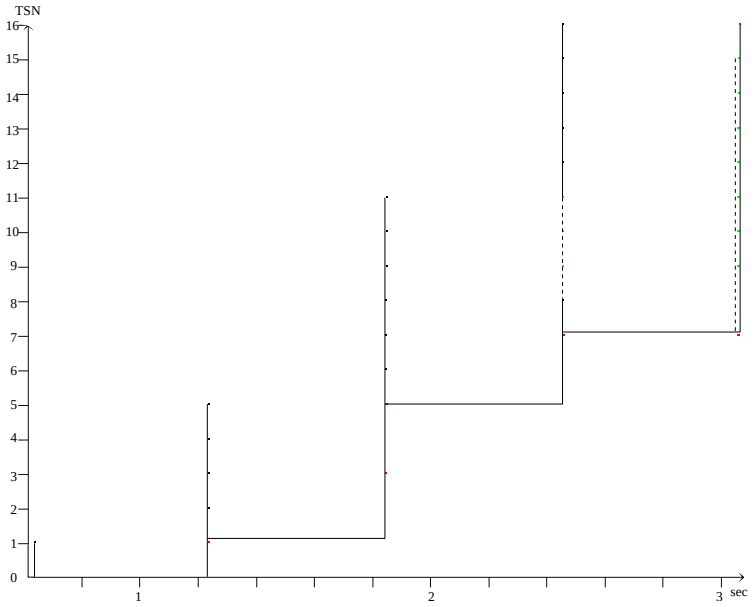


Figure 22: Transfer completes within four bursts (KAME, server side)

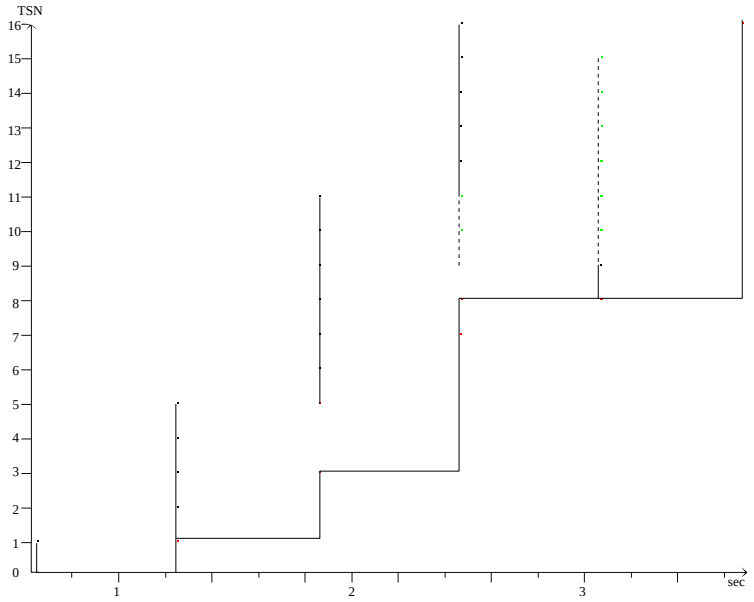


Figure 23: Transfer completes within five bursts (KAME, server side)

col implementations. One idea of future work that may be possible in the light of this is to further analyze the different implementations (maybe by source code inspection) in order to fully pin-point some of the anomalies we have seen. Another interesting approach is to evaluate the impact of fast retransmit inhibitions using a more representative traffic pattern, as we only used a bulk transfer of fixed size with no competing traffic. Examples of such traffic patterns could be web traffic for TCP and signaling traffic for SCTP. It would also be nice to investigate if the other deterministic features of the KauNet network emulation software (including deterministic bit errors, delay changes, and bandwidth changes) could be helpful in analyzing protocol performance and behavior.

9 Summary

In this technical report we have investigated the dependency between packet loss position and protocol behavior/performance of short TCP and SCTP flows. By using a network emulation software that allowed us to decide which packets that should be lost in a flow, we showed that the position of a packet loss is an important factor for the total transmission time of a flow. There are several reasons for this, either because the general protocol specification forces a certain behavior for certain packet losses, or because of specific implementation choices that lead to different behavior. We have also shown that some of the most popular implementations of TCP and SCTP include implementation bugs that cause their behavior to be strange in some situations.

References

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

[2] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, Internet Engineering Task Force, October 2002.

[3] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989.

[4] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, volume 24, pages 24–35, October 1994.

[5] IBM Corporation. The lksctp Project. http://lksctp.sourceforge.net.

[6] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, Internet Engineering Task Force, April 2004.

[7] The FreeBSD Foundation. FreeBSD. http://www.freebsd.org.

[8] P. Hurtig. Fast retransmit inhibitions for TCP. Master's thesis, Karlstad University, February 2006.

[9] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, Internet Engineering Task Force, May 1992.

[10] Inc Linux Kernel Organization. The Linux Kernel Archives. http://www.kernel.org.

[11] M. Mathis and J. Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *SIGCOMM'96*, pages 281–191, August 1996.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Internet Engineering Task Force, October 1996.

[13] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, Internet Engineering Task Force, November 2000.

[14] J. Postel. Internet Protocol. RFC 791, Internet Engineering Task Force, September 1981.

[15] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981.

[16] The KAME Project. Kame. http://www.kame.net.

[17] WIDE Project. WIDE. http://www.wide.ad.jp.

[18] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[19] L. Rizzo. Dummynet and Forward Error Correction. In *Freenix 98*, New Orleans, June 1998.

[20] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. RFC 4460, Internet Engineering Task Force, April 2006.

[21] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, Internet Engineering Task Force, October 2000.

[22] R. Stewart, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758, Internet Engineering Task Force, May 2004.

[23] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. RFC 3309, Internet Engineering Task Force, September 2002.

[24] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. *INFOCOM*, 2004.

A TCP & SCTP Congestion Control

Both TCP and SCTP include reliability mechanisms that are used for retransmitting data that has been lost in the network. The reason why this packet loss occurs is usually that an intermediate router has run out of buffer space and therefore must discard incoming packets. If the end hosts do not consider these buffer space limitations, but instead continue to transmit data with a constant rate, problems will occur. What happens is that a reliable sender will detect this packet loss and retransmit the lost data, possibly along with new data. Thus, increasing the amount of data in the network even further. Eventually, if this behavior continues, the network will suffer from a congestion collapse which results in that no meaningful data reaches its destination. The problem of congestion collapses was discovered in the mid 80's and as a reaction to this phenomenon two different algorithms were introduced, to prevent further collapses. The idea behind these algorithms, slow start and congestion avoidance, was that packet loss almost always happens as a consequence of congestion, and therefore these algorithms were designed with the presence of packet loss as a cornerstone.

The remainder of this appendix is structured as follows; firstly, the slow start algorithm is described. Secondly, the congestion avoidance algorithm is described. Finally, a short summary of these mechanisms is provided.

A.1 Slow Start

There is no easy way of correctly determining the bandwidth that is available for a certain network path, and it can therefore be hard for a host to choose a transfer rate that does not cause network congestion. To overcome this problem TCP/SCTP uses an algorithm called slow start. Slow start is conducted in the beginning of every TCP/SCTP connection and its main purpose is to find the maximum available bandwidth at which it can send data without causing the network to be congested. To accomplish this, slow start forces the TCP/SCTP sender to transmit at a slow sending rate and then rapidly increasing it until the available bandwidth between the hosts is believed to be found. The slow start mechanism introduces a new window to the sender: the congestion window (`cwnd`). When a new connection is established `cwnd` is initialized according to

$$\texttt{cwnd} = \min(4 * \texttt{MSS}, \max(2 * \texttt{MSS}, 4380)) \text{ bytes}$$

in TCP, and in SCTP this variable is set according to the following condition

$$\texttt{cwnd} \leq 2 * \texttt{MSS}$$

Each time an acknowledgment is received the size of `cwnd` is increased by one segment, allowing the sender to transmit at least two new segments. This approach will lead to an almost exponential growth of the `cwnd`. Even though this strategy causes the cwnd to grow large very quickly the sender is never allowed to transmit more data than the receiver advertised window allows, even

if the cwnd is larger. Eventually some intermediate router will not be able to handle this growing traffic flow, without dropping packets. When this happens TCP/SCTP will interpret the lost packets as a sign of congestion and enter congestion avoidance.

A.2 Congestion Avoidance

If the receiver window is large enough, the slow start mechanism described in the previous section will eventually increase `cwnd` to a point where one (or more) intermediate router(s) start discarding packets. As mentioned earlier both TCP and SCTP interpret packet loss as a sign of congestion, and when this happens the congestion avoidance mechanism is invoked. Even though slow start and congestion avoidance are two different mechanisms they are more easily described together. In the joint description below a new variable is introduced. This variable, `ssthresh`, is the slow start threshold which TCP/SCTP use to determine if slow start or congestion avoidance is to be conducted.

1. When a new connection/association is established `cwnd` is initialized to

$$0 < \texttt{cwnd} \leq \begin{cases} \min(4 * \texttt{MSS}, \max(2 * \texttt{MSS}, 4380)) \text{ bytes} & \text{for TCP,} \\ 2 * \texttt{MSS} \text{ bytes} & \text{for SCTP.} \end{cases}$$

2. The sender can then send a maximum of

$$\min(\texttt{cwnd}, \texttt{rwnd}) \text{ bytes}^{23}.$$

3. When congestion is detected by the sender the `ssthresh` variable is set to

$$\texttt{ssthresh} = \min\left(\frac{\texttt{cwnd}}{2}, 2 * \texttt{MSS}\right) \text{ bytes}$$

   If the congestion was detected due to timeout slow start is conducted, otherwise congestion avoidance is performed.

4. When acknowledgments for new data is received `cwnd` is increased differently depending on which congestion control mechanism that is currently in use. If slow start is performed (`cwnd` $<$ `ssthresh`) `cwnd` will be increased as described in the previous section. If we, on the other hand, are in congestion avoidance then `cwnd` will be increased according to

$$\texttt{cwnd} = \texttt{cwnd} + \frac{1}{\texttt{cwnd}} \text{ bytes}$$

   which will result in a linear increase of the `cwnd` variable.

[23] This is not completely true as SCTP is allowed to always have at least one packet in flight to enable receiver window updating.

As mentioned in point (3) in the above list, congestion avoidance, and not slow start, is performed if congestion was detected using duplicate acknowledgments. This behavior allows a higher throughput under congestion, especially when using large congestion windows. Receiving three duplicate acknowledgments tells the sender more than the expiration of the retransmission timer. Since the receiver only can generate duplicate acknowledgments when it is receiving other segments it is an indication that data still flows between the different hosts, and that the congestion is not that severe. By using this approach, skipping the slow start, TCP/SCTP do not reduce the transfer rate unnecessarily much.

A.3 Summary

The slow start and congestion avoidance mechanisms that were described earlier in this appendix are summarized in Figure 24.

The first part of the graph shows how slow start is conducted until a packet loss occurs. As can be seen in the figure the packet loss is detected with a timeout at round-trip time 3, and when this happens the `ssthresh` variable is set to half of `cwnd`s current value and slow start is performed again. Furthermore, at round-trip time number 9 a packet loss is recovered via the fast retransmit algorithm, and therefore congestion avoidance is entered instead of slow start.

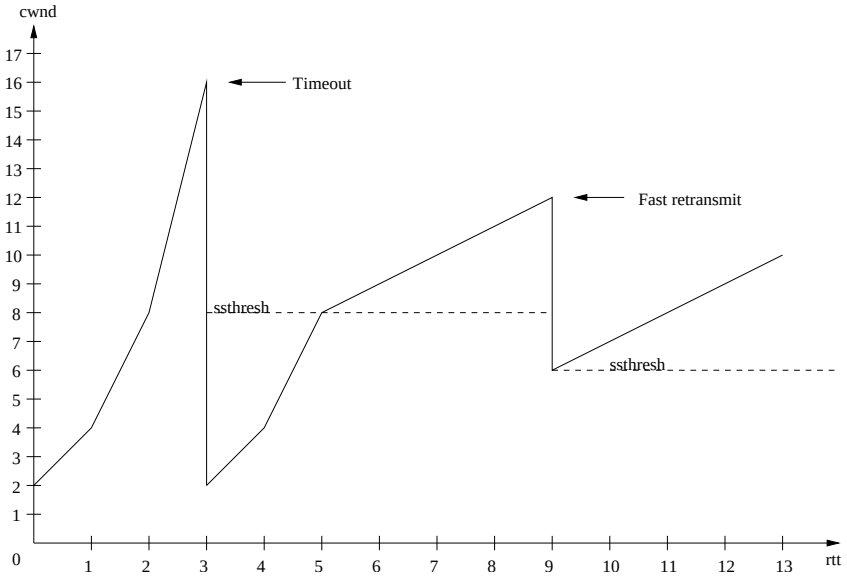


Figure 24: Congestion Control Summary

Experiment	Flow length	Bandwidth	Delay	Replications	Values
Baseline	20	40, 1000	10, 300	3	252
Inflight enabled	20	40, 1000	10, 300	3	252

Table 1: Experiments with host cache enabled

B Experiments conducted

In this appendix all experiments that have been conducted are listed. For each combination of protocol (TCP/SCTP) and protocol implementation the conducted experiments are listed in table form. These tables contain information about the parameters that were evaluated[24], how long the data flow was, which combinations of bandwidth and delay that were used, how many replications that were made of the experiment, and finally how many actual experiments the test case consisted of. In the next section the TCP experiments are listed and in Section B.2 the SCTP experiments are listed.

B.1 TCP

B.1.1 FreeBSD 6.0

As we conducted the FreeBSD experiments certain problems arose. The first problem we encountered during the TCP experiments was that the host cache that FreeBSD uses in order to optimize connections could not be controlled in a satisfying way. Before this was discovered a couple of tests were conducted. These are listed in Table 1. After this problem was corrected (see Appendix C.1.3) a large set of experiments were run. These experiments were used in a previous study [8], and are listed in Table 2. However, as this work was completed a bug in the calculation of the initial window was discovered. This bug caused the initial window, the window that TCP uses as congestion window immediately after connection establishment, to be larger than specified. After this bug was corrected (see Appendix C.1.4) a new series of experiments was conducted. These experiments are the experiments presented in this report. A full listing of these experiments is given in Table 3.

B.1.2 Linux 2.6.15

Even though the results from the Linux TCP experiments are not presented in this report, as they did not differ significantly from the FreeBSD results, the list of experiments conducted with this implementation is given in Table 4.

[24] For more information on these parameters, please see Appendix C for TCP and Appendix D for SCTP.

Experiment	**Flow length**	**Bandwidth**	**Delay**	**Replications**	**Values**
Baseline	20	40, 80, 160 320, 500, 1000 2000, 4000, 8000 10000	5, 10, 20 40, 60, 80 100, 150, 200 250, 300	3	6930
Inflight enabled	20	40, 80, 160 320, 500, 1000 2000, 4000, 8000 10000	5, 10, 20 40, 60, 80 100, 150, 200 250, 300	3	6930
Performance extensions disabled	20	40, 1000	10, 300	3	252
Limited transmit disabled	20	40, 1000	10, 300	3	252
Larger initial windows disabled	20	40, 1000	10, 300	3	252
Min RTO 1s	20	40, 1000	10, 300	3	252
Min RTO 1s & Inflight enabled	20	40, 1000	10, 300	3	252
SACK disabled	20	40, 1000	10, 300	3	252
Baseline	100	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	16968
Inflight enabled	100	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	16968

Table 2: Experiments with host cache disabled

Experiment	Flow length	Bandwidth	Delay	Replications	Values
Baseline	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Inflight enabled	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Limited transmit disabled	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Larger initial windows disabled	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Min RTO 1s	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527

Table 3: Experiments without IW bug

Experiment	Flow length	Bandwidth	Delay	Replications	Values
Baseline	20	40, 80, 160 320, 500, 1000 2000, 4000, 8000 10000	5, 10, 20 40, 60, 80 100, 150, 200 250, 300	3	6930
Vegas Congestion Control	20	40, 80, 160 320, 500, 1000 2000, 4000, 8000 10000	5, 10, 20 40, 60, 80 100, 150, 200 250, 300	3	6930
Forward Acknowledgments (fack) disabled	20	40, 80, 160 320, 500, 1000 2000, 4000, 8000 10000	5, 10, 20 40, 60, 80 100, 150, 200 250, 300	3	6930

Table 4: Linux 2.6.15

Experiment	Flow length	Bandwidth	Delay	Replications	Values
Baseline	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Initial RTO 1s	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Min RTO 3ms	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Early fast retransmit enabled	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527

Table 5: KAME experiment list

B.2 SCTP

B.2.1 Kame (FreeBSD)

In this section of the appendix the different experiments that were run for the KAME SCTP implementation are presented. The different test runs are given in Table 5.

B.2.2 LKSCTP (Linux)

In this section of the appendix the different experiments that were run for the LKSCTP implementation are presented. The different test runs are given in Table 6.

Experiment	Flow length	Bandwidth	Delay	Replications	Values
Baseline	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Initial RTO 1s	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Min RTO 3ms	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527
Partial reliability disabled	20	40, 80, 160 500, 1000, 4000 10000	5, 10, 20 40, 60, 100 200, 300	3	3527

Table 6: LKSCTP experiment list

C TCP Implementation details

Although TCP is standardized by the IETF in a number of different RFC's, different implementations of the protocol differ from each other in a number of aspects. This is due to a variety of reasons. Firstly, it can be due to ambiguities in the specification, which inevitably will lead to different interpretations and implementation details. Secondly, some implementors may introduce enhancements/optimizations which are not standardized, but do not interfere and/or break the standard behavior of the protocol. Finally, different operating systems have different development cycles which may lead to that new TCP standards, or enhancements, are available much earlier in operating systems with a short development cycle.

In this appendix we will shortly describe two different TCP implementations; the FreeBSD 6.0 implementation, and the TCP implementation found in version 2.6.15 of the Linux kernel. The FreeBSD version is examined more closely in this appendix, as the experiments that are described in this technical report were performed with this implementation.

C.1 FreeBSD 6.0

The FreeBSD 6.0 TCP implementation is a modern TCP implementation which supports a majority of the IETF standards that currently are available. In addition to these standards, the FreeBSD implementation also employs an experimental algorithm that tries to prevent congestion by pacing the packet transmission. This algorithm is said to be roughly equivalent to the TCP Vegas [4] algorithm, and is enabled by default in the operating system[25].

In the following subsections of this appendix we will look into different aspects of this TCP implementation. Firstly, we will see what features a regular user can tweak without making any changes to the implementation itself. Secondly, we will look into how the initial RTO is calculated, how we managed to modify the implementation to allow independent experiments to be run, and how we eliminated an implementation bug that caused the initial window to be larger than intended.

C.1.1 TCP Parameters

In FreeBSD, and most other operating systems, a user is allowed to configure certain aspects of the kernel behavior without having to make modifications to the kernel itself. In order to make this possible the operating system must provide the user with an interface to the kernel. In FreeBSD, and other Unix dialects, this interface is called `sysctl` and it can be used to control a large number of kernel related parameters.

In Table 7 a subset of the kernel parameters that are used by the TCP implementation is shown. If one wants to change the behavior of the TCP

[25] This algorithm is called "Inflight bandwidth limiting".

TCP Parameter	Default value	Description
rfc1323	Enabled	Performance extensions for TCP [9].
rfc3042	Enabled	The Limited Transmit extension [1].
rfc3390	Enabled	The Larger Initial Windows extension [2].
newreno	Enabled	The NewReno TCP implementation [6].
sack.enable	Enabled	The use of selective acknowledgments (SACK's) [12].
inflight.enable	Enabled	Congestion prevention algorithm.
rexmit_min	3ms	Minimum RTO allowed.
rexmit_slop	200ms	RTO slop, a value that is added to the RTO timer each time it is recalculated.

Table 7: FreeBSD TCP Parameters

implementation by disabling the Limited Transmit algorithm, for example, it can be done by issuing the command:

```
sudo sysctl net.inet.tcp.rfc3042=0
```

All the TCP parameters that are shown in Table 7 can be accessed with the help of the `sysctl` command, and the prefix `net.inet.tcp.` For a complete list of the parameters that are related to the TCP implementation, one can enter the following commands:

```
sudo sysctl -a | egrep tcp
```

C.1.2 Initial RTO Calculation

The initial RTO calculation in FreeBSD 6.0 is conducted in two different steps, and takes place in the `sys/netinet/tcp_input.c` file. First of all the smoothed round-trip time, and the round-trip time variation are set according to

```
tp->t_srtt = rtt << TCP_RTT_SHIFT;
tp->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
```

where `TCP_RTT_SHIFT` is defined as `5`, and `TCP_RTTVAR_SHIFT` as `4`. After this step is completed the RTO timer (`tp->t_rxtcur`) is set with the macro `TCPT_RANGESET` according to

```
TCPT_RANGESET(tp->t_rxtcur, TCP_REXMTVAL(tp),
  max(tp->t_rttmin, rtt + 2), TCPTV_REXMTMAX);
```

where `TCPT_RANGESET` is defined as

```
#define TCPT_RANGESET(tv, value, tvmin, tvmax) do { \
  (tv) = (value) + tcp_rexmit_slop; \
  if ((u_long)(tv) < (u_long)(tvmin)) \
    (tv) = (tvmin); \
  else if ((u_long)(tv) > (u_long)(tvmax)) \
    (tv) = (tvmax); \
} while(0)
```

and `TCP_REXMTVAL` as

```
#define TCP_REXMTVAL(tp) \
  max((tp)->t_rttmin, (((tp)->t_srtt >> (TCP_RTT_SHIFT - \
  TCP_DELTA_SHIFT)) + (tp)->t_rttvar) >> TCP_DELTA_SHIFT)
```

and the other constants are defined as; `TCPTV_REXMTMAX = 64000`, `TCP_DELTA_SHIFT = 2`.

C.1.3 Host Cache

FreeBSD uses a host caching mechanism in order to store and reuse TCP state information. This in order to optimize new connections to hosts that the machine has had previous experience with. The experiments that are summarized in this report are always conducted between two distinct hosts and therefore this cache must be flushed between the experiment runs, to ensure that they are totally independent from each other. Luckily, this cache can be flushed with the use of a sysctl variable named `net.inet.tcp.hostcache.purge`. The problem is that the actual flushing happens five minutes after the sysctl is used, which is inconvenient if a large number of experiments is to be executed. To solve this problem line 134 in the source file `sys/netinet/tcp_hostcache.c` was changed from

```
#define TCP_HOSTCACHE_PRUNE 5*60 /* every 5 minutes */
```

to

```
#define TCP_HOSTCACHE_PRUNE 5 /* every 5 seconds */
```

C.1.4 Initial Window bug

A relatively new TCP standard that is enabled by default in the FreeBSD 6.0 TCP implementation is the proposal of increased initial windows [2]. The initial window is, in TCP, defined as the number of bytes that a TCP sender is allowed to transmit to the receiver after the connection establishment phase is completed. Previously TCP has used an initial window which has been equal to one or two MSS worth of bytes, but with this extension the initial window can be as large as three full-sized segments.

There are, though, a serious problem with the incorporation of this standard in the FreeBSD TCP implementation. The problem is that TCP misinterprets the final acknowledgment in the connection establishment phase as an ordinary

acknowledgment, thereby increasing the congestion window with additionally MSS bytes. This will result in that the number of segments that are initially sent from a sender to a receiver is four instead of three.

In Listing 1 we can see that the intention the TCP implementers had was to "skip rest of ACK processing" if the acknowledgment only was acknowledging a `SYN` segment in the connection establishment phase. By skipping the rest of the acknowledgment processing, the code which increases the congestion window, shown in the second part of Listing 1, is supposed to be skipped. The problem with this approach, though, is that the variable `acked`, which is the amount of data that the acknowledgment covers, will be `1` when a `SYN` segment is acknowledged. This is due to the fact that `SYN` segments use one byte of the sequence number space, even if no data is actually sent.

This implementation problem was solved by making a small change to the code, that inhibited the congestion window to be increased upon the reception of the first acknowledgment.

```
/*
 * If no data (only SYN) was ACK'd,
 *    skip rest of ACK processing.
 */
if (acked == 0)
        goto step6;

/*
 * When new data is acked, open the congestion window.
 * If the window gives us less than ssthresh packets
 * in flight, open exponentially (maxseg per packet).
 * Otherwise open lienarly: maxseg per window
 * (maxseg^2 / cwnd per packet).
 */
if ((!tcp_do_newreno && !tp->sack_enable) ||
    !IN_FASTRECOVERY(tp)) {
        register u_int cw = tp->snd_cwnd;
        register u_int incr = tp->t_maxseg;
        if (cw > tp->snd_ssthresh)
                incr = incr * incr / cw;
        tp->snd_cwnd = min(cw+incr, TCP_MAXWIN
                                    <<tp->snd_scale);
}
```

Listing 1: Lines 2093–2098 in netinet/tcp_input.c

C.2 Linux 2.6.15

The TCP implementation that is included in version 2.6.15 of the Linux kernel is a cutting edge implementation that does not only support a large set of the

TCP Parameter	Default value	Description
`congestion_control`	`bic`	BIC congestion control [24].
`fack`	`Enabled`	Forward acknowledgments [11].
`reordering`	`3`	Fast retransmit threshold.
`sack`	`Enabled`	Selective acknowledgments (SACK's) [12].
`window_scaling`	`Enabled`	Part of the performance extensions specified in [9].
`timestamps`	`Enabled`	Part of the performance extensions specified in [9].

Table 8: Linux TCP Parameters

IETF standards, but also includes a large set of experimental features. For example, a user can choose which congestion control scheme to use. In addition to the standardized congestion control, that was shortly described in Appendix A, this version of the Linux kernel provides congestion control schemes that have been developed for certain types of networks. The congestion control that is enabled by default in Linux is BIC [24] which is a congestion control scheme that was developed for fast long-distance networks.

In the next subsection of this appendix a list of user controllable features are listed.

C.2.1 TCP Parameters

The parameters shown in Table 8 contains a subset of the TCP parameters that a user can change without making any modifications to the actual implementation. Like FreeBSD, Linux also gives the user access to these parameters via the `sysctl` command. To be able to change a TCP parameter, a user must add the prefix `net.ipv4.tcp_` to the parameters listed in the table below.

D SCTP Implementation details

Like TCP, SCTP is standardized in a number of RFC's by the IETF. However, different implementations of SCTP may incorporate different features and standard parameterizations for a number of different reasons. This can be due to ambiguities in the specification, implementation specific enhancements/optimizations, or the development cycle of the implementation.

In this appendix we will shortly describe two different SCTP implementations; the KAME implementation that is available for FreeBSD, and the LKSCTP implementation found in version 2.6.15 of the Linux kernel.

D.1 KAME

The KAME SCTP implementation is the result of a subproject of the KAME project [16] that was launched 1998 (and terminated March 2006). The primary mission of the KAME project was to develop and deploy reference code of IPv6/IPsec and other, related, protocols. A majority of the code that was developed during the KAME project has been merged into a number of BSD operating systems, such as FreeBSD and NetBSD. However, the SCTP implementation (and some other protocols) that were developed during this project have not yet been merged into any operating system, as they were not considered to be ready yet. Instead, the development of these protocols are currently conducted by a working group in the WIDE project [17].

By using the `sysctl` command in FreeBSD it is possible for a user to configure certain SCTP features. In Table 9 a subset of these user controllable parameters is shown.

D.2 LKSCTP

The LKSCTP implementation [5] is a result of a project started by Randall Stewart, one of the SCTP coinventors. The first development release of this implementation was made public in January 2001. The first "real" release of the implementation was in March the same year, and it was later included within the Linux kernel (from version 2.5).

In Table 10 a subset of the different parameters that can be changed via use of the `sysctl` command is given.

SCTP Parameter	Default value	Description
rto_max	60000	The maximum value that the RTO timer can be set to.
rto_min	1000	The lowest value that the RTO timer can be set to.
rto_initial	3000	The initial value of the RTO timer.
init_rto_max	60000	The maximum value of the INIT retransmission timer.
init_rtx_max	8	The maximum number of times an INIT chunk can be retransmitted.
delayed_sack_time	200	The value of the SACK timer.
nr_outgoing_streams	10	The default number of outbound streams for an association.
early_fast_retran	0	Allows early fast retransmits according to the associated timer value.
early_fast_retran_msec	250	The lower bound of the early fast retransmit timer.

Table 9: KAME SCTP Parameters

SCTP Parameter	Default value	Description
rto_max	60000	The maximum value that the RTO timer can be set to.
rto_min	1000	The lowest value that the RTO timer can be set to.
rto_initial	3000	The initial value of the RTO timer.
max_init_retransmits	8	The maximum number of times an INIT chunk can be retransmitted.
sack_timeout	200	The value of the SACK timer.
prsctp_enable	1	The partial reliability extension to SCTP [22][26].

Table 10: LKSCTP Parameters

Loss Recovery in Short TCP/SCTP Flows

The Transmission Control Protocol (TCP) has been the dominant transport protocol within IP-based networks for many years, mainly due to the reliability it provides to its users and the congestion control it employs. However, as the amount of signaling traffic within IP-based networks have increased significantly in recent years, it has become clear that TCP is not suited for this kind of traffic. In order to meet the requirements of signaling traffic the Stream Control Transmission Protocol (SCTP) was developed by the Internet Engineering Task Force (IETF). SCTP is heavily influenced by TCP and is therefore similar to TCP in many ways. One example is the SCTP loss recovery and congestion control mechanisms which are almost identical to those of TCP. The primary purpose of this work is to study the performance and behavior of the TCP/SCTP loss recovery mechanisms for short flows. Using a simple client/server model, we evaluate the performance of these mechanism over a wide range of bandwidths, link delays and packet loss patterns. The experiments evaluate one TCP implementation and two SCTP implementations, and were conducted using network emulation. The experimental results show that there exist strong dependencies between the position of packet loss and the actual transmission time of the corresponding flow. In addition to these dependencies, we also found a number of implementation mistakes in the examined protocol implementations.

Karlstad University Studies
ISSN 1403-8099
ISBN 91-7063-101-8