UNIVERSITY OF OSLO
Department of Informatics

Evaluation of SCTP retransmission delays

Master thesis

Jon Pedersen

24th May 2006

Abstract

Many applications today (e.g., game servers and video streaming servers) deliver time-dependent data to remote users. In TCP based systems, retransmission of data might give high varying delays. In applications with thin data streams (e.g., interactive applications like games), the interaction between players raise stringent latency requirements and it is therefore important to retransmit lost or corrupted data as soon as possible.

In the recent years, SCTP has been developed to improve several requirements not found in TCP. In this thesis, SCTP is compared, tested and evaluated against the default Linux TCP protocol with respect to retransmission latency in different and varying RTT and loss scenarios. Various enhancements are proposed, implemented and tested.

Contents

Chapter 1

Introduction

1.1 Background and motivation

The large improvement in computer technology in the recent years have provided new requirements to the applications. Today, it is more and more common that many applications interact with each other across large distributed systems such as the Internet. Examples are distributed video stream applications and massive multi-player online games (MMOGs) such as role-playing games, first person shooter games and real-time strategy games. MMOGs are increasing in size and complexety and supports hundreds or thousands of concurrent players [17]. Many players are moving around in a virtual world and frequently interact with each other at the same time, in a way that is experienced as a real world. Thus, the players need to get the same information at more or less the same time to have a consistent view of the game. Because of the interaction between players in the game, MMOGs must deliver time-dependent data to remote users which require stringent latency requirements. E.g, data loss should not affect the experience of the game.

Most MMOGs are running on a centralized server and distribute game data in a point-to-point communication with the players. To recover from errors, the servers are mostly running the standard protocols of the system kernels. In most cases, this means running TCP as it is one of the most used transport protocols on the Internet today. In the recent years, a new transport protocol named SCTP has been developed to answer the requirements for transmission of signaling data. It is based on TCP, but has been considered more appropriate than TCP for congestion controlled streaming because of its support of partial reliability,

multi-homing and several streams inside a connection.

Both TCP and SCTP are developed to achieve the highest possible throughput in various network scenarios. Thus, they will try to make use of all the available bandwidth in the network to transmit as much data as fast as possible to remote users. This is characterized as a thick stream. When there are much data in transit, recovery from errors such as data loss is achieved quickly because of the quick feedback from receivers. This is important when data needs to get retransmitted as it leads to low retransmission delays. However, applications like MMOGs do make use of very thin streams when transmitting data to remote users. The characteristics of a thin stream is that very few packets are sent once in a while without the need to make use of the available bandwidth and achieve the highest possible throughput. The packets are often small compared to the available payload.

1.2 Problem definition

Both TCP and SCTP have not been optimized for thin streams. This leads to a behaviour where data is not retransmitted fast enough when data loss occurs. When data is sent in thin streams and needs to get retransmitted, the feedback from the receivers are so infrequent that it leads to high and varying delays. This is criticial for interactive applications that must deliver time-dependent data.

In this thesis, we will test and compare SCTP against TCP NewReno with respect to retransmission delays. TCP NewReno is the standard TCP protocol in the Linux kernel version 2.6.15. As SCTP is based on TCP, both protocols have much of the same properties, but there are also several differences that affect the retransmission mechanisms in various ways. Based on our findings from the testing and comparison, we propose several enhancements for SCTP that improve the retransmission delays in thin streams.

In this thesis, we are considering sender side modifications only as it is impossible to change the implementation of hundreds or thousand of receivers. The reason is that they are running various operating systems and also different versions of the SCTP protocol that is impossible to change.

1.3 Outline

The thesis is organized as follows. In chapter 2 we are presenting TCP NewReno and SCTP. In chapter 3, TCP NewReno and SCTP are tested and compared in both thin and thick streams with respect to retransmission delays in different loss scenarios. Based on our findings, we evaluate and implement several proposed enhancements for SCTP in chapter 4 with the goal of improving the retransmission mechanisms in thin streams. In chapter 5 we test and evaluate the proposed enhancements before we conclude our work in chapter 6.

Chapter 2

Background

In this chapter, we are presenting TCP NewReno and SCTP. As TCP NewReno is an extension to TCP Reno, we will first describe the properties of TCP Reno and later explain what sort of extensions TCP NewReno makes to TCP Reno.

2.1 Transmission Control Protocol

The Transmission Control Protocol (TCP) [9] is standard transport protocol in most computer networks today. Examples of its usage is web-traffic, e-mail and file-transfer. TCP is connection-oriented which means that a connection is established to send packets between the sender and receiver. Each packet consists of a specific header and a data payload. The data payload inside a TCP packet is organized as a continous sequence of bytes which governs the sequence number assignments. Thus, TCP is said to be byte-oriented. TCP ensures that all packets are sucessfully received in the same order as they were sent.

TCP uses congestion control to determine the capacity of the network and further adjust the number of packets it can have in transit. If there is little traffic in the network, TCP increases the sending rate. Otherwise, if congestion is detected, TCP decreases the sending rate. This leads to a Additive-increase, multiplicative-decrease (AIMD) of the sending rate. TCP uses acknowledgments (ACK) from the receiver to adjust the number of packets it can have in transit. An ACK is a verification of that a packet has left the network and new packets could be sent without increasing the level of congestion. If ACKs fail to arrive in a given time or

packet loss is detected, this is an indication of congestion in the network and the sender must reduce the sending rate accordingly.

2.1.1 TCP Reno

TCP Reno [10] is a varition of the TCP protocol which TCP NewReno is built on. TCP Reno makes use of four mechanisms during congestion control. These are slow-start, congestion avoidance, fast retransmit and fast recovery.

Slow-start and congestion avoidance are used by the sender to control the amount of outstanding data being injected into the network. Outstanding data is data that is sent, but not yet acknowledged. In order to control this, a congestion window is used. The congestion window is a limit on the amount of data transmitted before an ACK is received. The congestion window is determined by how much congestion there is in the network in addition to the amount of data the receiver is capable to receive. The latter is known as the advertised window and its capacity is placed in each ACK to report the sender of how much data that can be received. This means that the sender can have the minimum of the congestion window bytes and the advertised window bytes in transit without getting an ACK. This is known as flow-control.

The slow-start mechanism inhibits the sender from sending large bursts of data into the network without knowing anything about its capacity and is mainly used when a connection is initialized. Initally, the congestion window is set less or equal to twice the Sender Maximum Segment Size (SMSS). During slow start, the congestion window is incremented by SMSS bytes for every ACK received in one Round-Trip Time (RTT). Initially, one packet is sent and when the ACK for this packet is received, the congestion window is incremented from one to two. The sender continues to send two packets and two ACKs are received. When both packets are acknowledged, the congestion window is incremented to four packets and so on. This gives an exponential growth per RTT. If the congestion window reaches the slow-start threshold (ssthresh), TCP enters congestion avoidance. Its initial value could be arbitrary large.

During congestion avoidance, the congestion window is incremented by one per RTT. This process continues until congestion is detected. Congestion is indicated by a retransmission timeout where an ACK does not arrive before the retransmission timer expires or when three duplicate ACKs are received. If a packet is lost, then the sender continues to send duplicate ACKs for the last packet in the received sequence until the se-

quence is restored. In both cases, a retransmission is neceassary. When a retransmission timeout occurs, ssthresh is reduced to half of the current congestion window before going back to slow-start. Further the lost packet is retransmitted.

If three duplicate ACKs are received, the fast retransmit mechanism is used. Fast retransmit ensures that the packet is retransmitted before the retransmission timer expires. Next the fast recovery mechanism is used instead of slow-start to transmit new packets until a non-duplicate ACK arrives. Fast recovery uses the receipt of duplicate ACKs to indicates that packets have left the network and no longer consume network resources. The sender can therefore continue to transmit new packets, but ssthresh is reduced to half of the congestion window. As 3 duplicate ACKs have left the network, the congestion window is set to ssthresh+3*SMSS to be sure to avoid congestion at the first time. In this way, the congestion window is increased by the number of packets (three) that have left the network. For each duplicate ACK that is received, the congestion window is incremented by one and sender can continue to send new packets for each duplicate ACK that is received. When a non-duplicate ACK is received, fast-recovery is left and the congestion window is set to ssthresh before the congestion control enters congestion avoidance. The non-duplicate ACK should at least acknowledge the retransmitted packet and also all new packets sent under fast recovery.

The TCP Retransmission timer value (RTO) is calculated according to [12]. But Linux TCP differs from this calculation by setting the minimum RTO to hz/5 = 200 ms. The RTO calculations results in an RTO equal to minimum RTO + the measured RTT. When the retransmission timer expires, TCP performs an exponential backoff by doubling the RTO value.

TCP NewReno

TCP NewReno [8] is an extension to TCP Reno with the goal of improving the fast retransmit and fast recovery mechanisms. In TCP Reno, fast recovery is left after a non-duplicate ACK acknowledges the retransmitted packet. If there are multiple loss of packets in a window, TCP Reno will retransmit each lost packet by fast retransmit and fast recovery where both need to start over again for each retransmitted packet. Each time, the congestion window and ssthresh is reduced accordingly. The ACK for a retransmitted packet will acknowledge some, but not all packets. This is known as a partial acknowledgment.

When three duplicate ACKs are received, TCP NewReno will reduce ss-

thresh to half of the current congestion window and enter fast retransmit. The lost packet is retransmitted and the current congestion window is reduced to ssthresh+3*SMSS as in TCP Reno. When an ACK is received, TCP NewReno will check if it acknowledges the packet with the highest sequence number. If that is not the case, the ACK is a partial acknowledgment and confirms that one or more packets are lost in the same window. Hence, the packet acknowledged by the partial acknowledgement is retransmitted and TCP NewReno continues to retransmit one packet per RTT until it receives an ACK for the packet with the highest sequence number. The reason for this is that it will take an RTT to get an acknowledgment for each retransmitted packet. Thus, if there are multiple loss of packets, TCP NewReno does not know the next lost packet before an acknowledgment is received for the previous packet. When the packet with the highest sequence number is acknowledged, TCP NewReno will leave fast recovery, reduce the congestion window to ssthresh and enter congestion avoidance.

2.1.2 Selective Acknowledgement

TCP does not acknowledge packets that are not located in the left edge of the receiver window. If a packet is lost, then the next packets are not acknowledged before the expecting packet arrives although they are successfully received. If there are multiple loss of packets in a window, then the sender must wait an entire RTT to discover each lost packet. This may lead to uneccesary retransmissions and a reduction of the throughput.

Selective Acknowledgment (SACK) [6] is a strategy that corrects this problem by informing the sender of which packets that are received. This allows the sender to discover multiple packet loss and only retransmit those packets that are actually lost. The SACK option provides this information by storing each received block's first and last 32 bits in the header where each block is acknowledging a continous stream of received bytes.

2.1.3 Forward Acknowledgement

Forward Acknowledgment (FACK) [19] is developed to improve TCP congestion control during recovery and works in combination with SACK. The FACK algorithm uses the additional SACK information to explicitly measure the number of outstanding packets in the network. TCP with

or without the SACK extension both estimates this by assuming that a duplicate ACK represents a received packet which is taken out of the network.

FACK uses the state variables snd_nxt and $retran\_data$ to estimate the amount of outstanding data in the network. The first is increased when a new packet is sent, the latter is increased when a lost packet is retransmitted. The highest sequence number acknowledged by SACK is stored in snd_fack. These variabels are used in combination with $cwnd$ to decide if a packet should be sent in the following way:

```
while(snd.nxt < snd.fack + cwnd - retran_data){
  send_something()
}
```

2.1.4 Duplicate SACK

Duplicate SACK (DSACK) [7] is an extension to SACK and uses this option to acknowledge duplicate packets. When duplicate packets are received, the first block of the SACK option should be a D-SACK block and used to report the sequence numbers of the duplicate packets that triggered the acknowledgement. This extension makes it possible for the sender to find the order of packets at the receiver and find out if a packet is retransmitted unnecessary.

2.2 Stream Control Transport Protocol

The Stream Control Transmission Protocol (SCTP) [11] is a new transport protocol which provides new functionality to the transport layer, compared to the functionality found in TCP. Originally, SCTP was developed to provide a transport protocol for message-oriented applications such as transportation of signalling data. SCTP provides a number of functions that are considered critical for signaling data, but they can also provide transport benefit to other applications requiring additional performance and reliability.

SCTP is message-oriented, unlike TCP which is byte-oriented. This means that SCTP sends a sequence of messages within a stream that are delivered to applications requiring message-oriented data transfer. Data

is transmitted in chunks which are a unit of user data or control information within a SCTP packet consisting of a specific chunk header and specific contents dependent of its usage. A message from the application layer is transmitted in a data chunk which has its own unique Transmission Sequence Number (TSN). Several chunks of different types may get bundled into one packet as long as the total size of the packet does not exceed the Maximum Transmission Unit (MTU) of the network path. If a message does not fit into a single packet according to the MTU, it is fragmented into multiple data chunks where each fits into a packet. When fragmented messages are received, the data contents of the data chunks are defragmented into the original message.

SCTP uses SACK to acknowledge the receipt of data chunks. A SACK chunk does in addition to the specific chunk headers consist of the cumulative TSN and the size of the advertised receiver window. Further, a SACK chunk consists of several gap ack blocks, denoting received chunks when there are holes in the data chunk sequence. Each gap ack block contains the offset number of the first TSN and last TSN of a continous block of data chunks. The TSNs can be found by adding the offset number to the cumulative TSN. At last, a SACK chunk consists of blocks of duplicate TSNs. Multiple blocks of SACK information can get bundled into one packet as long as the total size of the packet does not exceed the network MTU. In the absence of loss, a SACK is sent back for every second packet received or within 200 ms of the arrival of any unacknowledged data chunks. If one or more holes in the received data chunk sequence is detected, the receiver will immediately send a SACK back for every incoming packet until the data chunk sequence is restored.

Data could get transmitted in one or more streams within a single association and are subject to a common congestion and flow control. These mechanisms are based on the mechanisms found in TCP Reno. This means that SCTP is using slow-start and congestion avoidance in its procedures. During slow-start the initial congestion window is set to 2 * MTU. The initial value of ssthresh could be arbitrarily large. If a SACK advances the cumulative TSN, then the congestion window is increased by at most the lesser of the total size of the previously outstanding data chunks acknowledged and the MTU. During congestion avoidance, the congestion window is increased by 1 * MTU per RTT. A data chunk is not considered to be fully delivered until the cumulative TSN of a SACK passes the TSN of the data chunk. This is because the incoming SACKs could give different reports of what data chunks are missing and delivered caused by packet reordering in the network. Thus, it is the size of the data chunks acknowledged by the cumulative TSN that controls the size of the congestion window. the network.

SCTP supports multi-streaming which allows data to be partitioned into multiple streams. These are sent sequentially, independent of each other. Message loss therefore only affect delivery within that stream and none of the others. This makes it possible to continue sending messages to unaffected streams while the receiver is buffering messages in the affected stream until retransmission occurs. Unlike SCTP, TCP sends data in a single stream. When messages are lost or appear to be out-of-order, TCP must delay delivery of all data until the lost message is retransmitted or the sequence of messages is restored.

SCTP supports multi-homing which allows a SCTP endpoint to support multiple IP addresses within an association. A single address is chosen as the primary address and is used when data is transmitted under normal conditions. Retransmitted data could use one or more alternate addresses to improve the probability of reaching the endpoint. If a primary address is inaccessible, then one of the alternate addresses is chosen as the primary address. To support multi-homing, endpoints need to exchange lists of available IP addresses during the initiation of the association. Each endpoint receives data from any of the addresses associated with a remote endpoint and share a common port number.

A data chunk can get retransmitted by a retransmission timeout or the receipt of 4 SACKs that all report the data chunk to be lost and trigger a fast retransmit. If multi-homing is used, there is one retransmission timer per destination transport address. An RTT measurement is made each round trip and is used to calculate the RTO in the following way:

- If no RTT measurement has been made, set RTO = 3000 ms.
- When the first RTT measurement R has been made:
    1. Set the smoothed RTT (SRTT) = R.
    2. Set the RTT variance (RTTVAR) = $\frac{R}{2}$
    3. Set RTO = SRTT + 4 * RTTVAR.
- When a new RTT measurement R' has been made:
    1. Set RTTVAR = (1 - β) * RTTVAR + β * $|SRTT - R'|$ where $\beta = \frac{1}{4}$
    2. Set SRTT = (1 - α) * SRTT +α * R' where $\alpha = \frac{1}{8}$.
    3. Set RTO = SRTT + 4 * RTTVAR.
- If the RTO is less than 1000 ms, set RTO = 1000 ms
- If the RTO is greater than 60 seconds, set RTO = 60 seconds.

Each time a data chunk is transmitted, the retransmission timer is started if it is not already running. The retransmission timer is stopped whenever all outstanding data chunks have been acknowledged. If a SACK acknowledges some, but not all outstanding data chunks, then the retransmission timer is restarted with the current RTO. When the retranmission timer expires, then the retransmission timer will backoff exponentially by setting the new RTO to twice the old RTO and no more than 60 seconds. If a new RTT estimate is made after an exponential backoff, it will result in a newly calculated RTO. If the retransmission timer is restarted after an exponential backoff, it will use the newly calculated RTO to collapse the exponential RTO back to its normal value. After a retransmission timeout, SCTP will enter slow start by setting ssthresh to half of the current congestion window and the new congestion window to 1 * MTU. Further, the number of the earliest outstanding data chunks that will fit into a single packet is determined, starting with the earliest outstanding data chunk. These data chunks are then bundled and retransmitted.

Fast retransmit is triggered by four SACKs. Whenever the sender receives a SACK that reports missing data chunks, it will wait for three further SACKs reporting the same data chunks as missing before going into fast retransmit. When data chunks is reported missing in the fourth consecutive SACK, SCTP will:

- Mark the missing data chunks for fast retransmit.
- Set ssthresh to half of the current congestion window and the new congestion window to ssthresh.
- Determine how many of the earliest data chunks marked for fast retransmit that will fit into a single packet and retransmit this packet.
- Restart the retransmission timer only if the last SACK acknowledged the earliest outstanding data chunk, or the endpoint is retransmitting the first outstanding data chunk sent.

By waiting for fourt consecutive SACKs, SCTP tries to reduce spurious retransmissions caused by packets that are received out of order. SCTP has no fast recovery mechanism specified.

2.3 Summary

TCP NewReno and SCTP are both transport protocols with similar properties. The biggest difference is that TCP is byte-oriented while SCTP is message-oriented. This means that TCP sends and acknowledge bytes where each packet has its own sequence number. In contrast, SCTP puts data into data chunks where each data chunk has its own sequence number inside a packet. If there is room in a packet according to the MTU, several data chunks can get bundled inside a SCTP packet.

TCP NewReno and SCTP use the same congestion control algorithm, but they differ in how it is controlled. TCP NewReno increases the congestion window by 1 SMSS for each ACK. In contrast, SCTP increases the congestion window by at most the lesser of the size of the acknowledged data chunks and 1 MTU. Both protocols uses slow-start in the initialization of a connection or after a retransmission timeout. In addition, SCTP can bundle several outstanding chunks in a packet after a retransmission timeout. If ssthresh is exceeded, both go into congestion avoidance and continue to increase the congestion window by SMSS or MTU bytes once per RTT until packet loss is detected. TCP NewReno uses 3 duplicate ACKs to trigger a fast retransmit before entering fast recovery. In contrast, SCTP uses 4 SACKs to trigger a fast retransmit and has no specified fast recovery mechanism. The RTO calculation of SCTP differ from the one found in TCP NewReno. In TCP NewReno the minimum RTO is 200 ms. In SCTP the minimum RTO is 1000 ms.

In the next chapter, we will test TCP NewReno and SCTP with respect to retransmission delays and further evaluate and compare both protocols against each other.

Chapter 3

Testing SCTP and TCP NewReno

We wanted to test and compare SCTP against TCP NewReno with respect to retransmission delays. In this chapter we will describe the test configuration and further present and evaluate the results from the tests.

3.1 Test Configuration

In order to compare SCTP with TCP, streams are sent by both protocols in turn with the same test conditions. To get a complete evaluation and comparison of retransmissions in both protocols, they are tested with thick and thin streams, respectively.

Testing the protocols in thick streams gives a view of how both protocol's retransmission strategies are performing when the sender injects as much data as possible into the network. This way, it is possible to see the effects of the retransmission strategies in an environment they are developed to work well in. This is important for evaluating the retransmission strategies when streams get thin and better understand the performance of the retransmission strategies. Testing the protocol in thin streams gives a view of how both protocol's retransmission strategies are performing when the sender injects very few packets into the network during a given time period. As SACK, DSACK and FACK are developed to improve TCP's retransmission strategies, it is important for the evaluation to see how TCP is performing with various combinations of these mechanisms, both in thin and thick streams.

3.1.1 Test tools

In this section, the tools used in the tests are described.

Netperf

Netperf [4] is a benchmark tool which sends specified TCP streams from a sender to receiver. Basically Netperf sends as much data as it can, but the thickness of the stream can be reduced by the optional parameters -w, -b and -m. The first one specifies a burst interval in milliseconds, the second one specifies the burst size (denoting how many TCP messages that are sent from Netperf in a given burst interval) and the latter one specifies the message size.

Netem

Netem [3] is a tool which makes it possible to emulate various network types by setting delay rules and packet loss rules on chosen network interfaces. Netem is used in combination with tc [14] which manipulates traffic control settings in the Linux kernel.

Sctpperf

Sctpperf [13] is a SCTP benchmark tool which sends SCTP data from a sender to a receiver. The size of each message can be specified and the tool supports the multiple streams and multi-homing features of SCTP. Originally, the tool sends as much data as possible, but it has been modified to sleep a given time interval before sending a new message which results in thinner streams.

Linux kernel implementation of the SCTP protocol

The Linux kernel implementation of the SCTP protocol (Lksctp) [2] is beeing used during the tests. It comes with the Linux kernel version 2.6.15 and is loaded into the kernel as a kernel module.

Tcpdump

Tcpdump [15] is a tool which let you specify a network interface for which packets are captured on. The packets can be dumped to screen or a tracefile which can be read and analyzed by programs based on libpcap [5], a packet capture library for Linux. As standard, Tcpdump uses a packet capture buffer of maximum 96 byte to minimize the overhead of handling a packet capture buffer in the kernel. Often, the length of the packets are found to be larger than 96 byte, limited by the network MTU. A buffer size of 96 byte is enough to extract TCP headers as they are all located in the beginning of the packet. Most of the time network behavior is analyzed, one is not interested in the data contents of a packet. However, when SCTP is bundling several chunks into one packet, the header of the next chunk is stored after the payload of the last chunk, making them sparse. An enlargement of the packet capture buffer is therefore necessary to extract all SCTP headers and its size is set by the -s parameter.

Tcptrace

Tcptrace is a tool which is used to analyze TCP traffic. It prints various network statistics including retransmission statistics to standard output. Tcptrace considers the minimum, maximum and average retransmission delay of consequtive retransmissions.

Sctp_trace

In order to get retransmission strategies from SCTP streams, a program *sctp_trace.c* has been written to analyze SCTP streams from a libpcap tracefile and print retransmission statistics to screen. In contrast to Tcptrace, Sctp_trace considers cumulative retransmission delays. Its code is listed in appendix B.

3.1.2 Test Setup

The test setup is as shown on figure 3.1. TCP and SCTP streams are sent from computer A to computer C through computer B where all computers are running version 2.6.15 of the Linux kernel and using 100

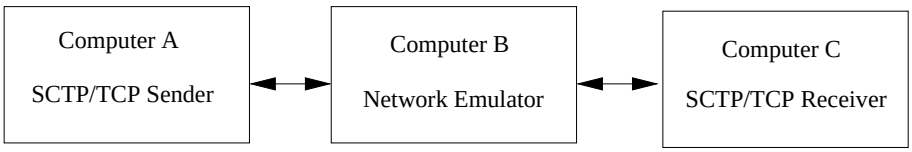

Figure 3.1: Test setup

Mbit/s ethernet cards. The intermediate computer acts as a network emulator and emulates different types of loss scenarios by using Netem.

Both protocols are first tested with thick streams in three loss scenarios with RTTs of 0 ms, 100 ms and 200 ms, respectively. As computer networks span from short distances to long distances, the time between packets are sent and received is varying. Hence, it is necessary to see how retransmission strategies are performing under various RTTs. The various RTTs are emulated by adding delay with Netem on the emulator machine. The RTT of 0 ms is achieved by not adding any packet delay which means the actual delay between the sender machine and the receiver machine, which is close to 0 ms. TCP streams are sent with Netperf by sending as many packets as possible between the sender and receiver machine in a single conection. In each loss scenario, TCP streams are sent plain, that is without any combination of SACK, DSACK and FACK. Then with SACK, SACK + DSACK, SACK + DSACK + FACK and at last SACK + FACK. This results in five different TCP streams in each test scenario. Each test is running for 600 seconds with 5% packet loss emulated by Netem. A 5% packet loss is chosen to trigger enough retransmissions to see the full effect of retransmission strategies during the tests. In an equivalent way, SCTP thick streams are sent with Sctp-perf by sending as many data chunks as possible between the sender and receiver machine in a single connection. To transfer as much data as possible, a message size of 1400 byte is chosen which is close to the MTU of the test network.

Next, both protocols are tested in thin streams in four loss scenarios with RTTs of 0 ms, 100 ms, 200 ms and 400 ms, respectively. The nature of a thin streams could vary and still be classified as a thin stream. The properties that every thin stream have in common is that very few packets are sent with a large time interval between each packet. In addition, the size of a message could be small compared to the network MTU. Thus, in order to simulate a thin, one message of 100 byte is sent every 250 ms by both protocols. TCP streams are sent plain and with the same combinations of SACK, DSACK and FACK as in the thick tests. Each test is running for 1800 seconds with a 5% packet loss. This results in enough

retransmissions to see the full effect of both protocol's retransmission strategies in thin streams.

Both TCP and SCTP streams are dumped by Tcpdump on the sender machine and the retransmissions are later analyzed by Tcptrace and Sctp_trace, respectively.

3.2 SCTP Test results

In this section, the results of the the SCTP thick and thin tests is presented and evaluated. We are evaluating the retransmission delays of data chunks since SCTP transports data in chunks which have their own sequence number. In each test scenario, the cumulative retransmission delay is evaluated. This means that we consider the time between a data chunk is sent and retransmitted for the first time, the time between a data chunk is sent and retransmitted for the second time and so on. This evaluation let us inspect each number of retransmissions and find the minimum and maximum retransmission delay of all data chunks independent of how many retransmissons were needed to retransmit them.

SCTP retransmits data chunks after a fast retransmit and a retransmission timeout. After a retransmission timeout, outstanding data chunks could get bundled into the packet if it is room and thus get retransmitted. Bundling of outstanding chunks could be divided into two retransmisson types to get a better analyzation. The first is the bundling of outstanding data chunks that have been reported missing by SACKs before they are retransmitted. The second is the bundling of outstanding data chunks that are in flight to the receiver where no SACKs have reported them to be lost.

An evaluation of what retransmission types have been used and how many retransmission this type caused ease our evaluation of the retransmission mechanisms and let us see their performance during the tests. If the retransmission delays are evaluated independent of retransmission type, then it is hard to see the effect of the retransmission strategies as several data chunks could get bundled into a single packet during retransmissions. This leads to retransmission delays that could be hard to analyze and understand the reason of. In the worst case, this could lead to no significant difference in retransmission delay at all. By separating retransmission by their type, it will also be possible to see how the distribution of the retransmission types change when the RTT increases and detect the performance of each retransmission type separately. Hence,

Loss Scenario	Retr	Min	Max	Avg
RTT = 0ms	1	0.7	2.5	1.2
	2	1.1	2.8	1.5
	3	1.5	3.2	2.0
	4	2.6	2.9	2.5
	5	2.6	3.3	2.8
RTT = 100ms	1	103.9	312.0	168.9
	2	104.1	312.1	225.8
	3	104.2	416.0	245.8
	4	207.8	312.0	240.8
	5	208.1	311.8	260.0
RTT = 200ms	1	208.0	624.1	338.4
	2	208.1	832.0	454.0
	3	208.2	1040.0	472.6
	4	415.9	1248.0	566.2
	5	416.1	427.9	422.0

Table 3.1: Thick stream: SCTP cumulative retransmission delays

each number of retransmissions is sorted by the minimum, maximum and average retransmission delay of the four retransmission types and shown in tables and plots for each scenario.

3.2.1 Thick stream scenarios

The test results of the SCTP thick streams are shown in table 3.1. The table shows the cumulative retransmission delay of each retransmission, sorted by loss scenario. The corresponding values are shown in figure 3.2. The retransmissions of thick streams are not sorted by their type. Instead, retransmissions caused by fast retransmits and retransmission timeouts are seen as a whole. The reason is that Sctp_trace uses incoming SACKs to determine if it is a retransmission timeout or a fast retransmit based on missing reports in SACK gap ack blocks. This works fine in thin streams as the incoming SACKs arrive more slowly than in thick streams. However, in thick streams, the incoming packet-flow is so large that it is impossible to distinguish between how many SACKs that trigger a retransmission. Sctp_trace assumes that it is a retransmission timeout when it is a fast retransmit and vice versa. Thus, they must be seen as a whole.

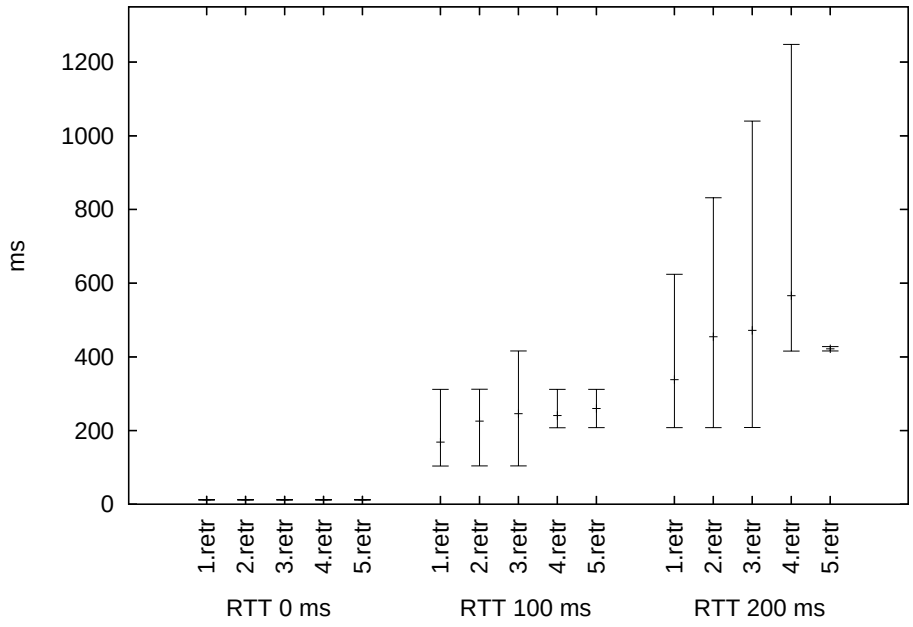

Figure 3.2: Thick stream: SCTP cumulative retransmission delays

RTT = 0 ms

When sending as many data chunks as possible according to the available bandwidth, incoming SACKs controls the number of data chunks injected into the network as there is an exponential growth of the congestion window when SACKs acknowledge outstanding data chunks. When no delay is added with Netem, the RTT constitutes of the network delay between the sender machine and the receiver machine which in this case is almost equal to 0 ms. With an RTT of 0 ms, SACKs need very short time to acknowledge outstanding data chunks and this leads to a fast increase of the congestion window until congestion occurs. With a large congestion window and the fact that many packets are in flight to the receiver at once, several SACKs reporting the lost data chunk are received in very short time.

By looking at the retransmission distribution sorted by each number of retransmissions, the minimum retransmission delay for the first retransmission is 0.7 ms. This is the fastest way of receiving the 4 SACKs needed to fast retransmit a data chunk. The maximum retransmission delay of the first retransmissions is 2.5 ms and is not considerable higher

than the minimum value due to the small RTT.

By looking at the next number of retransmissions, something unexpected is happening. The cumulative retransmission delay of the fifth retransmission of the same data chunks is not considerable higher than the first retransmissions and all the retransmissons are caused by fast retransmits. This behaviour shows that SCTP has no fast recovery mechanism implemented. After a fast retransmit, TCP NewReno does not allow new duplicate ACKs to trigger new fast retransmits until an ACK is received for the earliest retransmitted packet that is not acknowledged yet. This is the main purpose of the fast recovery mechanism. If the fast retransmitted packet is lost, then it have to get retransmitted by a retransmission timeout. If SCTP also should have followed this scheme, the second retransmissions should have been retransmitted over 1000 ms later since the minimum RTO is set to 1000 ms. The low cumulative retransmission delay for the fifth retransmission show that SCTP allows the next SACKs to trigger new fast retransmits independent of each other without receiving acknowledgemts for the previous retransmissions. Since the congestion window could be relatively large in thick streams, there could be many subsequent SACKs reporting the same data chunk to be lost. In these testresults, table 3.1 shows that SACK could trigger up to five fast retransmits of the same data chunk just after each other where each fast retransmit is halving the congestion window. As SCTP allows incoming SACKs to trigger independent retransmissions, all data chunks are retransmitted by a fast retransmit long time before the retransmission timer expires as the RTO minimum value is 1000 ms.

RTT = 100 ms

With an RTT of 100 ms, it will take longer time for SACKs to acknowledge outstanding data chunks. This leads to a slower growth of the congestion window as an exponential growth occurs every RTT until the congestion window is equal to ssthresh or a data chunk is lost. This affects the retransmission delays.

By looking at the distribution of the number of retransmissions of this loss scenario in table 3.1, the lowest cumulative retransmission delay is 103.9 ms and is the fastest way of receiving 4 SACKs and trigger a fast retransmit. To be able to trigger a fast retransmit after RTT ms, the data chunk that triggers the fourth SACK must be sent in the same window as the lost data chunk. If the lost data chunk is sent in the first window and the data chunk that triggers the fourth SACK is sent

in the next window, a fast retransmit is triggered after two RTTs. The maximum retransmission delay of the first retransmission is 312.0 ms. This is three times the size of the RTT and show occurrences where a data chunk sent in the third window trigger the fast retransmit of a data chunk sent in the first window as the windows are very small. In total, this leads to an average retransmission delay of 168.9 ms.

By looking at the distribution of the next number of retransmissions, there are occurrences of fast retransmits that is retransmitting the same data chunk just after each other as in the last scenario. The minimum retransmission delay of a third retransmission is 104.2 ms and is only 0.2 ms above the minimum retransmission delay of the first retransmission and close to the RTT. This proves again that SCTP allows subsequent SACKs to trigger new fast retransmits of the same data chunk without waiting for a SACK acknowledging the first retransmission. In addition, the retransmission timer never have the chance to expire as all subsequent fast retransmits of the same data chunk take care of retransmitting the data chunk a long time before a retransmission timeout occurs.

RTT = 200 ms

With an RTT of 200 ms, the retransmission delay is expected to increase as a result of increasing the RTT compared to the previous scenario. As it takes longer time for SACKs to arrive, it leads to a slower growth of the congestion window.

By looking at the distribution of the number of retransmissions in table 3.1, the lowest cumulative retransmission delay is 208.0 ms and shows the fastes way of receiving 4 SACKs triggering a fast retransmit where the lost data chunk and the data chunk triggering the fourth SACK are sent in the same window. The maximum retransmission delay of a first retransmission is 624.0 ms and show occurrences of data chunks sent in different windows triggers the fast retransmit. In total, this leads to an average retransmission delay of 338.4 ms.

By looking at the distribution of the next number of retransmission, there are also here occurrences of fast retransmits that are triggered just after each other as explained in the previous scenario. The minimum retransmsission delay of a third fast retransmit is only 0.25 ms higher than the minimum retransmission delay of a first retransmission and close to the RTT. In contrast to the previous scenario, the maximum retransmission delay of 1248.0 ms indicates that some data chunks are

retransmitted by a retransmission timeout. If there are multiple loss of the same data chunk in combination with a relatively large RTT where the subsequent fast retransmits fail to retransmit or SACKs are lost, then retransmission timeouts could be triggered. Because of SCTP's way of letting SACKs trigger consequtive retransmissions of the same data chunk, the maximum retransmission delay of a fifth retransmission delay is lower than the maximum retransmission of a fourth retransmission delay.

Summary

When the sender makes use of all the available bandwidth and injects as much data as possible into the network, all retransmissions are triggered by a fast retransmit expect of that some data chunks could get retransmitted by a retransmission timeout in a third retransmission when the RTT is 200 ms. Thus, the fastest way of triggering a retransmission is close to the RTT. One of the reasons that almost all retransmissions are triggered by a fast retransmit is because of the high RTO minimum value used in SCTP. When the RTO minimum value is so high, all retransmissions are supposed to be triggered by a fast retransmit.

The main reason for that almost all retransmissions are triggered by a fast retransmit, is because SCTP retransmits the same data chunk over and over again independent of each other as multiple SACKs arrives and triggers new fast retransmits. As an example, only 0.2 ms could elapse between a first retransmission and a fifth retransmission of the same data chunk. SCTP does not even know if the first retransmission was sucessfully received before starting on a new fast retransmit. Thus, SCTP is handling fast retransmit in an erroneous way and fast recovery is not implemented at all. Although there are multiple loss of packets, the subsequent fast retransmits will take care of retransmitting the lost data chunk long time before a possible retransmission timeout occurs. The worstest thing about this behaviour is that each subsequent fast retransmit cuts the congestion window in half and reduces the throughput drastically when it is unnecessary. In addition, many retransmissions, except for the first retransmissions, are spurious.

3.2.2 Thin stream scenarios

In this section, the test results of the SCTP thin tests are presented and evaluated. The characteristics of a thin stream is that very few packets

Retr	Type	Number	Min	Max	Avg
1.	Retransmission timeout	282	999.1	1256.6	1005.5
	Fast retransmit	24	1024.4	1280.4	1088.373
	B:Reported missing	34	464.0	744.0	592.7
	B:In flight	30	231.8	744.0	274.7
2.	Retransmission timeout	3	1256.0	2000.1	1752.1
	Fast retransmit	7	1279.7	1792.4	1646.0
	B:Reported missing	0	0	0	0
	B:In flight	0	0	0	0
3.	Retransmission timeout	3	2000.1	2000.1	2000.1
	Fast retransmit	0	0	0	0
	B:Reported missing	2	1487.4	1744.1	1615.7
	B:In flight	0	0	0	0

Table 3.2: SCTP cumulative retransmission statistics, RTT = 0 ms

are sent in a large time interval without the need to make use of the available bandwidth and achieve the highest possible throughput. The packets are often small compared to the available payload.

RTT = 0 ms

The retransmission distribution of this scenario is shown in table 3.2. The bundling of outstanding data chunks that are reporting missing is denoted **B:Reported missing** and the bundling of outstanding data chunks that are in flight to the receiver is denoted **B:In flight**.

In this loss scenario, most retransmissions are caused by a retransmission timeout. These are mostly occuring in the first retransmission of a data chunk. In the first retransmission, such retransmissions have a minimum retransmission delay of 999.1 ms and an average retransmission delay of 1005.5 ms. Both values are close to the RTO minimum value used in RTO calculations. The reason for that the minimum retransmission delay is lower than the RTO minimum value could be because of the conversion between jiffies and ms in the kernel that sometimes does not lead to correct values. The value is still close to the RTO minimum value of 1000 ms. The maximum value of 1256.6 ms occurs as late arrivals of SACKs acknowledge some, but not all outstanding data chunks. In these cases, SCTP performs a restart of the retransmission timer and the the expiration time gets further delayed if the earliest outstanding data chunk needs to be retransmitted by a retransmission timeout. Late

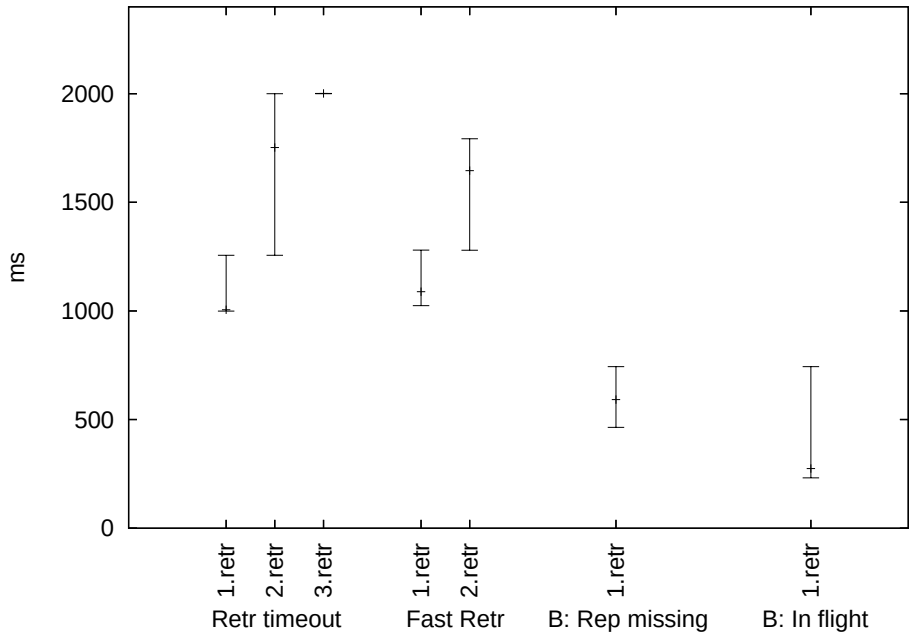

Figure 3.3: SCTP cumulative retransmission delays, RTT = 0 ms

SACK arrivals occurs as packets are sent every 250 ms. If no loss is detected, then the SACK acknowledging the data chunk arrives after 200 ms. During this time, new data chunks could be sent which are outstanding when the SACK arrives and then restarts the timer.

The high ratio of retransmission timeouts is because of the sender has a small chance to receive the fourth SACK needed to trigger a fast retransmit before the data chunk is retransmitted by a retransmission timeout. When the receiver discovers holes in the data chunk sequence, it will each time it receives a new packet immediately send a SACK reporting the lost data chunks until the sequence is restored. As packets are sent every 250 ms and the RTT is 0 ms, the time between a lost data chunk is sent and the sender receives the first SACK reporting the loss will be more than 250ms. This is because the receiver needs new data chunks to discover holes in the sequence. As a consequence it will take more than 1000 ms for the sender to receive 4 SACKs. At this time the data chunk has already been retransmitted by a retransmission timeout.

The results show that a few data chunks are still retransmitted by a fast retransmit in its first retransmission. This occurs when the expiration time of the retransmission timer is delayed by an intermediate

restart caused by late arrivals of SACKs. If a SACK delays the timer, then 4 SACKs have time to arrive and trigger a fast retransmit of the data chunk before the retransmission timer expires again. The maximum retransmission delay of 1280.4 ms shows that fast retransmits can have a relatively high delay as the timer is delayed. An average retransmission delay of 1088.4 ms shows the average time of receiving 4 SACKs and then trigger a fast retransmit. The minimum retransmission delay of 1024.4 is the fastest way of receiving 4 SACKs.

In special cases, fast retransmits and retransmission timeouts are triggered independently of each other. This mostly happens when data chunks are retransmitted two or three times in the absence of a SACK confirming that the data chunks are sucessfully received. A data chunk is not fully acknowledged and removed from the retransmisson queue until the cumulative TSN of a SACK passes the TSN of the data chunk. SCTP will continue to retransmit data chunks independet of retransmission type until the data chunk is acknowledged, either by a gap ack field or by the cumulative TSN in a SACK. Since there are large intervals between each sent packet, this leads to large intervals between the incoming SACKs.

The test results show that data chunks are retransmitted the second time by retransmission timeouts down to 1256.0 ms. At this point, the data chunk is first retransmitted by a fast retransmit, before it is retransmitted by a delayed retransmission timeout at 1256.0 ms for the second time independently of the fast retransmit. This also indicates that the timer is not restarted after a fast retransmit and that the retransmission timer and the fast retransmit mechanisms are triggered independently of each other.

The results of the second retransmission also show that the opposite could happen. The average retransmission delay of 1792.4 ms show occasions where the data chunk is first retransmitted by a retransmission timeout and fast retransmitted the second time. If the retransmitted data chunk is lost, then 4 SACKs arrives in time to trigger a second retransmission of the same chunk almost 792 ms after the retransmission timeout. This indicates that it is occurrences of multiple loss, it is difficult to know which retransmissions were necessary or not.

By looking at third retransmissions, there are one data chunk that is retransmitted by a retransmission timeout after 2000.1 ms. A cumulative retransmission delay of 2000.1 ms indicates that this is the second retransmission timeout as the value is twice the minimum RTO. This means that the second retransmission was triggered by an intervening fast retransmit. This also confirms that the timer is not restarted after a

Retr	Type	Number	Min	Max	Avg
1.	Retransmission timeout	275	1039.9	1612.1	1049.8
	Fast retransmit	23	1126.5	1386.2	1173.1
	B:Reported missing	27	460.0	1356.1	689.3
	B:In flight	314	15.3	532.0	51.2
2.	Retransmission timeout	17	1152.1	2048.1	1750.9
	Fast retransmit	8	1129.0	1896.2	1800.3
	B:Reported missing	4	1016.0	1528.1	1209.0
	B:In flight	0	0	0	0
3.	Retransmission timeout	7	2040.1	2048.1	2043.5
	Fast retransmit	2	1895.6	2919.7	2407.7
	B:Reported missing	0	0	0	0
	B:In flight	1	1276.0	1276.0	1276.0
4.	Retransmission timeout	1	3040.1	3040.2	3040.2
	Fast retransmit	0	0	0	0
	B:Reported missing	1	2016.1	2016.1	2016.1
	B:In flight	0	0	0	0

Table 3.3: SCTP cumulative retransmission statistics, RTT = 100 ms

fast retransmit.

The results show that 16.7% of the retransmitted data chunks are bundled with data chunks retransmitted by a retransmission timeout. As the SCTP packets contain small data chunks and are sent every 250 ms before they are retransmitted, there will be room for bundling of several data chunks as long as the total size of the packet does not exceed the network MTU. Results from the first retransmission show that bundled outstanding data chunks are retransmitted faster than data chunks retransmitted by a retransmission timeout or a fast retransmit. This is as expected since the average retransmission delay for retransmission timeouts is so high. During this time, one or more SACKs could arrive and report loss of some data chunks before a retransmission timeout. With an average retransmission delay of 592.7 ms, this leads to faster retransmissions of the bundled data chunks that are actually reported missing. Bundled data chunks that are in flight to the receiver may already have been sucessfully received. But since they are bundled, they will not increase the number of packets in the network.

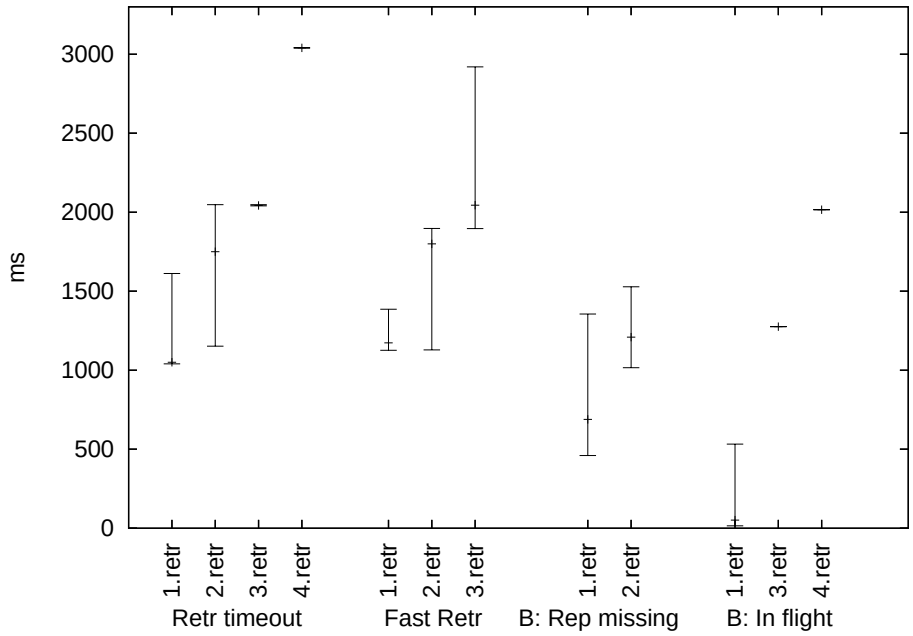

Figure 3.4: SCTP cumulative retransmission delays, RTT = 100 ms

RTT = 100 ms

The retransmission distribution of this loss scenario is shown in table 3.3. A plot of the same corresponding values is shown in figure 3.4 In this loss scenario, 300 retransmissions are caused by a retransmission timeout. In comparison, only 33 retransmissions are caused by a fast retransmit.

By looking at first retransmissions of data chunks, the average retransmission delay of retransmission timeouts is 1049.8 ms. This is close to the minimum value of 1039.9 ms. With an increase in RTT of 100 ms compared to the last scenario, the RTO calculation does not result in an RTO higher than the minimum RTO since the RTO minimum value is so high compared to the RTT. Thus, the calculated RTO used in this scenario is 1000 ms. The reason for the delayed retransmission timeouts are restarts of the retransmission timer caused by late SACKs as described in the last section. Since the RTT is increasing, SACKs are arriving later and lead to later restarts of the timer. The maximum retransmission delay for retransmission timeouts is 1612.1 ms. This occurs when one or more SACKs is lost and one of the next SACKs is arriving relatively

late without acknowledging the data chunk and restarts the timer before it expires.

As the SACKs are arriving in late intervals, multiple SACK loss could lead to further displacement of the expiration time of the retransmission timer. In such cases, a data data chunk could first get retransmitted by a fast retransmit although it does not happen often compared to retransmission timeouts. An average retransmission delay of 1173.1 shows that 4 SACKs can trigger fast retransmits before the retransmission timer expires.

By looking at the second retransmissions, one could see that the minimum retransmission timeout delay is 1152.1 ms. This occurs when data chunks is retransmitted by a fast retransmit the first time and then retransmitted by a delayed retransmission timeout the second time. The retransmission timer is triggered independently of and just after the fast retransmit. There are a few fast retransmits with an average retransmission delay of 1896.2 ms. These occurs when data chunks are retransmitted by a retransmission timeout the first time and fast retransmitted the second time. If the first retransmission is lost, then the SACKs are sent without delay and arrives in time to trigger the second retransmission before the retransmission timer expires again. The maximum values of the third and fourth retransmissions indicates that multiple loss of the same data chunk has occured, but it is hard to know which are spurious or not.

Almost 50% of the retransmissions are bundled with data chunks retransmitted by a retransmission timeout. With an increase in the RTT, SACKs will use longer time to arrive and report lost data chunks. This leads to an average retransmission delay of 689.3 ms and results in faster retransmissions of bundled data chunks that are actually reported missing despite that it is not many of them. Most bundled data chunks are data chunks that are in flight to the receiver and may already have been sucessfully received.

RTT = 200 ms

The retransmission distribution is shown in table 3.4 and figure 3.5. In this loss scenario, 290 retransmissions are caused by a retransmission timeout. This is still large compared to the number of fast retransmits (46).

By looking at the first retransmission of data chunks, one can see that

Retr	Type	Number	Min	Max	Avg
1.	Retransmission timeout	266	996.2	1460.1	1144.6
	Fast retransmit	35	1228.4	1740.7	1274.2
	B:Reported missing	24	487.9	976.0	780.7
	B:In flight	338	28.0	888.0	172.8
2.	Retransmission timeout	16	1460.1	2144.1	1672.1
	Fast retransmit	11	1230.2	1999.6	1835.6
	B:Reported missing	0	0	0	0
	B:In flight	0	0	0	0
3.	Retransmission timeout	8	2144.0	2144.1	2144.1
	Fast retransmit	0	0	0	0
	B:Reported missing	3	1376.1	1888.1	1546.7
	B:In flight	0	0	0	0

Table 3.4: SCTP cumulative retransmission delays, RTT = 200 ms

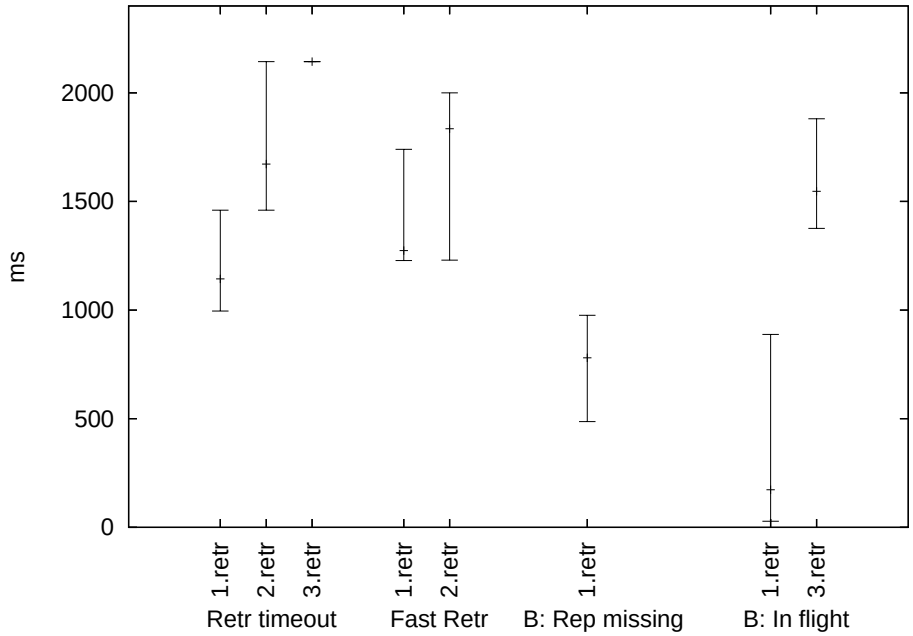

Figure 3.5: SCTP cumulative retransmission delays, RTT = 200 ms

the average retransmission delay for retransmission timeouts is 1114.6. The RTT has increased since the last scenario, but the calculated RTO is still to low compared to the minimum RTO. Thus, the RTO calculations always result in an RTO of 1000 ms. The results show an increase of 100 ms in average retransmission timeout delay caused by retransmission timeouts compared to the last scenario. Since the RTT is increasing, then intervening SACKs could arrive later and restart the timer. The minimum value of 1000 ms shows an example of a retransmission timeout that occurs when no SACKs restarts the timer. In addition, the maximum retransmission delay of 1460 ms shows what is happening if the retransmission timer is restarted late as one or more intervening SACKs get lost and the timer is restarted by the next SACK that arrives.

The fast retransmits occurs as a result of the restart of the retransmission timer with an average retransmission delay of 1274.2 ms. If a data chunk is sent just after a retransmission timeout, then the timer is set to twice the old RTO because of the following exponential backoff. The maximum fast retransmit retransmission delay is 1740.7 ms and is a result of this occurrence.

The second retransmissions show occurrences of retransmission timeouts with an average retransmission delay of 1672.1 ms. This occurs when the data chunk is first retransmitted by a fast retransmit and then retransmitted by a delayed retansmission timeout. The third retransmissions of data chunks consists of retransmission timeouts only. With an average retransmission timeout of 2144.0 ms, this is the second retransmission timeout where the data chunk has been retransmitted by a fast retransmit the second time.

Almost 50% of the data chunks are retransmissions of bundled outstanding data chunks. 3.85% consists of data chunks that are actually reported missing. With an average retransmission delay of 780.7 ms for such retransmissions, outstanding data chunks that are reported missing are still retransmitted faster than data chunks retransmitted by a retransmission timeout or a fast retransmit. There is an increase in the average retransmission delay of such data chunks compared to the last scenario which is a result of a higher RTT.

RTT = 400 ms

The retransmission distribution is shown in table 3.5 and figure 3.6. In this loss scenario, 29.93% of the data chunks are retransmitted by a retransmission timeout.

Retr	Type	Number	Min	Max	Avg
1.	Retransmission timeout	242	1343.0	1660.1	1352.0
	Fast retransmit	31	1427.2	1943.6	1496.2
	B:Reported missing	26	780.0	1430.1	1011.1
	B:In flight	567	11.8	832.0	213.4
2.	Retransmission timeout	21	1496.1	2348.1	1897.3
	Fast retransmit	11	1940.6	2202.0	2174.6
	B:Reported missing	2	1320.1	2088.1	1704.1
	B:In flight	1	1444.0	1444.0	1444.0
3.	Retransmission timeout	10	2344.1	2344.1	2344.1
	Fast retransmit	0	0	0	0
	B:Reported missing	0	0	0	0
	B:In flight	1	2088.1	2088.1	2088.1

Table 3.5: SCTP cumulative retransmission statistics, RTT = 400 ms

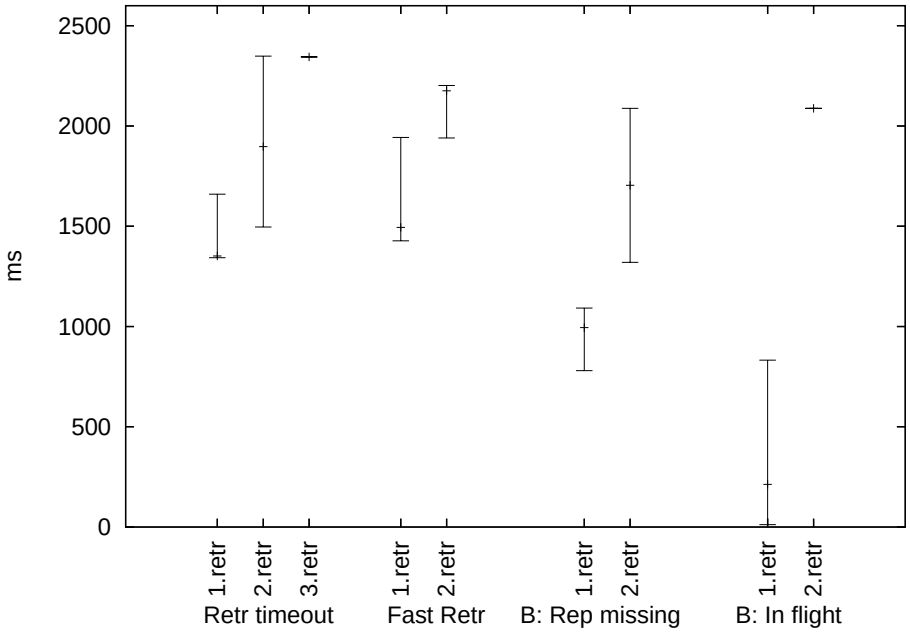

Figure 3.6: SCTP cumulative retransmission delays, RTT = 400 ms

By looking at the first retransmissions of data chunks, the average retransmission delay for retransmission timeouts are 1352.0 ms. An RTT of 400 ms is still to small to make the RTO exceed the RTO minimum value. Thus, the calculated RTO is always resulting in an RTO value of 1000 ms. An increase in RTT results in later arrivals of SACKs which leads to later restarts of the retransmission timer. These occurrences reflects that the retransmission timer is further delayed since the last scenario. As a consequence, some few fast retransmits can occur with an average retransmission delay of 1496.2 ms. The maximum retransmission delay of fast retransmits is 1943.6 ms. The retransmission timer is unlikely to be delayed that much, but a fast retransmit can happen if the retransmitted data chunk was sent just after a retransmission timeout, using the exponential RTO. The second and third retransmissions behaves in the same manner as in the previous tests.

Almost 3% of the data chunks are bundled data chunks that are reported lost. With an average retransmission delay of 1011.1 ms, this data chunks get retransmitted faster compare to data chunks retransmitted by a retransmission timeout or a fast retransmit despite there are few of them. As many as 62.4% of the retransmissions are bundled data chunks that are in flight to the receiver. With an increase in RTT, SACKs are arriving later and more data chunks could be in flight when the retransmission timeouts occur. With an average retransmission delay of 213.4 ms this is small compared to the others, but they may already have been successfully received as no missing reports are given.

Summary

In the thin streams used in the loss scenarios, almost all of the retransmissions are triggered by retransmission timeouts compared to fast retransmits. This occurs as 4 SACKs do not have the chance to arrive before the retransmission timer expires. When the RTT increases, the test results show that the retransmission timer can be considerably delayed by late SACKs which restarts the timer. In such situations, 4 SACKs may have the chance to arrive and trigger fast retransmits before a retransmission timeout occurs.

The results of the second and third retransmissions show that retransmission timeouts and fast retransmits are triggered independently of each other until the data chunk is acknowledged. This also show that the Linux kernel implementation does not follow the timer rules after a fast retransmit as the timer is not restarted after a fast retransmit of

the earliest outstanding data chunk. Thus, SCTP allows SACKs to trigger fast retransmits after a retransmission timeout without ignoring the next SACK until an acknowledgment for the first retransmission is received. In the same way, if no new SACKs acknowledge the data chunk, the timer is triggered just after a fast retransmit. This leads to an erroneous retransmission behaviour where it is hard to know what retransmissions are spurious or not. By looking at the average retransmission delays of the first retransmission timeouts and fast retransmits, they all have an average retransmission delay above 1000 ms. In addition to the RTT, a 1000 ms lag in an interactive application should be noticeable by the users.

There are few occurrences of bundling of outstanding data chunks that are reported missing. This leads to smaller retransmission delays for such data chunks. When the RTT increases, more than half of the retransmitted data chunks are bundled data chunks that are in flight to the receiver, but these may already have been successfully receieved. The average retransmission delay in all scenarios lies around 1000 ms which is further displaced when the RTT increases. Bundling of outstanding data chunks decrease this value. The maximum retransmission delay from a data chunk is sent and retransmitted for the last time is 3000 ms for all tests. After a retransmission timeout, the congestion window is set to 1 MTU. As the message size used in the tests is 100 bytes, this allows new data chunks to be sent after a retransmission timeout. If multiple loss occurs, these data chunks could trigger new fast retransmits independent of the the timer. These data chunks could generate new RTO calculations which collapse the exponential RTO value down to the normal value. Thus, in the test scenarios, exponential backoffs rarely occur and play a minor role in raising the retransmission delays.

3.3 TCP NewReno test results

In this section, the test results of TCP NewReno are presented and evaluated. It is important to notice that the representation of the retransmission delays differ from SCTP. In the TCP NewReno results, the retransmission delay between consequtive retransmissions is considered. The reason is that Tcptrace was the only alternative to analyze TCP retransmission delays. Because of SCTP's inconsistet handling of retransmissions, it would be impossible to analyze the retransmission delays without considering cumulative retransmission delays.

Loss Scenario	Extension	Retransmission Delay Statistics			
		Min	Max	Avg	Std
RTT = 0 ms	Plain	0.5	2997.1	46.6	113.5
	S	0.7	13608.2	42.1	188.3
	S+D	0.7	6912.4	40.1	121.0
	S+D+F	0.5	3238.4	36.9	108.0
	S+F	0.5	3456.2	34.6	98.8
RTT = 100 ms	Plain	104.0	5405.5	251.6	227.9
	S	104.0	3264.2	227.6	198.1
	S+D	104.0	11912.7	239.2	413.7
	S+D+F	104.0	1260.0	214.6	118.1
	S+F	104.0	2448.0	231.5	164.6
RTT = 200 ms	Plain	208.0	2088.1	443.4	227.7
	S	208.0	1880.1	395.1	164.8
	S+D	208.0	3138.8	400.1	217.0
	S+D+F	208.0	1672.0	366.5	154.6
	S+F	208.0	1680.1	375.1	160.8

Table 3.6: Thick streams: TCP NewReno retransmission delays

3.3.1 Thick stream scenarios

The results of the TCP NewReno thick streams is presented in table 3.3.1. A plot of the average retransmission delays for TCP NewReno thick streams is shown in figure 3.7. In both table and figure, **P** is abbreviation for plain, **S** is the abbreviation for SACK, **D** is the abbreviation for DSACK and **F** is the abbreviation for FACK.

RTT = 0 ms

When the sender sends as much data as possible, the number of received ACKs control the number of packets that are sent. When all packets in a window are acknowledged, TCP NewReno performs an exponential growth of the congestion window during slow-start. If a packet is lost, then TCP NewReno waits for three duplicate ACKs before starting fast retransmit and going into fast recovery. Otherwise, if fast retransmit is not triggered, a retransmission timeout occurs and TCP NewReno goes back to slow-start and retransmits the lost packet. With an RTT of 0 ms, an ACK acknowledging a packet arrives in short time and leads to a fast growth of the congestion window. In the same way, three duplicate ACKs arrive in short time and trigger a fast retransmit. In this scenario,

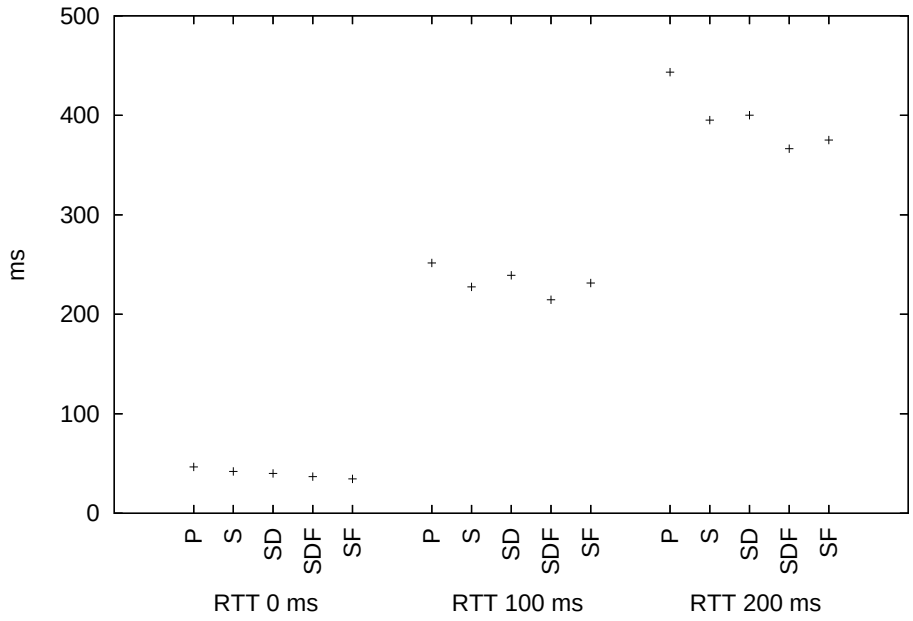

Figure 3.7: Thick streams: TCP NewReno avg retransmission delays

this leads to a behaviour where almost all packets are retransmitted by a fast retransmit and the minimum retransmission delay of 0.5 ms shows the fastest way of receiving three duplicate ACKs. As TCP NewReno enters fast recovery after a fast retransmit, it will retransmit one packet per RTT although more packets are lost at once. This means that it ignores subsequent duplicate ACKs after the first retransmission until the packet is acknowledged.

The maximum retransmission delays vary between 2997.1 ms and 13608.2 ms. These occur because of multiple loss of the same packet where three duplicate ACKs did not trigger a fast retransmit the first time or the fast retransmitted packet is lost. If a fast retransmitted packet is lost, it must be retransmitted by a retransmission timeout. After a retransmission timeout, the congestion window is set to 1 SMSS. This only allows the retransmitted packet to be sent. If duplicate ACKs are received, they are ignored until the ACK for the first retransmitted packet arrives. Thus, multiple loss of the same packet leads to subsequent retransmission timeouts where each is growing exponentially. As the average retransmission delays for the various streams are close to 40 ms, these high retransmission delays affect the results and lead to a small increase, but the average retransmission delay is still so small that exponential

backoffs occur often. Since duplicate ACKs are received in short time, they will mostly take care of retransmitting the packet before the timer expires.

Plain TCP NewReno has an average retransmission delay of 46.6 ms The results show that streams sent with the various combinations of SACK, DSACK and FACK lead to a decrease of 4 to 10 ms, compared to plain TCP NewReno. This confirm that the mechanisms are working propely when the streams are thick, as expected. No significant difference between the various SACK mechanisms can be seen. During this test, it is therefore hard to determine which of them is the best.

RTT = 100 ms

With an RTT of 100 ms, it takes 100 ms to perform an exponential growth of the congestion window as 100 ms is the time needed to receive all ACKs acknowledging the packets sent in a window. In the same way does it take 100 ms to receive the three duplicate ACKs needed to trigger a fast retransmit.This leads to the minimum retransmission delay of 104.0 ms which isalmost equal to the RTT and the shortest time to receive three duplicate ACKs. If several packets are lost, TCP NewReno ignores the following duplicate ACKs until the first packet is acknowledged. Hence, it can only fast retransmit one packet per RTT and the retransmission delay will therefore not get below the RTT as in SCTP.

The maximum retransmission delays from 1260.0 ms to 11912.7 ms reflects several retransmission timeouts of the same packet and the following exponential backoffs. With an RTT of 100 ms, this results in a calculated RTO of 300 ms. If the packet is not acknowledged in 300 ms, then it will be retransmitted by a retransmission timeout. If a packet is sent in the current window and the third duplicate ACK is caused by a packet sent in the second window, then the fast retransmit will get delayed by twice the RTT. These occurrences, in addition to exponential backoffs of the retransmission timer, do both raise the average retransmission delays considerably.

The average retransmission delay for plain TCP NewReno is 251.6 ms. The various combinations of SACK, DSACK and FACK decrease the average retransmission delay. These delays span from 214.6 ms to 239.6 ms and is a small reduction compared to plain TCP NewReno, as expected.

RTT = 200 ms

With an RTT of 200 ms, the minimum retransmission delay is 208.0 ms. This reflects the fastest way of receiving three duplicate ACKs and trigger fast retransmits. As in the last scenario, exponential backoffs of the retransmission timer occurs and lead to the maximum retransmission delays from 1672.0 ms to 3138.8 ms. With an RTT of 200 ms, the minimum calculated RTO equals 400 ms. If a fast retransmitted packet is lost, then it must be retransmitted by a retransmission timeout. In addition, duplicate ACKs could get triggered by packets sent in two different windows. This leads to a considerable raise of the average retransmission delay, but fast retransmits will take care of most retransmissions.

The average retransmission delay for plain TCP NewReno is 443.4 ms. The various combinations of SACK, DSACK and FACK do also here decrease the average retransmission delay. With average retransmission delays from 366.5 ms to 400.1, it is a reduction compared to plain TCP NewReno, as expected.

Summary

When the stream is as thick as possible, almost all retransmissions are triggered by a fast retransmit and the results show that three duplicate ACKs are mostly received in the time of the RTT as the RTO value is 200 ms higher than the RTT. There are still some retransmission timeouts with occurrences of exponential backoffs. In addition to delayed fast retransmits where the duplicate ACKs are triggered by packets sent in two different windows, this leads to a considerable raise of the average retransmission delays. The various combinations of SACK, DSACK and FACK are decreasing the average retransmission delays compared to plain TCP NewReno, as they are developed to do.

3.3.2 Thin stream scenarios

The results of the TCP NewReno thin streams is presented in table 3.7. A plot of the average retransmission delays of TCP NewReno thin streams is shown in figure 3.8.

Loss Scenario	Extension	Retransmission Delay Statistics			
		Min	Max	Avg	Std
RTT = 0 ms	Plain	203.6	1632.1	231.7	98.6
	S	203.2	816.1	224.5	72.0
	S+D	202.9	1632.1	233.4	101.9
	S+D+F	200.1	1632.1	234.6	108.9
	S+F	200.1	1632.1	225.2	87.8
RTT = 100 ms	Plain	308.1	1216.1	328.3	97.7
	S	308.1	1264.1	348.5	113.9
	S+D	308.1	11185.2	388.4	554.4
	S+D+F	308.1	9816.6	360.3	378.4
	S+F	308.1	16901.0	392.6	708.7
RTT = 200 ms	Plain	412.1	6614.4	481.6	305.1
	S	412.1	3328.2	488.2	277.7
	S+D	412.1	3360.2	461.1	180.5
	S+D+F	412.1	2752.1	464.6	179.0
	S+F	412.1	5912.4	487.3	404.5
RTT = 400 ms	Plain	612.1	4960.3	728.4	437.2
	S	612.1	2842.5	692.5	264.2
	S+D	612.1	2480.2	693.0	239.4
	S+D+F	612.1	2480.2	708.7	286.2
	S+F	612.1	2480.2	697.8	246.1

Table 3.7: Thin streams: TCP NewReno retransmission delays

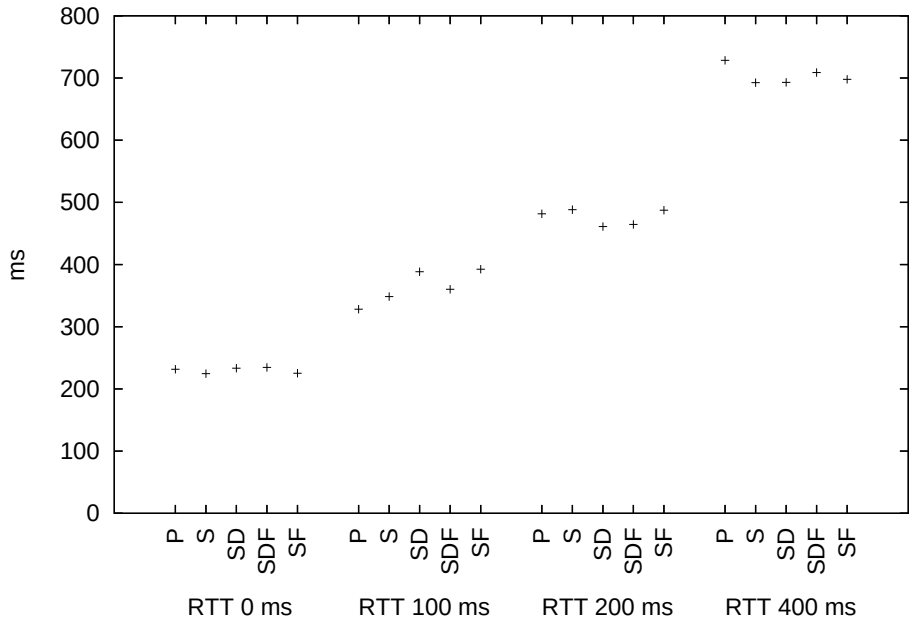

Figure 3.8: Thin streams: TCP NewReno avg retransmission delays

RTT = 0 ms

In this loss scenario, all retransmissions will be triggered by a retransmission timeout. Since the RTT is close to 0 ms, the retransmission timer will mostly expire at the TCP minimum RTO of 200 ms except for when an exponential backoff occurs just before the packet is sent.

When packets are sent every 250 ms, there will take at least 250 ms before the receiver discovers a lost packet as it needs new packets to discover a hole in the sequence. Hence, it takes at least 250 ms to trigger the first duplicate ACK when the RTT is close to 0. As the RTO value is 200 ms in this loss scenario, the packet has already been retransmitted by a retransmission timeout. This explains the minimum retransmission delays of this scenario where all are just above 200 ms which is equal to the RTO. With a RTT close to 0, the packet is already successfully received and acknowledged before the receiver has the chance to discover the packet loss. The first duplicate ACK will never be sent. If the first packet is lost, then the third duplicate ACK is sent after 750 ms. Thus, there is not enough time to receive a triple duplicate ACK that will trigger a fast retransmit before a retransmission timeout. When a packet

is retransmitted by a retransmission timeout, a new packet is not sent before the retransmitted packet is acknowledged. The reason for this is that the packet is retransmitted in the start of the following slow-start and an acknowledgment is needed to increase the congestion window in order to send more packets. In addition, TCP enters a loss state when a packet is retransmitted by a retransmission timeout. This means that it ignores subsequent duplicate ACKs until the packet is acknowledged.

In the same way, the various combinations of SACK, DSACK and FACK give no improvements as SACKs reporting loss is not received before the packet has been retransmitted by a retransmission timeout. This can be seen in the results as there is no significant difference between the average retransmission delays when running plain NewReno or with the various combinations of SACK, DSACK and FACK. For all streams, the average retransmission delays lie between 224.5 ms and 234.6 ms. The maximum retransmission delays span from 816.1 ms to 1632.1 ms and reflect several exponential backoffs of the retransmission timer during multiple loss of the same packet. This occurs if the packet is first retransmitted by a retransmission timeout and then gets lost between subsequent retransmissions. TCP enters a loss state when a packet is retransmitted by a retransmission timeout. This explains the high maximum retransmission delays.

The average retransmission delays are close to the minimum retransmisson delays and show that most times packets are not victim to exponential timeouts. The small difference in the average retransmission delays are caused by how Netem drops packets. Netem drops 5% of the packets, but the drop distribution could vary.

RTT = 100 ms

In this scenario, all retransmissions are triggered by a retransmission timeout as in the last scenario. With an increase in the RTT of 100 ms compared to the last scenario, it takes longer time between a packet is sent and a duplicate ACK or a SACK reporting lost packets can be received. With an RTT of 100 ms and absence of exponential backoffs, the retransmission timer expires at approxemately 300 ms according to the increased RTT's influence on RTO calculations. The results show that plain NewReno and the various combinations of SACK, DSACK and FACK have a minimum retransmission delay of approxemately 300 ms. This value reflects the minimum calculated RTO when the RTT is 100 ms as duplicate ACKs or SACKs never have the time to trigger fast re-

transmits before the timer expires. There are no significant difference in the average retransmission delays between the various streams as they all lie between 328.3 ms and 392.6 ms. The difference between the lowest and highest average value is due to Netems emulating of packet loss that could vary in each test. The values are expected since the RTO is increasing as result of an increase in the RTT. The maximum retransmission delays span from 1216.1 ms to 16901.2 ms and are results of several exponential backoffs as explained in the last scenario. Since the average retransmission delays for all streams are around 350 ms and close to the minimum value of 300 ms, exponential backoffs do rather here occur often enough to make a big influence on the average retransmission delays.

RTT = 200 ms

With an RTT of 200 ms, the retransmission timer uses a RTO of 400 ms as a result of the RTO calculations. This is the minimum RTO + the RTT which results in a calculated RTO of 400 ms. This reflects the minium retransmission delays of 412.1 ms which is close to the calculated RTO. As in the last scenario, most retransmissions are triggered by a retransmission timeout as duplicate ACKs or SACKs never have the chance to trigger fast retransmits. The average retransmission delays lie between 461.0 ms and 487.3 ms and there are rather here no significant difference between the retransmission delays for the various streams. The maximum retransmission delays span from 2752.1 ms to 6614.4 ms and is a result of several exponential backoffs as explained before. The average retransmission delay is still close to the minimum retransmission delays and show that exponential backoffs do not play a major role on the average retransmission delays.

RTT = 400 ms

With an RTT of 400 ms, the calculated minimum RTO in this scenario is 600 ms. As the RTT is increasing and the stream is still very thin, duplicate ACKs or SACKs have little chance to get received before the packet is retransmitted by a retransmission timeout and successfully received. The results show no significant difference in the average retransmisson delays which lie between 692.5 ms and 728.4 ms, as expected. The minimum retransmission delay of 612.1 ms, equal for each stream, is reflecting the minimum RTO. The maximum retransmission delays lie between 2480.5 to 4960.3 ms and show several exponential backoffs of

the retransmission timer. The lowest average retransmission delay is 660.3 ms which is close to the minimum retransmission delay of 612.1 ms. Although subsequent exponential backoffs play a larger role on the retransmission delays, they are neither here making a big influence on the results.

Summary

In thin streams, all retransmissions are triggered by a retransmission timeout without any significant difference between the various streams. This is because triple duplicate ACKs or SACKs reporting loss have no chance to arrive before the packet is retransmitted by a retransmission timeout. The maximum retransmission delays reflects several subsequent exponential backoffs as a consequence of multiple loss of the same packet. In such cases, the TCP ignores further duplicate ACKs. Because of the following slow-start, a new packet is not sent before the retransmitted packet has been acknowledged. Exponential backoffs are the main reason for a higher average retransmission delay compared to the minimum value that is also high because all retransmissions are triggered by a retransmisson timeout.

3.4 Comparision and evaluation

3.4.1 Thick streams

SCTP has two ways to trigger a retransmission of a data chunk; after a retransmission timeout with a minimum RTO of 1000 ms and after the retrieval of 4 SACKs that trigger a fast retransmit. In addition, SCTP bundles the earliest outstanding data chunks after a retransmission timeout as long as there is room in the packet according to the network MTU. In thick streams with a reasonable RTT, all data chunks are retransmitted by a fast retransmit. The reason for this behaviour is that the RTO minimum value is high. In addition, SCTP does not ignore the following SACKs reporting the loss of the retransmitted data chunk after a fast retransmit or a retransmission timeout. If many packets are sent in a window and one of the first is lost, this may lead to frequent occurrences of subsequent fast retransmits. As each fast retransmit is halving the congestion window, this leads to a drastic reduction of throughput and waste of bandwidth when this is unnecessary.

```
[87467.256] TSN: 6344
[87467.275] TSN: 6345
[87466.957] SACK CUM ACK: 6328 #GAP ACKs: 0
[87467.298] TSN: 6346
[87467.312] TSN: 6347
[87466.962] SACK CUM ACK: 6330 #GAP ACKs: 0
[87467.338] TSN: 6348
[87466.981] SACK CUM ACK: 6332 #GAP ACKs: 0
[87467.537] TSN: 6349
[87467.581] TSN: 6350
[87468.400] TSN: 6351
[87570.891] SACK CUM ACK: 6334 #GAP ACKs: 0
[87571.009] TSN: 6352
[87571.027] TSN: 6353
[87570.896] SACK CUM ACK: 6336 #GAP ACKs: 0
[87571.076] TSN: 6354
[87571.090] TSN: 6355
[87570.918] SACK CUM ACK: 6338 #GAP ACKs: 0
[87571.113] TSN: 6356
[87571.126] TSN: 6357
[87570.924] SACK CUM ACK: 6340 #GAP ACKs: 0
[87571.148] TSN: 6358
[87571.161] TSN: 6359
[87570.942] SACK CUM ACK: 6342 #GAP ACKs: 0
[87571.184] TSN: 6360
[87571.198] TSN: 6361
[87570.948] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6345
[87571.221] TSN: 6362
[87571.235] TSN: 6363
[87570.969] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6346
[87571.255] TSN: 6364
[87570.975] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6347
[87571.273] TSN: 6365
[87570.999] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6348
[87571.294] RETRANSMISSION:
          R1 6344 (104.038ms) Fast retransmit
[87571.005] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6349
[87571.041] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6350
[87674.900] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6352
[87674.906] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6353
[87675.173] RETRANSMISSION:
          R2 6344 (207.917ms) Fast retransmit
[87674.910] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6354
[87674.927] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6355
[87674.931] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6356
[87674.954] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6357
[87675.228] RETRANSMISSION:
          R3 6344 (207.972ms) Fast retransmit
[87674.959] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6358
[87674.978] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6359
[87674.983] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6360
[87675.006] SACK CUM ACK: 6343 #GAP ACKs: 1 GAP ACKED: 6345-6361
[87675.274] RETRANSMISSION:
            R4 6344 (208.018ms) Fast retransmit
```

Figure 3.9: Consequtive fast retransmits of the same data chunk in SCTP

The only positive thing about this behaviour is that the retransmission delays get very low, but most of them are spurious. Figure 3.9 shows a snapshot provided by Sctp_trace which documents SCTP's subsequent fast retransmits. In this snapshot, the data chunk with TSN 6344 is retransmitted three times as incoming SACKs triggers new fast retransmits without knowing if the data chunk has been successfully received or not. By looking at the cumulative retransmission delay between the third and second retransmission, there is only 0.04 ms between the retransmissions. As each fast retransmit is halving the congestion window, this leads to a drastic reduction in throughput.

In thick streams, most TCP NewReno retransmissions are triggered by 3 duplicate ACKs or SACKs. In contrast to SCTP, TCP NewReno ignores the following duplicate ACKs until the packet is acknowledged. If multiple packets are lost, each packet will get retransmitted per RTT in fast recovery. If a fast retransmitted packet gets lost, then it must be retransmitted by a retransmission timeout. Therefore, more packets are retransmitted by a retransmission timeout compared to SCTP. Since the RTO minimum value in TCP NewReno is lower than in SCTP, this does not lead to high retransmission delays unless there are exponential backoffs of the retransmission timer. The usage of the various combinations of SACK, DSACK and FACK improves the retransmission delays, as expected. But it is hard to tell which of them is the best.

3.4.2 Thin streams

When streams get thin, SCTP has major problems retransmitting data in a reasonable time. In most situations almost all retransmisisons are triggered by a retransmission timeout unless it is delayed by late restarts and 4 SACKs have the chance to trigger a fast retransmit before the timer expires. With a minimum RTO of 1000 ms, this leads to average retransmission delays above 1000 ms. In comparision, most TCP NewReno retransmissions are triggered by a retransmission timeout as 3 duplicate ACKs never have the chance to be sent before the packet is retransmitted by a retransmission timeout. The various combinations of SACK, DSACK and FACK mechanisms does not make any difference as there are to few packets and SACKs in thin streams to make them work properly.

SCTP's minimum RTO is five times higher than TCP NewReno's minimum RTO. The average retransmission delays of SCTP is therefore considerably higher than the average retransmission delays of TCP NewReno.

In addition does TCP NewReno's take late restarts of the retransmission timer into account when calculating a new RTO which lead to more stable RTO calculations although ACKs arriving late. In contrast, SCTP RTO calculations do not take late restarts of the retransmission timer into account which in many situations lead to delayed expirations of the retansmission timer in thin streams.

As TCP NewReno ignores the following duplicate ACKs until the packet is acknowledged while in fast recovery or after a retransmission timeout, it is a victim of several exponential backoffs which leads to high retransmission delays during multiple loss of the same packet. After subsequent retransmission timeouts, the exponential RTO could be extremely high, and is very noticeable by users of an interactive application.

In contrast, SCTP allows new data chunks to be sent after a retransmission timeout as the congestion window is set to 1 MTU and each data chunks is 100 bytes + size of headers. After the exponential backoff, this makes it possible to perform new RTT measurements and calculate a new RTO. If the timer is restarted by a late SACK, then the the newly calculated RTO is used in the following restart of the timer and collapses the exponential backoff. In addition, SCTP allows incoming SACKs to trigger new fast retransmits after the first fast retransmit or retransmission timeout without waiting for an acknowledgement of the data chunk in the first time. Thus, SCTP mostly avoids exponential backoffs in the tests compared to TCP NewReno in multiple loss scenarios.

Lksctp does not restart the timer after fast retransmits of the earliest outstanding data chunks according to the timer rules. Therefore, the timer and the fast retransmit mechanism could retransmit the same data chunk independent of each other as long as the data chunk is not acknowledged. Figure 3.10 shows a snapshot provided by Sctp_trace which documents independent fast retransmits and retransmissions timeouts of the same data chunk. All in all, this could lead to a behaviour where the data chunk is first retransmitted by a retransmission timeout before it is retransmitted by a fast retransmit independent of the retransmission timer. As the timer is not restarted after a fast retransmit, the timer is triggering a third retransmission of the same data chunk if a SACK is not acknowledging the data chunk. This way of handling retransmissions is erroneous, especially in thick streams. In thin streams, it could lead to faster retransmissions although many retransmissions are spurious. In this scheme, all first retransmissions are triggered as a reason of that the data chunk is lost. However, it is hard to determine which of the following retransmissions are triggered by actual data chunk loss or

```
[264205.476] TSN: 1007
[264248.117] SACK CUM ACK: 1006 #GAP ACKs: 0
[264461.498] TSN: 1008
[264564.135] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1008
[264717.514] TSN: 1009
[264820.146] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1009
[264973.525] TSN: 1010
[265076.161] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1010
[265229.542] TSN: 1011
[265245.495] RETRANSMISSION:
          R1 1007 (1040.019ms) Retr timeout
          R1 1011 (15.953) In flight
[265332.177] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1011
[265485.554] TSN: 1012
[265588.184] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1012
[265741.571] TSN: 1013
[265844.202] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1013
[265997.585] TSN: 1014
[266100.212] SACK CUM ACK: 1006 #GAP ACKs: 1 GAP ACKED: 1008-1014
[266100.284] RETRANSMISSION:
          R2 1007 (1894.808ms) Fast retransmit
[266245.564] RETRANSMISSION:
          R3 1007 (2040.088ms) Retr Timeout
[266253.583] TSN: 1015
[266509.618] TSN: 1016
[266552.232] SACK CUM ACK: 1015 #GAP ACKs: 0
```

Figure 3.10: SCTP erroneous retransmissions

not.

As the messages size in thin streams are small compared to what they are in thick streams there are room for mor data in packets. Bundling of outstanding data chunks can be seen as a good way to faster retransmit data chunks, especially those that are reported missing by some SACK. If data chunks in flight gets bundled, this can be an extra safety to retransmit faster if the data chunk unfortunalety gets lost. Since the number of packets in the network is the same as packets are just filled up, bundling of outstanding data chunks does not make any damage to the network. TCP NewReno does not perform bundling the same way in its retransmissions.

3.5 Ideas for proposed enhancements

There are several ideas for proposed enhancements for SCTP that all shoud be evaluated after the tests and comparision with TCP NewReno. SCTP needs a way to detect that the stream is thin in order to decide if thin streams mechanisms should be used or not. Thus, a mechanism

should be considered and evaluated to define and detect the nature of thin streams.

Since almost all chunks in SCTP are retransmitted by a retransmission timeout in thin streams, the minimum RTO should be reduced since its almost 5 times higher than TCP NewReno's minimum RTO and leads to high retransmission delays. But when the minimum RTO is reduced, it is important to see how thin streams affects the RTO calculations because delayed SACKs may lead to wrong RTT measurements. In addition, late timer restarts delay the expiration time of the retransmission timer. A solution should be considered to avoid a delayed retransmission timer after restarts. TCP New Reno suffers from several exponential backoffs of the retransmission timer, but it could also be a problem in SCTP if no new data chunks are sent within a couple of seconds or no new SACKs arrive and trigger a fast retransmit. Elimination of the exponential backoff mechanism should be considered to avoid high retransmission delays if there are occurrences of multiple loss when almost none SACKs are sent. Since the number of SACKs are small in thin streams and rarely have the chance to trigger a fast retransmit, a way to make use of fewer SACKs to trigger a fast retransmit should be considered. At last, as the data chunks are relatively small in thin streams according to the network MTU and therefore do not make use of all available room inside a packet, new ways to put more data into a packet should be considered if the stream is thin.

3.6 Summary

In this chapter we have seen that the retransmission mechanisms are working well in thick streams. However, when the stream gets thin, both TCP NewReno and SCTP suffer from very high retransmission delays. Several enhancements are needed to trigger retransmissions earlier. In the next chapter, we will evaluate and implement the ideas of the proposed enhancements for SCTP.

Chapter 4

Evaluation and implementation of proposed enhancements

In the previous chapter, we saw that SCTP works well for thick streams except for its lack of handling retransmissions correctly. However, SCTP's original retransmission strategies are not working well in thin streams. In this chapter, we will evaluate and implement several proposed enhancements for thin streams.

4.1 A way to define and detect thin streams

The original retransmission strategies are working well in thick streams and should therefore be used when the stream is thick. However, when the stream is thin, several enhancements are needed which are especially developed for thin streams only. In order to get retransmission strategies to work well in both thin and thick streams, a way is needed to detect and determine the nature of the stream to decide what retransmission strategies should be used. In some cases, the nature of a stream could change between being thin and thick. Something, that should be taken into acount.

4.1.1 Packets in flight to define thin streams

SCTP must be able to define the nature of thin streams to decide if thin stream strategies should be used or not. The number of packets in flight

is a good indication of how thin the stream is.

In thick streams, SCTP sends as much data as possible according to the available bandwidth. The test results show that the retransmission delays in thick streams are very low. This is because there are many packets in flight to the receiver. Once a packet is lost, SCTP uses no time to detect it as the next packets arrive quickly and trigger the SACKs needed to fast retransmit the lost data chunks.

In thin streams, the time between packets could be large and thus reducing the number of packets in flight to the receiver. If a fast retransmit should be triggered in a reasonable time, then a minimum of 5 packets must be in flight to the receiver. If a data chunk is lost, then the next four packets in flight will trigger the 4 SACKs that are needed to fast retransmit the lost data chunks provided that none of the SACKs are lost on the way. This also means that when more than 4 packets are in flight to the receiver, then then the original retransmission strategies should be used as 4 SACKs should arrive in time to trigger a fast retransmit of the data chunk before an expiration of the retransmission timer. Thus, if packets in flight is less than five, then the thin stream retransmission strategies is used.

4.1.2 Packets in flight handling in SCTP

As SCTP is message-oriented, it does not hold packets in the transmission queue. In contrast, TCP uses this strategy because ACKs acknowledge the receipt of each packet; if a packet is lost, the same packet is retransmitted. Instead, SCTP will hold data chunks in the transmission queue as packet contents could vary because of various bundling strategies, for instance when data chunks in a packet are transmitted and later need to get retransmitted. This is possible as a data chunk has its own TSN and SACKs acknowledge the receipt of each data chunk. However, a strategy is needed to keep track of how many packets that are in flight to the receiver in order to determine if thin stream retransmission strategies should be used or not. This is because it is the receipt of each packet, and not each data chunk, that triggers each SACK.

The problem can be solved by keeping the highest TSN in each packet, that are in flight from the sender to the receiver, in a linked list. Each list element will then represent each packet in flight. Each time a packet containing one or more data chunks is sent, the highest TSN in the packet is determined and added to the list. At the same time a variable packets_in_flight is increased by one. If a packet is successfully received

in the absence of loss, then the highest TSN in the packet will correspond to the cumulative TSN of the following SACK, as the cumulative TSN will acknowledge the last data chunk received in sequence and thus the highest TSN. When a SACK arrives, the sender checks if there exist any gap ack blocks. If it does, then the sender knows that one packet has left the network as a SACK is sent back immediately. The missing packets are still counted as in flight to the receiver as they could arrive at a later time. The variable packets_in_flight is decremented by one and a variable packets_left_network is incremented by one. The gap ack blocks indicate that more packets are received, but the sender will not inspect them to find out which packets have left the network as the sender knows that one packet has left the network when a SACK with gap ack blocks arrives.

Instead, when a received SACK does not contain any gap ack blocks, the sender knows that all packets with its highest TSN lesser than or equal to the cumulative TSN of the SACK has left the network. At this point, the sender does not know exactly how many packets that have left the network before the SACK was sent, as the receiver will send a SACK back for every second packet received or within 200 ms of the arrival of any unacknowledged data chunk when no loss is detected. But the sender removes each such packet from the list and count them by decrementing packets_in_flight by one for each removal. The sender must not count packets that already have left the network in the first case. Thus, packets_in_flight is incremented by packets_left_network, and then packets_left_network is reset to zero. The variable packets_in_flight can now be used as an estimate to decide if thin stream retransmission strategies should be used or not.

In SCTP, packets in flight handling should apply to each association. This means that the number of packets in flight are shared between each transport destination address if multi-homing is used. The reason for this is that the transmission queue and the cumulative ack point are shared between each destination transport address. Thus, the association takes care of incoming SACKs independent of which transport destination addreess the data chunks were sent to, manages and cleans up the transmission queue and if necessary send data chunks to another transport destination address.

In Lksctp, there is a struct named `sctp_association` declared in structs.h which holds information about each individual association. The variables `packets_in_flight` and `packets_left_network` is declared in this struct in addition to pointers to the head and tail of the list holding packets in flight. Each list element in the list is represented by the

following struct:

```
struct pkt_in_flight{

  __u32 highestTSN;
  struct pkt_in_flight *next, *prev;
};
```

The routine `sctp_packet_transmit()` located in output.c is responsible for traversing all data chunks in a packet, assign TSNs to data chunks and create specific packet and data chunk headers. Finally it creates the packet as it will appear on the network and sends it to the lower layer. When data chunks in a packet are traversed, the highest TSN in the packet is found by the following code:

```
currentTSN = ntohl(chunk->subh.data_hdr->tsn);

if(currentTSN > pktHighestTSN)
  pktHighestTSN = currentTSN;
```

Finally, the highest TSN representing the packet is added to the packets in flight list by calling `add_pkt_in_flight(asoc,pktHighestTSN)`.

The new routine `add_pkt_in_flight()` is listed in appendix A.1. It is located in output.c and is responsible for adding a new packet to the packets in flight list. It takes as parameters a pointer to the current association and the value of the highest TSN in the packet. Adding a new packet in flight is done by allocation memory for it by calling kmalloc. Finally, it is added to the tail of the list before packets in flight is increased by one.

The routine `sctp_outq_sack()` is responsible for processing an incoming SACK and remove acknowledged data chunks from the transmitted queue. The following test is performed to check if the SACK contains any gap acked blocks and further determine how to count packets in flight.

```
if(sack->num_gap_ack_blocks > 0){
  asoc->packets_in_flight--;
  asoc->packets_left_network++;
}
else{
  remove_pkts_in_flight(asoc, sack_ctsn);
}
```

The new routine `remove_pkts_in_flight()` located in outqueue.c is responsible for removing packets from the packet in flight list that have its highest TSN lower or equal to the cumulative TSN of a SACK. Finally, the value of packets in flight is updated according to the packets that already have left the network. The routnie takes a pointer to the current association and the value of the cumulative TSN in the SACK as paramaters.

4.2 Modifying fast retransmits in thin streams

4.2.1 Fast retransmit after 1 SACK

The test results show that the number of fast retransmits are drastically reduced in thin streams. As explained earlier, this is because not enough packets are sent to get the four SACKs needed to trigger a fast retransmit before the retransmission timer expires. Despite a minimum RTO value as high as 1000 ms, most retransmissions are triggered by a retransmission timeout.

Depending on the time between each sent packet, less than four SACKs could arrive before the retransmission timer expires. Since there are very few SACKs reporting loss in thin streams, a fast retransmit should be triggered by the first indication that a data chunk is lost. This means that if the stream is thin, then a fast retransmit should be triggered by one SACK. This is because there could be few SACKs reporting loss in thin streams and the time waiting for 4 SACKs could lead to high retransmission delays. As fast retransmits has been shown to be working properly in thick streams, SCTP should use the original fast retransmit mechanisms by waiting for 4 SACKs if the stream is thick.

The reason for waiting for 4 SACKs is to be sure that data chunks are actually lost and not reordered in the network. If the stream is thick, the sender uses short time to discover that SACKs were sent because of a reordering of packets in the network. If a SACK reports the data chunk to be lost during a possible reordering, then the next SACK acknowledging the data chunk arrives in short time. In thin streams, waiting for several SACKs to discover possible reorderings of packets in the network takes long time. In thin streams, the time between each packet could be so large that packet reordering should happen much less than in thick streams. xsIf a reordering unfortunately occurs, then the receiver will earn more by receiving a spurious retransmission than let the sender

wait for several SACKs to discover possible reorderings. As very few data chunks are sent, some spurious retransmissions will not eat much of the available bandwidth.

In Lksctp, the `routine sctp_mark_missing()` located in outqueue.c is responsible for traversing the transmitted queue and mark data chunks as missing when a SACK arrives and furhter start the fast retransmit procedure if necessary. The old algorithm for marking a data chunk as eligible for fast retransmit is replaced by the following algorithm:

```
if(q->asoc->packets_in_flight < 5)
  fr_threshold = 1;
else
  fr_threshold = 4;

if (chunk->tsn_missing_report >= fr_threshold) {
  chunk->fast_retransmit = 1;
  do_fast_retransmit = 1;
}
```

In the algorithm, `q` is a pointer to the current transmitted queue. As each transmitted queue belongs to a specific association, the number of packets in flight can be found by further following the `asoc` pointer to the current association. If the stream is thin, a variable `fr_threshold` of type int is set to 1 to tell SCTP to fast retransmit a data chunk after 1 SACK reporting it to be lost. Otherwise, if the stream is thick, `fr_threshold` is set to 4 to fast retransmit the original way. Finally, a test is performed to check if a data chunk's missing reports are greater than or equal to `fr_threshold`. If this is true, the data chunk is marked as eligible for fast retransmit by setting `chunk->fast_retransmit` to true. Finally, `do_fast_retransmit` is set to true to tell SCTP to start the fast retransmit procedure when the transmitted queue has been traversed.

4.2.2 Bundling of outstanding data chunks in fast retransmits

If there is room in the packet after a retransmission timeout, SCTP performs bundling of the earliest outstanding data chunks. In thick streams, this ensures that as much data as possible is transmitted to the reciever as a retransmission timeout is an indication of heavy congestion. In thin streams, a retransmission timeout is a indication of a very thin stream as the data chunks are not acknowledged before the retransmission timer

expires. This way, the earliest outstanding data chunks could get retransmitted faster and ensur that as many data chunks as possible are sent within a single packet. If the bundled outstanding data chunks are already received, then the receiver will just drop them and it will not lead to more packets on the network or increasing router costs. However, when SCTP fast retransmits in thin streams, then bundling of outstanding data chunks is not performed unless they are marked for fast retransmit. As packets must be filled up as often as possible, SCTP can also perform bundling of outstanding data chunks in fast retransmits, the same way as after a retransmission timeout. If the stream is thick, SCTP should use the original bundling strategies as these work properly in thick streams.

In Lksctp, the routine `sctp_retransmit_mark()` located in outqueue.c is responsible for traversing the transmitted queue and add data chunks that are marked for retransmission to the retransmit queue. The routine is called when SCTP is going to retransmit. If the routine is called as a reason of a retransmission timeout, then all outstanding data chunks are added to the retransmit queue as the earliest outstanding data chunk will get retransmitted and even more could get bundled if allowed by the network MTU. If the routine is called because of a fast retransmit, then only the data chunks that are marked for fast retransmit is added to the retransmit queue.

For each data chunk in the transmitted queue, the test is originally performed by the following algorithm expressing the requirements above:

```
if((fast_retransmit && chunk->fast_retransmit) ||
   (!fast_retransmit && !chunk->tsn_gap_acked)){
              .
              .
}
```

This test is replaced by the following algorithm which allows bundling of outstanding chunks in fast retransmits:

```
if(check_thinstream_before_add(transport, chunk,
                               fast_retransmit)){
              .
              .
}
```

The routine `check_stream_before_add()` is listed in appendix A.3. It is located in outqueue.c and takes a pointer to the struct representing

the current transport address, a pointer to the struct representing the current chunk and a variable `fast_retransmit` which tells SCTP if it is a retransmission timeout or a fast retransmit as paramaters. First, a test is performed to check if the stream is thin by following the pointer to the current transport address' association to get the number of packets in flight. If they are less than 5, all outstanding data chunks are added to the retransmit queue in addition to data chunks marked for fast retransmit independent of if it is a retransmission timeout or a fast retransmit. To find out if the data chunk is outstanding or not, a test is performed on the value of `chunk->tsn_gap_acked`. If the stream is thick, outstanding data chunks are only bundled if it is a retransmission timeout, as the original way.

4.3 Modifying the SCTP retransmission timer

4.3.1 Reducing the RTO minimum value

If the stream is thin, the test results show that most retransmissions occurs due to an expiration of the retransmission timer. If the time between each packet is large and one is lost, it could take long time before a SACK arrives and triggers a fast retransmit as the receiver needs new packets to discover loss before a SACK is sent. At this point, a retransmission is determined by the RTO value if SACKs do not arrive before the retransmission timer expires. With a minimum RTO value of 1000 ms, many retransmissions should be triggered by a fast retransmit unless the stream is very thin. However, the time between a data chunk is sent and gets fast retransmitted is determined by the time between data chunks are sent and could lead to large retransmission delays, especially if SACKs are lost. In such cases, the retransmission timer must get triggered earlier and the minimum RTO value must be reduced accordingly.

To trigger a retransmission timeout in a reasonable time, the RTO minimum value is reduced to 200 ms when the stream is thin. This way, data chunks will get retransmitted earlier than the old RTO minimum value of 1000 ms if a retransmission is not triggered by an intervening fast retransmit. The RTO calculations used in SCTP could get lead to an RTO that is close to the RTT [18]. Here, we are not considering to implement a retransmission timer that fixes this problem due to the available time we have during this thesis. Instead, we simply set the RTO minimum value to 200 ms. If the RTO calculation results in a value that is below

200 ms, then it is set to 200 ms. If the stream is thick, then the original RTO minimum of 1000 ms is used. That way, SCTP uses its original RTO calculation when the stream is thick.

In SCTP, each transport address belonging to an association has its own retransmission timer if multi-homing is used. When multi-homing is not used, SCTP uses a single retransmission timer for the primary transport address. . In Lksctp, the routine `sctp_transport_update_rto()` located in transport.c is responsible for calculating a new RTO and is called once an RTT measurement has been made. Each transport address has its own struct `sctp_transport` that is holding information about the transport and variables required for RTO calculation. When the time comes for the new RTO to be set, the old RTO update algorithm will be replaced by the following algorithm:

```
if(tp->asoc->packets_in_flight < 5){
  if(tp->rto < msecs_to_jiffies(200))
    tp->rto = msecs_to_jiffies(200);
}
else{
  if(tp->rto < tp->asoc->rto_min)
    tp->rto = tp->asoc->rto_min;
}
```

First a test is performed to check if packets in flight are less than 5. Each transport has a pointer to the association which it belongs to. A pointer `tp` points to the current transport and `packets_in_flight` can be found by following the `asoc` pointer to the current association. The RTO is set in jiffies, which are the number of ticks that have elapsed since the system is booted and is the time unit used by timers in the Linux kernel. The routine `msecs_to_jiffies()` is used to convert between ms and jiffies before the RTO is set. If packets in flight are greater than or equal to 5, then the RTO is set to the original RTO minimum value if it is less than RTO minimum.

4.3.2 Thin stream influence on RTO calculations

To trigger retransmissions earlier in the absence of SACKs, a reduction of the RTO minimum value has been done. When the RTO minimum value is reduced to 200 ms, it is necessary to evaluate thin streams influence on the RTO calculations. The reason for this is that the RTO value is based on measured RTT and is updated each time an RTT measurement has been made. If packets are sent in large intervals, the interval

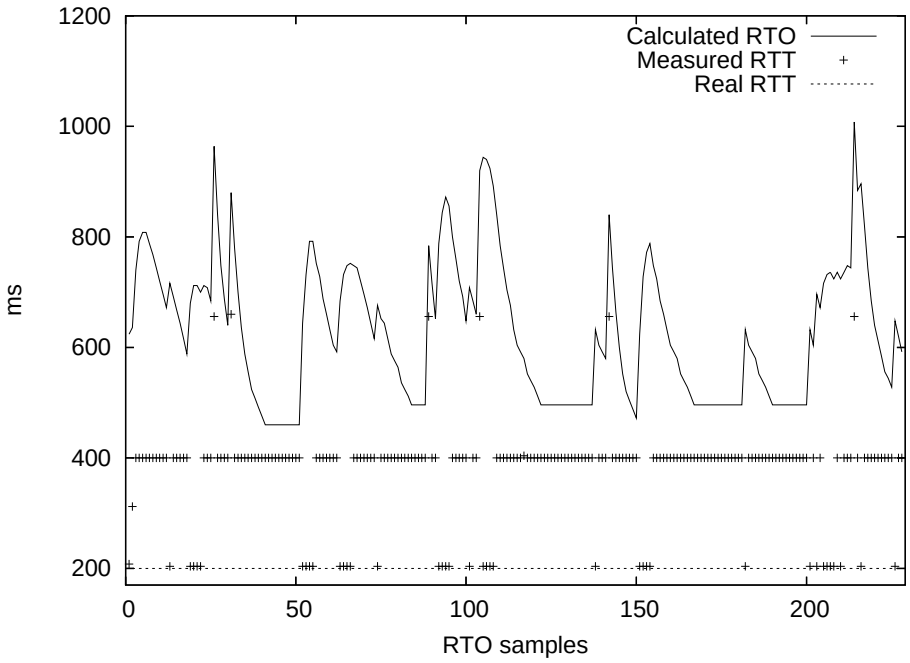

Figure 4.1: Calculated RTO in thin streams

time will determine the measured RTT as the receiver sends a SACK back After receiving the second packet or within the SACK delay of 200 ms if no new packets arrive. As a consequence, the measured RTT could be from from 0 to 200 ms too high, depending on the time between each arrived packet. This should result in an over estimation of the RTO as a result of incorrect RTT measurements. A correct RTT measurement is only made when the receiver discovers loss and immediately sends a SACK.

To see the thin stream influence on RTO calculations, a test is run where data chunks are sent in an interval of 250 ms with 5% packet loss and a constant RTT of 200 ms. Several samples of the calculated RTO and the corresponding RTT are collected during the test. Figure 4.1 shows the collected samples of the calculated RTO and the measured RTT. The dotted line shows the real RTT between sender and receiver which in this test is constant as it is emulated by Netem.

In this thin scenario, the measured RTT is mostly 400 ms as delayed SACKs lead to a 200 ms increase of the measured RTT compared to the real RTT. A few RTT measurements down to 200 ms occur when SACKs are sent back immediately. In addition, some few RTT measurements at

approxemately 650 ms occur when SACKs are lost and the RTT measurement is made based on the next SACK which is triggered by the next packet arriving 250 ms later.

If the measured RTT is suddenly dropping or raising, then the calculated RTO will increase at approxemately the same rate [18]. The reason for this is SCTP's computation of the RTTVAR parameter used in RTO calculations. In the computation of RTTVAR, the value $\alpha * |SRTT - R'|$[1] is added to $(1 - \alpha) * RTTVAR$. If R' is drastically reduced or increased, then the computation of $|SRTT - R'|$ is leading to large values as SRTT is the smoothed value of the last measured RTT. This results in adding a quarter of the difference in measured RTT to the new RTTVAR. The changes in the measured RTT will not affect SRTT as much as RTTVAR beacuse β results in less weightning of R' when $\beta * R'$ is used to update SRTT. This is because β is set to half the value of α. When the new RTO is computed, then $4 * RTTVAR$ will approxemately add the difference in measured RTT to the new RTO as $4 * RTTVAR$ upweights $\frac{1}{4} * |SRTT - R'|$.

The figure shows that rapid changes in RTT measurements will lead to almost an equivalent rapid growth of the RTO. This results in several peaks up to almost 1000 ms in addition to an already over estimated RTO. If data chunks are sent in varying intervals, the variance in RTT measurements leads to the same effect on the calculated RTO. A modification must be considered to get more correct RTO estimates despite that packets could be sent in large and varying intervals.

The best solution to the problem is to avoid delayed SACKs if the stream is thin despite this will change the strategies of the receiver. Another solution is to change the α and β values used in the weightening of RTTVAR and SRTT. This will make the RTO not so affected by rapid changes in the RTT. Changing the variables used in RTO calculations require a complete evaluation of the following RTO behaviour, especially in thin streams. There is a reason for the choice of RTO calculation algorithm that is strongly evaluated under most conditions. Without a omplete evaluation, changing the RTO calculation algorithm may lead to critical errors if it is not proved to work properly. Hence, we will not touch the RTO calculations, but consider delayed SACKs removal to be the best experimental solution.

Figure 4.2 shows the RTO calculations of a rerun of the same test where SACK delay is removed. This means that the receiver will send a SACK immediately without waiting up to 200 ms in the absence of the next

[1] $\alpha = \frac{1}{4}$, R' = the new RTT measurement

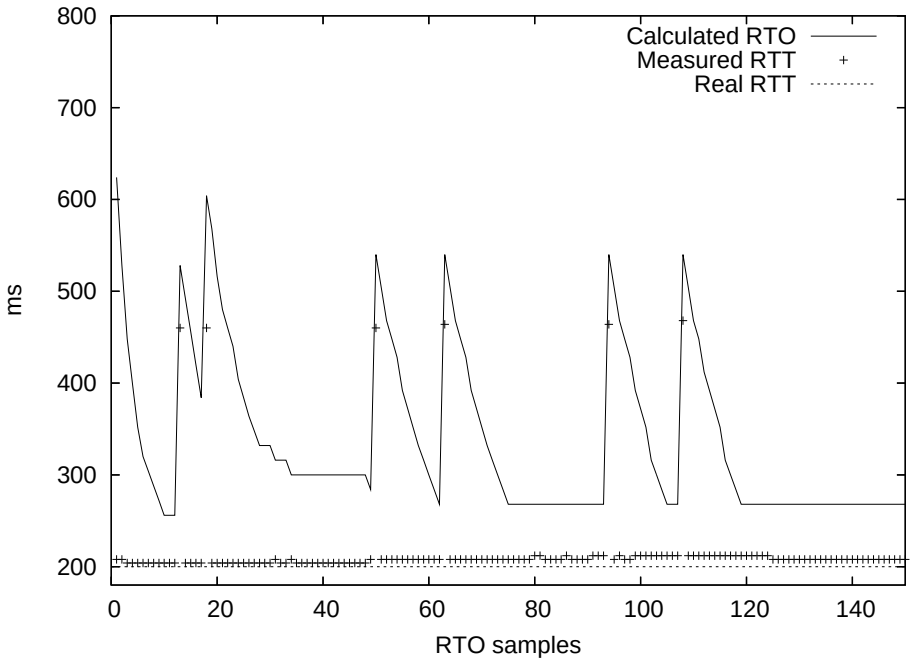

Figure 4.2: Calculated RTO without SACK delay

packet. Figure 4.2 shows that removal of SACK delays leads to correct RTT measurements despite that packets are sent in large intervals. The only exception is when SACKs are lost and the RTT measurement is done based on the next SACK. This leads to some peaks in the calculated RTO up to 600 ms.

Subsequent correct RTT measurements leads to a calculated RTO that falls down to a constant value around 250 ms, 50 ms above the actual RTT. Totally, the overall RTO values are lower when SACK delay is removed, compared to the RTO values when SACK delay is used. As a result, SACK delay should be considered removed unless the timer is taking delayed SACKs into acount in its RTO calculations. Since we have not implemented and evaluated such a timer, removal of delayed SACKs could give a good view of the retransmission delays if such a timer existed.

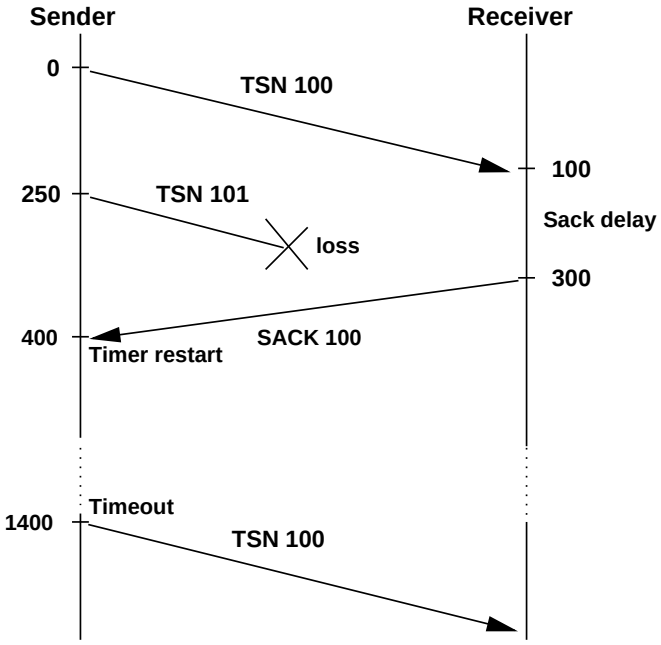

Figure 4.3: Timer restart in thin streams

4.3.3 Correcting the RTO value in timer restarts

SCTP performs a restart of the retransmission timer with the current RTO at the time an incoming SACK acknowledges some, but not all outstanding data chunks. In thin streams, late arrivals of such SACKs often occur as the time between packets could be large in thin streams. This leads to a delayed expiration time of the timer. An example of a timer restart in thin streams is shown in figure 4.3. In this example, data chunks are sent every 250 ms with an RTT of 200 ms. At time 0 ms, the data chunk with TSN 100 is sent and successfully received at time 100 ms. As no new data chunks arrives within the SACK delay of 200 ms, the SACK is sent back at time 300 ms and arrives at the receiver at time 400 and acknowledges the data chunk with TSN 100. As the data chunk with TSN 101 is sent at time 250, it is outstanding when the SACK arrives. Thus, at time 400 ms, SCTP performs a restart of the timer with the current RTO of 1000 ms. The data chunk with TSN 101 is lost and is then retransmitted by a retransmission timeout at time 1400 ms, 1150 ms after it was sent. In the example, this leads to a 150 ms delay in the expiration time of the timer.

Depending on the nature of the thin stream, the retransmission timer

could be a victim of large delays when SACKs arrive late. The testresults show that retransmission timer restarts create large retransmission delays in thin streams which should be reduced accordingly. The delayed expiration of the retransmission timer is avoided by adjusting the timer expiration time before it is restarted. No more than RTO ms should have elapsed before the current outstanding data chunks will get retransmitted by a timeout although SACKs arrive later and restart the timer.

When the retransmission timer expires, the earliest outstanding data chunk is always retransmitted first. In the same way; since the earliest outstanding data chunk is the oldest data chunk, it suffers most when the timer is restarted if it later needs to get retransmitted by a retransmission timeout. According to [18], the solution to the problem is to always subtract the age of the earliest outstanding data chunk from the current RTO value each time the timer is restarted. If this data chunk is lost, then no more than RTO ms will elapse before it is retransmitted by a timeout.

In Lksctp, the routine `sctp_check_transmitted()` in outqueue.c is responsible for cleaning up the transmitted queue when a SACK is received and further stop or restart the retransmission timer. The transmitted queue is holding data chunks that are sent, but not yet acknowledged and is structured as a list of sctp_chunk structs where each struct is holding relevant information about each chunk. Once data chunks are acknowledged by the cumulative TSN of a SACK, they will get removed from this queue. This means that data chunks that are gap acked by a SACK is still on the queue in combination with outstanding data chunks. Data chunks are added to the tail of the queue once they are transmitted which means that the earliest outstanding data chunk is the first data chunk on the queue that is not gap acked yet.

A modified restart timer algorithm takes care of handling the requirements described above. It's code is listed in appendix A.4. If the retransmission timer needs to get restarted, a variable `restart_timer` is set to true. A test on this variable is performed to check if a restart of the timer should be made. At this point, the current management of retransmission timer restarts is replaced by the modified restart timer mechanism. First, the earliest outstanding data chunk needs to be found. This is done by traversing the transmitted queue and look for the first data chunk that has not been acknowledged yet. If this data chunk is found, the time in jiffies of when it was last sent is extracted from the `sent_at` variable in the data chunk's struct. When a data chunk is sent, the current time is always set in this variable and is originally used for

RTT measurement if necessary. The age of a data chunk can now be found by subtracting this value from the current jiffies value. When the timer is restarted, the age of the data chunk is subtracted from the current RTO before this RTO is used in the restart of the timer. If the earlist outstanding data chunk is not found, the timer is restarted the original way. To avoid late timer restarts both in thick and thin streams, the modified restart timer algorithm should be considered implemented as the usual way of restarting timers in SCTP.

4.3.4 Exponential backoff elimination

If there are very few SACKs to trigger a fast retransmit or no new packets are sent to let the receiver discover loss, retransmissions could be triggered by subsequent retransmission timeouts without any intervening fast retransmits. At this point, an exponential backoff of the retransmission timer is performed, leading to an exponential growth of the retransmission delays when it should be unecessary. If no new packets are sent or few SACKs arrive as the stream is so thin, this is not necessary an indication of heavy congestion which is the reason for doubling the retransmission timer. By eliminating exponential backoffs when the stream is thin, subsequent retransmissions timeouts will not lead to an exponential growth of the following retransmission delays.

In Lksctp, exponential backoff of the retransmission timer is handled in the routine `sctp_do_8_2_transport_strike()`, located in sm_sideeffect.c. The routine is responsible for handling path failure detection by marking the destination transport address as inactive if the maximum number of retransmissions after subsequent retransmission timeouts is exceeded. In such cases, each timer expiration leads to an exponential backoff of the retransmission timer. The `packets_in_flight` variable is accessible by following the `asoc` pointer to the current association. If packets_in_flight is less than 5, then exponential backoff is eliminated by the following code:

```
if(asoc->packets_in_flight >= 5){
   transport->rto = min((transport->rto * 2),
                        transport->asoc->rto_max);
}
```

4.4 Bundling of outstanding data chunks with new data chunks

In thin streams, packets could be sent in large intervals which means that the retransmission delay will increase as a result of waiting for the timer to expire or late arrivals of SACKs that trigger fast retransmits. In the time between a data chunk is sent and gets retransmitted, new packets could be sent since the retransmission delays in thin streams could be high.

If the size of a new packet is small according to the network MTU, then it is suitable for bundling of outstanding data chunks if there is room for more data chunks in the packet. Bundling is possible in SCTP although there are gaps between each data chunk. The reason for this is that each data chunk has its own sequence number and appears independent of each other inside the packet. The bundling is performed by starting with the earliest outstanding data chunks until the packet is filled up or there are no more data chunks eligible for bundling. The earliest outstanding data chunks are bundled first as they are the data chunks that need to be delivered first. Bundling of outstanding data chunks is only done when the stream is thin. If the stream is thick, bundling is not performed.

The effect of this procedure is that bundling of outstanding data chunks is a way to faster deliver data chunks to the receiver since the next packet is sent before a possible retransmission of the outstanding data chunks occurs. This procedure does not lead to an increase in router costs or a decrease of the available bandwidth in the network as the same number of packets are sent with or without bundling. It does not affect the delivering of new data chunks as bundling of outstanding data chunks is only performed if there is room in the packet after the new data chunks are placed in the packet. In addition, bundling of outstanding data chunks is not considered to be a normal retransmission. Thus, it does not affect the congestion control by reducing the congestion window per retransmission. As the size of the congestion window is determined by the size of the data chunks, bundled data chunks affect the congestion window the same way as new data chunks.

Situations where the receiver benefits from this procedure are when data chunks are lost and possibly are bundled in the next packet that arrives. If all data chunks that unfortunately get lost are bundled in the next packet, then the receiver has no chance to discover this as the earliest outstanding data chunks are the next data chunks which the reciever expects to arrive. In addition, lost data chunks could arrive even before

1 SACK or a retransmission timeout triggers a retransmission of them. However, the disadvantage of this procedure is that the loss rate in the network could be small which leads to data chunks that are delivered up to several times. The receiver solves this problem by dropping the data chunks that are already delivered. Since the stream is thin, there are not many data chunks that need to get dropped compared to if this type of bundling was performed in thick streams.

In Lksctp, the routine `sctp_outq_flush()` located in outqueue.c is responsible for traversing the transmission queue and the retransmission queue, and send or retransmit data chunks by sending packets to the lower layer. The packets could be sent to different transport addresses if multi-homing is used. First, the retransmit queue is traversed to retransmit data chunks before sending new data chunks. Afterwards, the transmission queue is traversed to send new data chunks. Next, bundling of control chunks with data chunks is done if there are any control chunks ready to be sent. Further, if a packet is filled up, then it is sent to the lower layer with either new data chunks or retransmitted data chunks. Finally, all unsent packets are sent to the lower layer. These are packets that are not filled up yet or for another reason not sent. If these packets are small according to the network MTU, they are eligible for bundling of outstanding data chunks. The following test is added to check if the stream is thin before trying to bundle outstanding data chunks in the packet before it is sent:

```
if (!sctp_packet_empty(packet)){
  if(asoc->packets_in_flight < 5){
     bundle_outstanding_chunks(packet, t);
  }
  error = sctp_packet_transmit(packet);
}
```

The new routine `bundle_outstanding_chunks()` is responsible for bundling of outstanding chunks with new chunks and is listed in appendix A.5. It takes a pointer to the packet and a pointer to the current transport destination address as parameters. First, a test is performed to check if it is a valid transport destination address. If it is not, then bundling is aborted. Next, chunks that are already stored in the packet, represented by `packet->chunk_list`, is traversed to find the first data chunk in the packet. If this data chunk already has a TSN, this packet contains data chunks that will get retransmitted. Bundling of outstanding data chunks prior to the retransmission has already been done and the current bundling is aborted. Otherwise, the packet contains new data

chunks as TSN assignments are done later by `sctp_transmit_packet()`, just before the packet is sent.

Before bundling can start, the size of the packet and the network MTU is found. Next, a list named `outstanding_list` is initialized to hold all outstanding data chunks that will get bundled into the packet. The reason for this is that all data chunks in a packet must be in TSN increasing order. As the outstanding data chunks have a lower TSN than the new data chunks will get, the outstanding data chunks must be placed in the start of the packet and are first added to the list.

Then, the transmitted queue is traversed to find outstanding data chunks. If an outstanding data chunk is found, the size of the data chunk including headers and padding is calculated. If there is no more room in the packet according to the MTU, then bundling is aborted. Otherwise, the data chunk length is added to the packet size and the size of the current bundled outstanding data chunks, before the data chunk is added to the tail of the outstanding list. Finally, the list containing the outstanding data chunks is merged with the packet's chunk list such that all outstanding chunks appear in the start of the packet. The final size of the packet is calculated and stored in the packet struct.

4.5 Summary

In this chapter, we have evaluated and implemented several proposed enhancements to improve the retransmission delays in thin streams. Since the original retransmission strategies are working well in thick streams, SCTP needed a way to detect a thin stream as the proposed enhancements is used in thin streams only. We have considered the number of packets in flight to indicate that the stream is thin and determine the retransmission strategies. A way of determine the number of packets in flight is evaluated and implemented as SCTP does not have this functionality built-in. Further, we have considered fast retransmit after 1 SACK as necessary in thin streams as there could be few SACKs. In addition, bundling of outstanding data chunks is also performed with data chunks that are fast retransmitted.

We have reduced the RTO minimum value to 200 to let the retransmission timer expire earlier, in the absence of SACKs triggering fast retransmits. When the RTO value is decreased, we saw that the RTO value could get over-estimated as an effect of incorrect RTT measurements caused by delayed SACKs or large time intervals between each packet. Thus, re-

moving delayed SACKs have been considered as a good solution since we have not evaluated and implemented a timer that handles the problem. Late restarts could lead to delayed expiration time of the retransmisison timer. This is solved by letting no more than RTO ms elapse before the timer expires independent of that the restart is delayed or not.

We eliminated exponential backoff as this could lead to high retransmission delays if data chunks are continously retransmitted by subsequent retransmission timeouts. Finally, we saw that bundling of outstanding data chunks with new data chunks is a good solution to faster retransmit data chunks. As the same number of packets is sent with or without bundling, this does not lead to higher routing costs or bandwidth requirements. In addition, the retransmission of bundled outstanding data chunks does not affect the congestion window as a fast retransmit or retransmission timeout does. Bundling is possible in SCTP although there are gaps between each data chunk. The reason for this is that each data chunk has its own sequence number and appears independent of each other inside the packet. In the next chapter we will test and evaluate the proposed enhancements' effect on the retransmission delay.

Chapter 5

Testing of proposed enhancements

We have evaluated and implemented several proposed enhancements for thin streams. In this chapter, we will test and evaluate their effect on the retransmission delays in thin streams. The test results are compared against the test results of the original SCTP protocol which is discussed and evaluated in chapter 3.

5.1 Test Layout

5.1.1 Testing various enhancements

Each proposed enhancement of the SCTP protocol has been developed to improve the retransmission strategies according to the various characteristics of a thin stream. First, we will show that the modified restart timer mechanism is working properly by running a simple test where we look at a single snapshot of a retransmission timeout where the timer is restarted by a late SACK.

Next, in order to see the full effect of the new mechanisms, they are introduced one by one and evaluated against the test results where the original SCTP protocol has been used. We will first test and evaluate the modified fast retransmit mechanism in combination with the RTO minimum reduction. In the second test, we are including the modified restart timer algorithm. In the third test, we are removing delayed SACKs

in addition to keep using the proposed enhancements of the previous test. As described in section 4.3.2, delayed SACKs have been shown to have a negative effect on the RTO calculations as they lead to incorrect RTT measurements when the time interval between each packet is large or changes rapidly. Although there is impossible to remove delayed SACKs at all receivers, it is interesting to see the effect on the retransmission delays when delayed SACKs are removed. This will give an indication on how the retransmission delays performs if the RTO calculation is taking delayed SACKs into account. In total, this results in three tests which let us evaluate each mechanism's influence on the retransmission delays separately.

All tests are running the original test configuration as described in section 3.1. This means that a data chunk of 100 byte + size of headers is sent every 250 ms with a 5% packet loss. This let us compare and evaluate the proposed enhancements against the original protocol. SCTP has shown to have the same behaviour in thin streams independent of the RTT except that the retransmission delays are increasing as a result of an increase in the RTT. Hence, all tests are run in the same loss scenario with an RTT of 200 ms only.

5.1.2 Testing modified fast retransmit

To see the full effect of the modified fast retransmit mechanisms, three tests are run by sending a burst of 4 data chunks, of 100 byte each, every 1000 ms. To avoid the data chunks to get bundled into one packet, there is a 10 ms time interval between each data chunk. As the modified fast retransmit is triggered by 1 SACK, it is possible to trigger enough fast retransmits to see its full effect in thin streams when a burst of 4 data chunks is sent. To make the tests comparable, a test is run with the original protocol to see the performance of the retransmission strategies in this scenario. The second test is run with the modified fast retransmit mechanism and the reduced RTO, but without bindling in fast retransmits. The final test includes bundling in fast retransmits to see if there is possible to bundle more chunks. All tests are run with an RTT of 200 ms and 5% packet loss.

5.2 Omitted tests

We are not testing and evaluating the effect of exponential backoff. In the original tests, there were few occurrences of exponential backoff because SCTP allows sending new data chunks after a retransmission timeout which could trigger new fast retransmits or collapse the exponential RTO value down to its normal value. Thus, to be able to see the effect of some consequtive exponential backoffs, the time between each data chunk must be several seconds. If data chunk loss occurs, this will only trigger the retransmission timer. During multiple loss, the effect of several exponential backoffs can be seen since no new data chunks are sent to trigger fast retransmits or collapse the timer within several seconds. However, this sort of testing tries to enforce exponential backoffs to get the desired effect on retransmission delays as it does not appear clearly in the original tests. Thus, it is dropped.

Bundling of outstanding data chunks with new data chunks has not been tested. The reason is that the effect of bundling does not appear clearly by just looking at the retransmission delays performed by Sctp_trace. If the same data chunk is bundled in consequtive packets, then Sctp_trace will measure the bundling as a retransmission which could lead to high cumulative retransmission delays of the same data chunks. Thus, the bundling gives a strange view on the retransmission delays where the bundling's real effect do not appear clearly. To be able to test bundling and see its full effect, involving the receiver should be considered. At the receiver, it is possible to measure the effect of receiving data chunks as fast as possible. The available time we have during this thesis is too short to be able to test this out.

5.3 Test results

The improvement of the new mechanisms is best seen in the first retransmissions of data chunks and is therefore most important for the evaluation. The reason for this is SCTP's erroneous handling of retransmissions by letting SACKs and the retransmission timer trigger retransmissions independently of each other. Thus, it is almost impossible to know if consequtive retransmissions of the same data chunk are spurious or not. However, the first retransmissions of data chunks are always triggered as a reason of that the data chunk is actually lost. In this way, possible improvements of the mechanisms should appear clearly in the first retransmissions of data chunks. The new modifications should

Test Scenario	Type	Number	Min	Max	Avg
orig. SCTP	Retransmission timeout	266	996.2	1460.1	1144.6
	Fast retransmit	35	1228.4	1740.7	1274.2
	B:Reported missing	24	487.9	976.0	780.7
	B:In flight	338	28.0	888.0	172.8
	Retr timeout & FR	301	996.2	1740.7	1159.6
+reduce RTO+FR	Retransmission timeout	197	435.8	732.0	626.7
	Fast retransmit	284	460.3	731.2	472.3
	B: Reported missing	0	0	0	0
	B: In flight	635	3.8	471.8	197.6
	Retr timeout & FR	481	435.8	732.0	535.5
+mod timer restart	Retransmission timeout	331	436.0	696.0	525.0
	Fast retransmit	288	460.2	464.3	462.3
	B: Reported missing	0	0	0	0
	B: In flight	574	12.0	616.0	189.6
	Retr timeout & FR	619	436.0	696.0	495.8
+no SACK delay	Retransmission timeout	633	255.7	448.0	282.9
	Fast retransmit	1	463.0	463.0	463.0
	B: Reported missing	0	0	0	0
	B: In flight	425	8.0	255.9	47.1
	Retr timeout & FR	634	255.7	463.0	283.2

Table 5.1: SCTP 1. retransmission statistics, RTT = 200 ms

affect the cumulative retransmission delays of each number of retransmissions. Thus, we will take a look at how the proposed enhancements are affecting the retransmission delays when retransmissions caused by retransmission timeouts and fast retransmits are seen as a whole.

5.3.1 Various enhancements

The test results are presented in table 5.1 and will be discussed and evaluated in the following sections.

Restart timer modification

To be able to show that the modified restart timer algorithm is working properly, one simple test is run with the original protocol including the modified restart timer algorithm. Figure 5.1 shows a snapshot from the

```
pkt 1 [43007.009] TSN: 167
pkt 2 [43096.221] SACK CUM ACK: 165 #GAP ACKs: 0
pkt 3 [43263.033] TSN: 168
pkt 4 [43352.230] SACK CUM ACK: 166 #GAP ACKs: 0
pkt 5 [43519.038] TSN: 169
pkt 6 [43668.262] SACK CUM ACK: 166 #GAP ACKs: 1 GAP ACKED: 168-168
pkt 7 [43775.059] TSN: 170
pkt 8 [43924.259] SACK CUM ACK: 166 #GAP ACKs: 1 GAP ACKED: 168-169
pkt 9 [44007.036] RETRANSMISSION:
                  R1 TSN: 167 (1000.027ms) Retr Timeout
                  R1 TSN: 170 (231.977ms) In flight
```

Figure 5.1: Snapshot of a timer restart

packet-by-packet listing of the test which is provided by Sctp_trace. The snapshow shows a retransmission timeout of data chunk 167 and the following timer restart. First, packet 1 containing data chunk 167 is sent at time 43007.0 ms. Then, a SACK arrives 345.2 ms later and restarts the timer with the current RTO as the SACK acknowledges data chunk 166 when data chunk 167 is still outstanding. Finally, data chunk 167 is retransmitted by a retransmission timeout at time 44007.036 ms, RTO ms after it was sent. The modified restart timer algorithm ensures that no more than RTO ms elapse before the timer expires although it is restarted by late arrivals of SACKs.

RTO min reduction and modified fast retransmit

In this test scenario, the test is run with the modifed fast retransmit mechanism and the RTO min reduction only. With this modifications, we expect to trigger retransmissions earlier, both by retransmission timeouts and fast retransmits.

By looking at the first retransmissions of the test, denoted **+reduce RTO+FR** in table 5.1 caused by retransmission timeouts, we see an average retransmision delay of 626.7 ms. Although the RTO minimum value is reduced to 200 ms, the reason for the high average retransmission delay is the delayed SACKs' influence on the RTO calculations as shown in section 4.3.2. With a reduced RTO minimum value only, the tests show that the minimum retransmission delay of a retransmission timeout is 435.8 ms. As the RTO minimum value in the modified protocol is 200 ms, this shows the lowest value possible when the RTO calculations are influenced by delayed SACKs. In comparision, the original protocol, de-

noted **orig. SCTP** in the table, has an average retransmission timeout delay of 1144.6 ms. By reducing the RTO minimum value only, this gives an improvement of 517.9 ms.

The modified fast retransmit mechanisms are meant to make use of fewer SACKs reporting loss by triggering a fast retransmit after the first SACK. The retransmissions caused by fast retransmit show a minimum retransmission delay of 460.3 ms. Assume that a lost packet is sent at time 0. The next packet is then sent 250 ms later and is successfully received. With an RTT of 200 ms, the second packet is received 350 ms after the lost packet is sent. The receiver discovers the loss and sends a SACK immediately which arrives at the sender 450 ms after the lost packet is sent and triggers the fast retransmit. With protocol overhead and other possible network delays,this gives a minimum retransmission delay of 460.3 ms. With an average retransmission delay of 472.3 ms, the modified fast retransmit always make use of the first SACK that arrives and reduce the retransmission delays to almost a third compared to the corresponding retransmission delay of the original protocol where it is 1274.2 ms. The maximum retransmission delay is 727.5 ms and occurs if a SACK is lost and the modified fast retransmit is triggered by the next SACK. At this point, the RTO value is so high that the next SACK triggers a fast retransmit before the timer expires. We see that the number of fast retransmits has increased compared to the original protocol. In cases where retransmission timeouts are displaced by delayed SACKs, the modified fast retransmit will take care of retransmitting the data chunks before the timer expires and trigger retransmissions earlier.

As a fast retransmit is triggered by 1 SACK, there are no bundling of outstanding data chunks that are reported missing as they will get retransmitted by a fast retransmit. By looking at the retransmission delays of bundled data chunks in flight, no significant difference in the retransmission delays can be seen. In the test scenario of the original protocol, 338 data data chunks in flight are bundled in their first retransmission. In comparision, 635 data data chunks in flight are bundled in this test scenario. A decrase in the RTO minimum value increases this number as retransmissions are triggered earlier in the following retransmissions. This makes it possible to bundle more outstanding data chunks.

By looking at the first retransmissions of both retransmission timeouts and fast retransmits as a whole, denoted **Retr timeout & FR**, in each test scenario, it gives an average retransmission delay of 535.5 ms. In the original protocol, the corresponding value is 1159.6 ms. This gives an improvement of 624.1 ms which is a considerable reduction.

Including the restart timer modification

In this test scenario, we are including the new restart timer modification in addition to continue using the mechanisms of the previous test. The goal is to show the modified timer restart's influence on the retransmissions caused by retransmission timeouts. We are now making use of all proposed mechanisms except for bundling of outstanding data chunks with new data chunks. The results of the test, denoted **+mod timer restart** is shown in table 5.1.

By looking at the retransmissions caused by retransmission timeouts, the average retransmission delay is 525.0 ms. In comparision, the corresponding average retransmission delay of the previous test scenario is 626.7 ms. Thus, the modified restart timer algorithm alone gives an improvement of 101.7 ms on the retransmission timeouts.

By looking at the first retransmissions caused by fast retransmits, we see a minimum retransmission delay of 462.3 ms. This is caused by 288 fast retransmits and show that fast retransmits could still get triggered in a first retransmission although the modified restart timer algorithm is in use. In this loss scenario, the time needed to trigger a fast retransmit is 450 ms as in the previous scenario. With an average retransmission delay of 462.3 ms and a maximum retransmission delay of 464.3 ms, all first fast retransmits are caused by the first possible SACK to report it lost.

By looking at the first retransmissions of both retransmission timeouts and fast retransmits as a whole, it gives an average retransmission delay of 495.8 ms. Compared to the last test, this is an improvement of 39.7 ms. There is no difference in the retransmission delay of bundled data chunks although the retransmission delay of retransmission timeouts are reduced.

Removing delayed SACKs

In this test scenario, we continue to use the same mechanisms as in the previous test in addition to remove the SACK delay at the receiver. The goal of the test is to show the removal of delayed SACKs influence on the retransmission delays as we have no timer implementation that could handle them. We are still using the same modifications as in the previous test. The results of the test is shown in table 5.1 and denoted **+no SACK delay**.

By looking at the first retransmissions caused by retransmission timeouts, we see an average retransmission delay of 282.9 ms. In comparision, the average retransmission delay of a first retransmission timeout with delayed SACKs is 525.0 ms which is a reduction of 241.2 ms. A removal of delayed SACKs leads to correct RTT measurements as the SACK is always sent back immediately independent of the high arrival time between each packet. The minimum retransmission delay is 255.7 and shows the lowest RTO value possible in this test scenario when the RTO calculations are based on correct RTT measurements of 200 ms. If the first SACK is lost, the RTT measurement is based on the next SACK that arrives 250 ms later. As shown in section 4.3.2, this leads to a rapid increase in the calculated RTO which allows a first fast retransmit to get triggered before the timer expires. As described earlier, the minimum time needed to trigger a fast retransmit is 450 ms in this scenario. Thus, with an average retransmission timeout delay of 282.9 ms, fast retransmits occurs rarely in a first retransmission unless the RTO value exceeds the time needed to trigger a fast retransmit.

In this situation, a first fast retransmit is triggered after 463.0 ms, before the timer expires. If a data chunk is successfully received, the removal of delayed SACKs makes it possible to acknowledge each data chunk close to RTT ms which in this case is 200 ms. With an average retransmission timeout of 282.9 ms in this test scenario, the absence of a SACK acknowledging the data chunk triggers the retransmission timeout 82.9 ms after the lowest possible time of getting an acknowledgment. This shows that the RTO calculations could get close to the RTT.

By looking at the first retransmissions of both retransmission timeouts and fast retransmits as a whole, we see an average retransmission delay of 283.2 ms. Compared to the last test, this is an improvement of 212.6 ms and shows that delayed SACKs play a major role on the retransmission delays. This also makes it possible to bundle data chunks earlier and the average retransmission delay of bundled data chunks has been decreased from 189.6 ms to 47.1 ms compared to the previous test scenario.

In the results of the test, we have been looking at the first retransmissions to see the effect of the proposed enhancements on the retransmission delays. As retransmission timeouts and fast retransmits trigger retransmissions, we will take a look at the cumulative retransmission delays when these are seen as a whole. Figure 5.2 shows the cumulative retransmission delays for each number of retransmissions in the previous tests compared to the original SCTP protocol. The figure shows that the retransmissions has been considerable reduced compared to the ori-

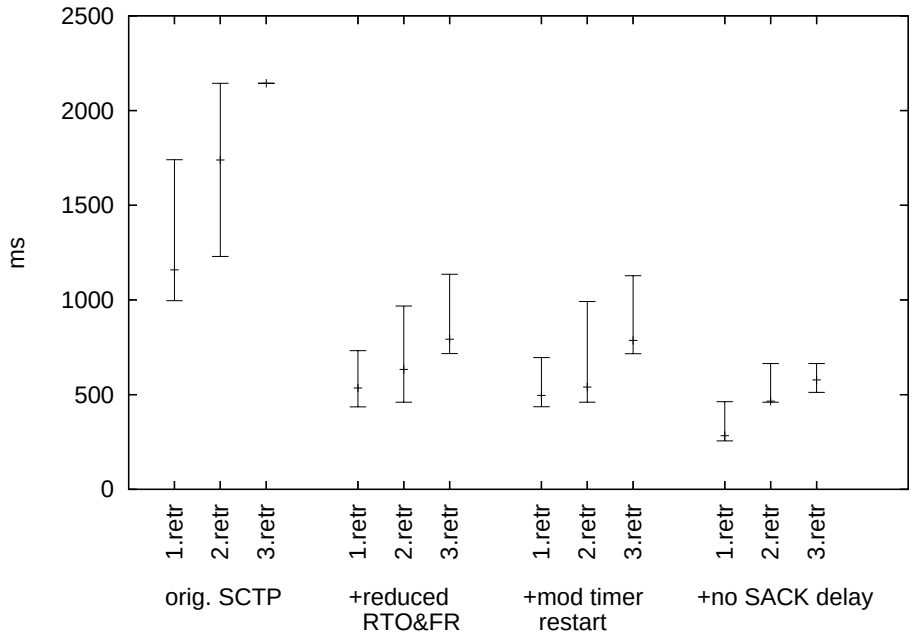

Figure 5.2: SCTP effect of proposed enhancements

ginal SCTP protocol.

By looking at the average retransmission delays of reduced RTO and modified fast retransmit only (denoted +**reduced RTO&FR**), they have been halved compared to the original SCTP protocol. The maximum retransmission delay of a third retransmission is reduced by about 1000 ms, which is considerable for interactive applications. By introducing the modified restart timer mechanism (denoted +**mod timer restart**), a small decrease in the retransmission delays can be seen. But it is not very noticeable in these tests. The best improvement is by removing delayed SACKs which further decreases the retransmission delays by a half compared to the other proposed mechanisms.

5.3.2 Modified fast retransmit

In the following two tests, we are considering the first retransmissions of each test to demonstrate the effect of the modified fast retransmit mechanism. The results of the tests are presented in table 5.2.

By looking at the results of the original protocol we see that the min-

Test Scenario	Type	Number	Min	Max	Avg
orig. SCTP	Retransmission timeout	286	996.2	1388.1	1163.9
	Fast retransmit	35	1255.4	1268.1	1260.9
	B:Reported missing	20	343.7	1224.1	764.3
	B:In flight	885	60.0	1192.0	248.1
	Retr timeout & FR	321	996.2	1388.1	1174.4
+reduce RTO+FR	Retransmission timeout	150	264.2	351.0	270.8
	Fast retransmit	243	216.9	254.2	223.8
	B: Reported missing	0	0	0	0
	B: In flight	185	213.9	334.5	241.7
	Retr timeout & FR	393	216.9	351.0	241.7
+reduce RTO+FR	Retransmission timeout	148	264.1	1071.9	279.4
+FR bundling	Fast retransmit	244	216.8	242.8	223.5
	B: Reported missing	0	0	0	0
	B: In flight	405	15.8	555.8	212.7
	Retr timeout & FR	402	216.9	351.0	241.7

Table 5.2: SCTP fast retransmit test results

imum retransmission delay of a retransmission timeout is 996.2 ms. This value is almost equal to the RTO minimum value of 1000 ms. The reason for that the retransmission delay is 3.8 ms below the minimum RTO could be because the conversion between jiffies and ms in the kernel does not always lead to exact values. This does not make any difference as the RTO is still close to the RTO minimum value of 1000 ms. The average retransmission delay of a first retransmission timeout is 1163.9 ms and confirms situations where the retransmission timer is delayed by late SACKs. In such cases, some fast retransmits has the time to get triggered by 4 SACKs before the retransmission timer expires with an average retransmission delay of 1260.9 ms. If the last data chunk in the burst of 4 data chunks is lost, then the 4 data chunks of the next burst triggers the 4 SACKs needed to fast retransmit. By considering first retransmission timeouts and fast retransmits as a whole, this leads to an average retransmission delay of 1174.4 ms. This value is close to 5 times higher compared to the thick test of the corresponding scenario in table 3.1 where the average retransmission delay of the first retransmissions is 208.0 ms.

By looking at the first retransmissions of the modified fast retransmit test denoted **+reduce RTO+FR**, we see an average retransmission delay of 216.9 ms when retransmission timeouts and fast retransmits are seen as a whole. In comparision, the corresponding average retransmission delay without the modified fast retransmit mechanism is 1260.9 ms and

the new mechanism gives a great reduction and is close to the minimum value of the thick stream scenario. The modified fast retransmit mechanism will take care of retransmitting the lost data chunk if it is sent as one of the three first in the burst of 4 data chunks. This is because the next data chunk, sent 10 ms later, will trigger the SACK needed to fast retransmit. By taking the RTT of 200 ms into account and other possible protocol and network delays, this leads to the minimum retransmission delay of 216.9 ms. The maximum retransmission delay of 254.2 ms occurs as the first data chunk in the burst is lost and the data chunk is fast retransmitted by a SACK triggered by the fourth consequtive data chunk. In total, this leads to an average retransmission delay of 223.8 ms. If the fourth data chunk in the burst is lost, then it must be retransmitted by a retransmission timeout as the next data chunk is sent 960 ms later. During this time, a retransmission timeout will take care of retransmitting the data chunk. The average retransmission delay of a retransmission timeout is 270.7 which is close to the average RTO when SACK delay is removed as described in section 5.3.1. Although delayed SACKs is beeing used in this test, bursts of four data chunks lead to more correct RTT measurements as the second or fourth data chunk in a burst triggers the second SACK which is sent back by waiting 10 ms, the time between each data chunk is sent. This leads to almost correct RTT measurements. By considering first retransmission timeouts and fast retransmits as a whole, this lead to an average retransmission delay of 241.7 ms which is a great reduction compared to the original protocol where the average retransmission delay is 1174.4 ms.

The third test considers bundling in fast retransmits. By looking at the first retransmissions of this test denoted **+reduce RTO+FR + FR bundling** in table 5.2, there are no significant difference between the average retransmission delays or the number of retransmissions except that the number of bundled data chunks has increased. Table 5.3 shows the number of bundled data chunks in the last two tests independent of the number of retransmissions. With bundling in fast retransmits enabled, there are 509 bundled data chunks. In comparison, without bundling in fast retransmit enabled, there are 204 bundled data chunks. Thus, when the modified fast retransmit mechanism is triggered often as in these tests, the number of bundled data chunks will increase considerable. As the stream is thin, there is always important to transfer as many data chunks as possible inside a packet if there is room according to the MTU.

Figure 5.3 shows the modified fast retransmit's effect on the cumulative retransmission delays when fast retransmit and retransmission timeout is seen as a whole. Compared to the original protocol, the modified

Without FR bundling	With FR bundling
204	509

Table 5.3: SCTP Number of bundled data chunks in fast retransmits

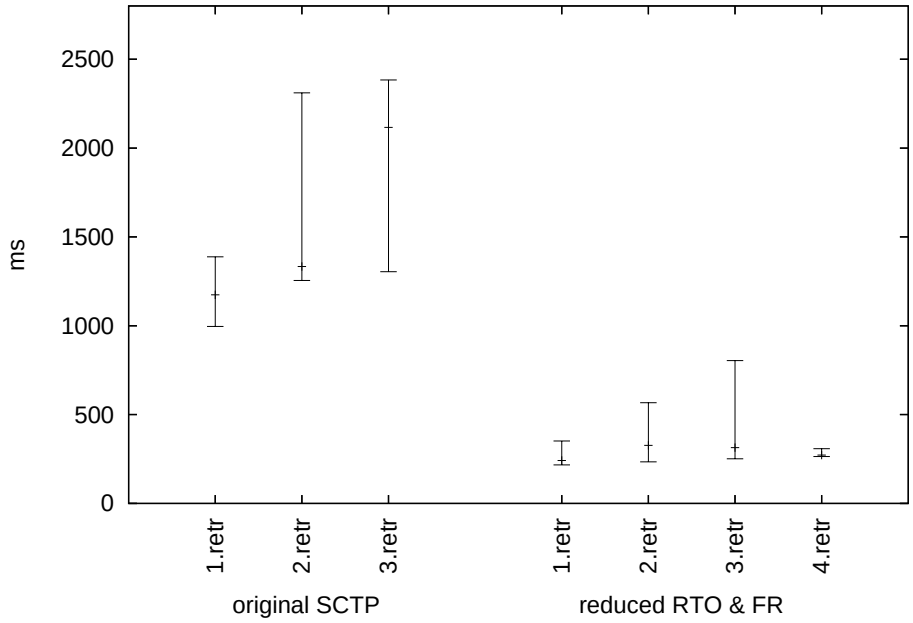

Figure 5.3: SCTP effect of modified fast retransmit

fast retransmit and the reduced retransmission timer alone reduces the retransmission delays drastically in all number of retransmissions. The maximum cumulative retransmssion delay is reduced from almost 2500 ms to below 1000 ms. This should be very considerable in interactive applications.

5.4 Evaluation

We have been running several tests that demonstrates the effect of the proposed enhancements in thin streams. We have seen that the new mechanisms are improving the retransmission delays compared to the original protocol. As the stream in the tests is thin, the improvements in retransmission delays show that the packets in flight algorithm is in-

voking the modified mechanisms as it is supposed to do. In the tests, we have been looking at the effect on the first retransmissions and especially the modified mechanism's influence on retransmission timeouts and fast retransmits in addition to bundling of outstanding data chunks.

Because of SCTP minimum RTO value of 1000 ms and the fact that 4 SACKs rarely have the chance to trigger a fast retransmit before the timer expires, the test results of the thin streams in section 3.2 show a minium retransmission delay of 1000 ms in all scenarios. This is a great increase compared to the similar scenario running thick streams. The nature of the thin stream used in the tests are able to trigger several SACKs, but 4 SACKs are mostly to much to trigger a fast retransmit in a first retransmission before the timer expires. However, the large time interval between the incoming SACKs are able to displace retransmission timeouts by restarting the timer. This lead to some occurrences of fast retransmits before the timer expires, but the retransmission delays are higher in average compared to retransmission timeouts.

The tests of the proposed enhancements, discussed in section 5.3.1, show that the most effective way of reducing the retransmission delays in thin streams is by decreasing the minimum RTO value to trigger retransmissions earlier. If the time between each sent data chunk is high, the retransmission timer will take care of retransmitting them as early as possible as the time needed to trigger a fast retransmit could be high. But the tests confirms that if the time between each sent data chunk is higher than the SACK delay of 200 ms, there is impossible to trigger retransmission timeouts below 435.8 although the minimum RTO value has been reduced to 200 ms. This confirms that in such scenarios, the RTO value is always 200 ms to high. In addition, the average RTO value could be considerably higher than the minimum value. This is critical for the retransmission delays in thin streams if no SACKs are able to trigger fast retransmits earlier.

Although the test including the modified restart timer mechanism has been shown to give improvements in the average retransmission timeout delays, the tests show that a removal of the SACK delay gives most improvements. In the tests, it is almost halving the retransmission delays and proves that SACK delay makes a big influence on the RTO calculations by increasing the retransmission delays considerably. Thus, if the stream is thin and SACK delay could not be removed at the receiver, a retransmission timer should be considered to handle this problem. If SACK delay is not taken into account, it leads to higher retransmission delays than necessary in the absence of SACKs that could trigger fast retransmits before the timer expires.

Reducing the number of SACKs needed to trigger a fast retransmit from 4 to 1 gives large improvements in the tests. The effect depends on the time interval between each data chunk. The test of the modified fast retransmit mechanism show that waiting for 4 SACKs takes too long time. If the nature of the thin stream is like sending some few data chunks in a burst, then the modified fast retransmit mechanism comes to its rights. It also reduces the retransmission delays when data chunks are sent in large intervals by triggering fast retransmits much earlier compared to the original protocol. In situations where a SACK arrives before the retransmission timer expires, the tests show that it is necessary to make use of the first indication of data chunk loss as it leads to lower retransmission delays.

5.5 Summary

In this chapter we have tested the proposed enhancements. The tests show that the proposed enhancements were invoked by the packets in flight mechanism. Thus, it shows that the packet in flight mechanism is working properly. By reducing the RTO value, the average retransmission delays reflected an over estimated RTO because of delayed SACKs. By introducing the modified restart timer mechanism, some decrease in the retransmission delays were seen although they were not considerable lower. The best effect was simply to remove delayed SACKs although it modifies the receiver. However, the modification could indicate the achieved retransmission delays if the retransmission timer considers delayed SACKs in its RTO calculations. The modified fast retransmit mechanism is very impressive when it came to its rights by lowering the retransmission delays considerably.

Chapter 6

Conclusion and remaining challenges

In this thesis we have investigated TCP NewReno and SCTP with respect to retransmission delays in thick and thin streams. The observations have been used to propose various enhancements in SCTP with the goal of improving the retransmission delays in thin streams.

6.1 Conclusion

The tests showed that TCP NewReno and SCTP have different ways to retransmit data in regards to how the retransmissions are handled and what is retransmitted in different situations.

In thick streams, TCP NewReno retransmissions are mostly caused by a fast retransmit. Because of its fast recovery mechanism, new packets are not allowed to be sent or retransmitted before an ACK acknowledges the earliest lost packet. If a packet is lost, it has to get retransmitted by a retransmission timeout which during situations of multiple loss grows exponentially. This affects the retransmission delays. The tests showed that the various combination of SACK mechanisms gives improvement on the retransmission delay without pointing out a winner.

In thick streams, the tests showed that SCTP retransmits almost all packets by a fast retransmit without triggering the retransmission timer at all. It was quickly discovered that SCTP has neither a fast recovery mechanism implemented nor a limit on how many times a data chunk

could get subsequently fast retransmitted without getting an acknowledgement. This lead to much lower retransmission delays compared to TCP NewReno. In addition, the subsequent fast retransmits are each cutting the congestion window in half and reduces the throughput drastically. This is unecessary as most retransmissions are spurious.

In thin streams, the tests showed that all retransmissions are triggered by the retransmission timer in TCP NewReno as the stream is so thin that three duplicate ACKs never have the chance to trigger fast retransmits before the timer expires. Thus, the various SACK extensions do not come to its rights. During multiple loss, this lead to several exponential backoffs of the retransmission timer which caused some extremly high retransmission delays.

SCTP has similar retransmission characteristics as TCP NewReno in thin streams, but there are also differences. Most retransmissions are triggered by the retransmission timer, but because of the high RTO minimum value, most first retransmission delays are higher, compared to TCP NewReno. Because of late arrivals of SACKs, the retransmission timer could get considerable delayed. This makes it possible to trigger some fast retransmits before the timer expires. In addition, SCTP triggers the retransmission timer and fast retransmit independently of each other. This avoids the extreme maximum values in TCP NewReno caused by exponential backoffs, but several retransmissions are spurious. In contrast to TCP NewReno, SCTP could retransmit some data earlier because of different bundling strategies. All in all, SCTP first retransmission delays are considerable higher than TCP NewReno's first retransmission delays because of the RTO minimum value. If the packet or data chunk is lost only once, TCP NewReno is preferred. As TCP NewReno suffers from several exponential backoffs, the suceeding retransmission delays are considerable higher than SCTP's suceeding retransmission delays. During multiple loss of the same packet or data chunk, SCTP is preferred although it leads to several spurious retransmissions beacuse of its retransmission handling.

To improve the retransmission delays in thin streams, several enhancements were proposed and implemented. This contain a mechanism to detect thin streams, fast retransmit after 1 SACK, bundling of data chunks in fast retransmits, reducing the RTO minimum value, correcting RTO value in timer restarts and eliminate the exponential backoff mechanism. In addition, bundling of outstanding data chunks with new data chunks has been proposed and implemented. The tests show that reducing the RTO value triggers retransmissions earlier, but because of delayed SACKs it was still considerable high. By removing delayed

SACKs in the lack of a retransmission timer handling their presence, there is possible to trigger retransmission even earlier. The danger is that this could lead to RTO values close to the RTT which could trigger spurious retransmissions without a retransmission timer handling this problem. But the retransmission delays of first retransmissions are almost 1/5 lower compard to the original SCTP protocol. The modified timer restart algorithm prevents the retransmission delays of going too high, but its effect is not very well seen in the tests because of the nature of the stream. The modified fast retransmit mechanism also improves the retransmission delays drastically when data chunks are sent in a way to let it come to it rights. Compared to the original protocol, the retransmission delays of first retransmissions is reduced to 1/5. In addition, allowing bundling in fast retransmits leads to a doubling in the number of bundled data chunks.

6.2 Remaining challenges

SCTP needs to handle retransmissions correctly by avoiding several fast retransmits of the same data chunk as it leads to a drastic reduce in throughput. In addition a fast recovery mechanism should be implemented to avoid subsequent lowerings of the congestion window during multiple loss scenarios. A way to incorporate the NewReno fast recovery mechanisms in SCTP is suggested by [16]. In addition, prevention of subsequent fast retransmits of the same data chunks and a fast recovery mechanism in SCTP is suggested by [1].

The SCTP retransmission timer should be changed to handle RTO calculations more correctly, especially to handle delayed SACKs and rapid changes in the RTT measurements when the stream is thin. The Eifel Retransmission Timer [18] solves several of these problems and should be considered implemented into SCTP.

We have not been looking at the multi-homing and multiple streams features of SCTP. Since each destination address has its own retransmission timer, one approach could be to always send data to the address with the lowest RTO. Thus, SCTP will always try to retransmit as early as possible when a retransmission timeout occurs.

The nature of the thin streams used in tests are very limited. They will not manage to cover all scenarios the proposed mechanisms is designed for. Thus, they should be further tested, especially with real applications in real networks.

Bibliography

[1] IETF SCTP Implementors Guide 16. http://www3.ietf.org/proceedings/06mar/IDs/draft-ietf-tsvwg-sctpimpguid%e-16.txt.

[2] Linux kernel SCTP implementation. http://lksctp.sourceforge.net/.

[3] Netem. http://linux-net.osdl.org/index.php/Netem.

[4] Netperf. http://www.netperf.org/netperf/NetperfPage.html.

[5] Pcap - packet capture library. http://www.tcpdump.org/pcap3_man.html.

[6] RFC 2018 - TCP Selective Acknowledgment Options. http://http://www.rfc.net/rfc2018.html.

[7] RFC 2883: An extension to the selective acknowledgement (SACK) option for tcp. http://rfc.net/rfc2883.html.

[8] RFC 3782: The NewReno Modification to TCP's Fast Recovery Algorithm. http://www.rfc.net/rfc3782.html.

[9] RFC 793: Transmission Control Protocol (TCP). http://rfc.net/rfc793.

[10] RFC2581: TCP Congestion Control. http://rfc.net/rfc2581.html.

[11] RFC2960 Stream Control Transmission Protocol (SCTP). http://www.rfc.net/rfc2960.html.

[12] RFC2988: Computing TCP's Retransmission Timer. http://rfc.net/rfc2988.html.

[13] Sctpperf. http://drakkar.imag.fr/~phadam/work/sctpperf/index.html.

[14] Tc man page. http://lartc.org/manpages/tc.txt.

[15] Tcpdump. http://www.tcpdump.org.

[16] A. Caro, K. Shah, J. Iyengar, P. Amer, and R. Stewart. SCTP and TCP variants: Congestion control under multiple losses. Technical Report TR2003-04, CIS Dept, University of Delaware, February 2003.

[17] C. Griwodz and P. Halvorsen. The Fun of using TCP for an MMORPG. Technical report. http://heim.ifi.uio.no/~jonped/funtrace.pdf.

[18] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communications Review*, 30(3), July 2000.

[19] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP Congestion Control. *Association for Computer Machinery (ACM)*, 26(4):11, 1996.

Appendix A

Code for new SCTP routines

A.1 add_pkt_in_flight()

```
1  void add_pkt_in_flight(struct sctp_association *a, __u32 tsn){

     if(a->pkt_in_flight_head == NULL){
       a->pkt_in_flight_head = (struct pkt_in_flight*)
5                              kmalloc(sizeof(struct pkt_in_flight),
                               GFP_KERNEL);
       a->pkt_in_flight_head->highestTSN = tsn;
       a->pkt_in_flight_head->next = NULL;
       a->pkt_in_flight_head->prev = NULL;
10     a->pkt_in_flight_tail = a->pkt_in_flight_head;
     }
     else{
       a->pkt_in_flight_tail->next = (struct pkt_in_flight*)
                                     kmalloc(sizeof(struct pkt_in_flight),
15                                   GFP_KERNEL);
       a->pkt_in_flight_tail->next->highestTSN = tsn;
       a->pkt_in_flight_tail->next->prev = a->pkt_in_flight_tail;
       a->pkt_in_flight_tail = a->pkt_in_flight_tail->next;
       a->pkt_in_flight_tail->next = NULL;
20   }
     a->packets_in_flight++;
   }
```

A.2 remove_pkts_in_flight()

```
1  void remove_pkts_in_flight(struct sctp_association *a, __u32 sack_cum_tsn){

     struct pkt_in_flight *pkt, *pkt_temp, *pkt_next;

5    for(pkt = a->pkt_in_flight_head; pkt != NULL; pkt = pkt_next){
```

```
    if(pkt->highestTSN <= sack_cum_tsn){

      if(pkt == a->pkt_in_flight_head){
        /* Removing head of list */

        a->pkt_in_flight_head = pkt->next;

        if(a->pkt_in_flight_head != NULL){
          a->pkt_in_flight_head->prev = NULL;
        }
        pkt->next = NULL;
        kfree(pkt);
        pkt_next = a->pkt_in_flight_head;
      }
      else if(pkt == a->pkt_in_flight_tail){
        /* Removing tail of list */

       if(pkt->prev != NULL){
          a->pkt_in_flight_tail = pkt->prev;
          pkt->prev->next = NULL;
       }
        pkt->prev = NULL;
        kfree(pkt);
        pkt_next = NULL;
      }
      else{
        /* Removing an inbetween element */
        pkt->prev->next = pkt->next;
        pkt->next->prev = pkt->prev;
        pkt_temp = pkt->next;
        pkt->next = NULL;
        pkt->prev = NULL;
        kfree(pkt);
        pkt_next = pkt_temp;
      }
      a->packets_in_flight--;
    }
    else{
      pkt_next = pkt->next;
    }
  }

  /* Do not count packets that already have left the network */

  a->packets_in_flight += a->packets_left_network;
  a->packets_left_network = 0; /* Reset variable */
}
```

A.3 check_stream_before_add()

```
int check_stream_before_add(struct sctp_transport *t,
                            struct sctp_chunk *chunk,
                            __u8 fast_retransmit){

  if(t->asoc->packets_in_flight < 5)
    return ((fast_retransmit && chunk->fast_retransmit) ||
              !chunk->tsn_gap_acked);
```

```
  else
    return ((fast_retransmit && chunk->fast_retransmit) ||
            (!fast_retransmit && !chunk->tsn_gap_acked));
}
```

A.4 Modified restart timer algorithm

```
if (restart_timer) {

  oldest_outstanding_chunk = NULL;

  list_for_each(list_chunk, &tlist){
    cur_chunk = list_entry(list_chunk, struct sctp_chunk,
                           transmitted_list);

    if(sctp_chunk_is_data(cur_chunk)){
      if(!cur_chunk->tsn_gap_acked){
        oldest_outstanding_chunk = cur_chunk;
        break;
      }
    }
  }

  if(oldest_outstanding_chunk != NULL){
    if (!mod_timer(&transport->T3_rtx_timer,
                   (jiffies -
                   (jiffies - oldest_outstanding_chunk->sent_at)
                   + transport->rto)))
      sctp_transport_hold(transport);
  }
  else{
    if (!mod_timer(&transport->T3_rtx_timer,
                   jiffies + transport->rto))
      sctp_transport_hold(transport);
  }
}
```

A.5 bundle_outstanding_chunks()

```
void bundle_outstanding_chunks(struct sctp_packet *packet,
                               struct sctp_transport *transport){

  size_t packet_size, pmtu, outstanding_chunks_size = 0;
  struct sctp_chunk *chunk;
  struct list_head *chunk_list, *list_head;
  struct list_head outstanding_list;
  int bundling_performed = 0;
  __u16 chunk_len;

  if(transport == NULL){
    return;
  }
```

```
15    list_for_each(chunk_list, &packet->chunk_list){
        chunk = list_entry(chunk_list, struct sctp_chunk,
                           list);

        if(sctp_chunk_is_data(chunk)){
20        if(chunk->has_tsn)
            return;
          else
            break;
        }
25    }

      packet_size = packet->size;

      pmtu  = ((packet->transport->asoc) ?
30             (packet->transport->asoc->pmtu) :
               (packet->transport->pmtu));

      INIT_LIST_HEAD(&outstanding_list);

35    list_for_each(list_head, &transport->transmitted){

        chunk = list_entry(list_head, struct sctp_chunk,
                           transmitted_list);

40      if(sctp_chunk_is_data(chunk)){
          if(chunk->has_tsn){

            chunk_len = WORD_ROUND(ntohs(chunk->chunk_hdr->length));

45          if((packet_size + chunk_len) > pmtu){
              break;
            }
            else{
              packet_size += chunk_len;
50            outstanding_chunks_size += chunk_len;
              chunk->transport = packet->transport;
              list_add_tail(&chunk->list, &outstanding_list);
              bundling_performed = 1;
            }
55        }
        }
      }

      if(bundling_performed){
60      list_splice(&outstanding_list, &packet->chunk_list);
        packet->size += outstanding_chunks_size;
      }
    }
```

Appendix B

Sctp_trace source code

B.1 sctp_trace.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pcap.h>
#include <errno.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h> /* includes net/ethernet.h */
#include "sctp_trace.h"

struct timeval global_time;
struct datachunk datachunks[MAX_CHUNKS];

/* Hold retransmissions of specific number and type*/
/* 0 = FR+RTO */
/* 1 = Retransmission Timeout */
/* 2 = Fast Retransmit */
/* 3 = Bundled reporting missing */
/* 4 = Bundled in flight */
struct retr retr[5][MAX_RETRANSMISSIONS];

/* Max retransmissions of a data chunk encountered */
int max_retransmissions = 0;

double calc_timediff(const struct timeval *t1, const struct timeval *t2){

  double sec_diff, msec_diff;

  sec_diff = (double)((t2->tv_sec - t1->tv_sec) * 1000);
  msec_diff = ((double)(t2->tv_usec) - (double)(t1->tv_usec)) / 1000;

  return sec_diff + msec_diff;
}
```

```
/* Update retransmission statistics for a specific retransmission*/

void update_retr(double cumulative_retrtime,
                 struct retr *retransmission,
                 u_int32_t TSN){

  retransmission->retransmissions++;

  retransmission->avg_cumulative_retrtime =
    ((retransmission->avg_cumulative_retrtime *
      (retransmission->retransmissions - 1))
    + datachunks[TSN].cumulative_retrtime) / retransmission->↩
        ↪retransmissions;

  if(retransmission->retransmissions < 2){

    retransmission->min_cumulative_retrtime = datachunks[TSN].↩
        ↪cumulative_retrtime;
    retransmission->max_cumulative_retrtime = datachunks[TSN].↩
        ↪cumulative_retrtime;
  }
  else{
    if(datachunks[TSN].cumulative_retrtime > retransmission->↩
        ↪max_cumulative_retrtime){
      retransmission->max_cumulative_retrtime = datachunks[TSN].↩
          ↪cumulative_retrtime;
    }
    if(datachunks[TSN].cumulative_retrtime < retransmission->↩
        ↪min_cumulative_retrtime){
      retransmission->min_cumulative_retrtime = datachunks[TSN].↩
          ↪cumulative_retrtime;
    }
  }
}

/* Handles a data chunk retransmission in the modified SCTP protocol for ↩
    ↪thin streams
   Also allows bundling of chunks in fast retransmits.
*/

void handle_chunk_mod_retransmission(int firstInPacket,
                                     double retr_time, u_int32_t TSN,
                                     const struct pcap_pkthdr *header, int ↩
                                         ↪*is_rto){
  double first_chunk_time;
  int retr_nr = datachunks[TSN].retransmissions;

  if(firstInPacket){
    if(datachunks[TSN].sacks < 1){

      /* Retransmission Timeout */
      *is_rto = 1;
      printf("␣Retr␣Timeout\n");
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[1][retr_nr], ↩
          ↪TSN);
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[0][retr_nr], ↩
          ↪TSN);
    }
    else{
      /* Fast retransmit */
```

```
      printf("␣Fast␣Retransmit\n");
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[2][retr_nr], ↩
          ↪TSN);
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[0][retr_nr], ↩
          ↪TSN);
    }
  }
  else{
    /* Bundled chunks in retransmission timeout and modified fast ↩
        ↪retransmit */

    if(datachunks[TSN].sacks == 0){
      printf("␣In␣flight\n");
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[4][retr_nr], ↩
          ↪TSN);
    }
    else{
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[2][retr_nr], ↩
          ↪TSN);
      update_retr(datachunks[TSN].cumulative_retrtime, &retr[0][retr_nr], ↩
          ↪TSN);
    }
  }
}

/* Handle a data chunk retransmission in the original SCTP protocol */
void handle_chunk_retransmission(int firstInPacket,
                                 double retr_time, u_int32_t TSN,
                                 const struct pcap_pkthdr *header, int *↩
                                     ↪is_rto){
  double first_chunk_time;
  int retr_nr = datachunks[TSN].retransmissions;

  if(firstInPacket){
    if(datachunks[TSN].sacks < SACK_THRESHOLD){

      /* Retransmission Timeout */
      *is_rto = 1;
      printf("␣Retr␣Timeout\n");

      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[1][retr_nr], TSN);

      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[0][retr_nr], TSN);
    }
    else{
      /* Fast retransmit */
      printf("␣Fast␣Retransmit\n");
      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[2][retr_nr], TSN);

      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[0][retr_nr], TSN);
    }
  }
  else{
    /* Bundling */
    if(*is_rto == 1){
      if(datachunks[TSN].sacks == 0){
        printf("␣In␣flight\n");
        update_retr(datachunks[TSN].cumulative_retrtime,
```

```
                    &retr[4][retr_nr], TSN);
      }
      else{
        printf("␣Reported␣missing\n");
        update_retr(datachunks[TSN].cumulative_retrtime,
                    &retr[3][retr_nr], TSN);
      }
    }
    /* If first is not a retransmission timeout, bundled chunk is a fast ↩
        ↪retransmit */
    else{
      printf("␣Fast␣Retransmin\n");
      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[2][retr_nr], TSN);
      update_retr(datachunks[TSN].cumulative_retrtime,
                  &retr[0][retr_nr], TSN);
    }
  }
  datachunks[TSN].sacks = 0; /* Reset SACK missing report */
}

/* Sniff packet:*/

void sniff_packet(const struct pcap_pkthdr *header,
                  const u_char *packet){

  static u_int32_t firstTSN, expectedTSN;
  double retr_time;

  const struct sniff_ip *ip; /* IP header */
  const struct sctpHeader *sctpPacketHeader;  /* SCTP packet header*/
  const struct sctpChunkDesc *chunkDescPtr; /* chunk header */
  const void *sctpPacketEnd; /* End of SCTP packet */
  const u_char *chunkEnd; /* End of chunk */
  const struct sctpDataPart *dataChunkHeader;
  const void *nextChunk; /* Next chunk in packet */

  static int first_datachunk = 1; /* First data chunk encountered? */
  int firstInPacket = 1; /* First data chunk in a packet?*/
  int isRTO = 0; /* Is this a retransmission timeout ?*/

  double timediff;

  u_int8_t chunkID;
  u_int sctpPacketLength;
  u_int size_ipheader;
  u_int16_t chunkLength;
  u_int16_t align;

  u_int32_t TSN;

  if(header->caplen < header->len){
    printf("Error:␣Packet␣is␣truncated,␣need␣whole␣packet␣to␣extract␣SCTP␣↩
        ↪headers\n");
    exit(1);
  }

  /* Extract packet and chunk headers */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ipheader = IP_HL(ip)*4;

  sctpPacketLength = header->len - (SIZE_ETHERNET + size_ipheader);
```

```
    sctpPacketHeader = (const struct sctpHeader*)(packet + SIZE_ETHERNET + ↩
        ↪size_ipheader);
    sctpPacketEnd = (const u_char*) sctpPacketHeader + sctpPacketLength;
205
    chunkDescPtr = (const struct sctpChunkDesc *)
      ((const u_char*) sctpPacketHeader + sizeof(struct sctpHeader));

    /* Pring global time: */
210 timediff = calc_timediff(&global_time, &header->ts);
    printf("[%.3f]", timediff);

    /* Traverse chunks in packet */

215 while(chunkDescPtr != NULL && ((const void *)((const u_char *) ↩
        ↪chunkDescPtr
                                    + sizeof(struct sctpChunkDesc))) <= ↩
                                        ↪sctpPacketEnd){

      chunkLength = EXTRACT_16BITS(&chunkDescPtr->chunkLength);
      chunkID = chunkDescPtr->chunkID;
220   chunkEnd = ((const u_char*)chunkDescPtr + chunkLength);

      /* Handle data chunks */

      if(chunkID == SCTP_DATA){
225
        dataChunkHeader=(const struct sctpDataPart*)(chunkDescPtr + 1);
        TSN = EXTRACT_32BITS(&dataChunkHeader->TSN); /* Extract TSN */

        if(chunkLength == 0){
230       printf("ERROR:␣CHUNK␣LENGTH␣=␣0\n");
          break;
        }

        /* If the first data chunk discovered, set the inital TSN */
235     if (first_datachunk){

          expectedTSN = TSN + 1;
          firstTSN = TSN;

240       /* Reset retransmission statistics */
          datachunks[TSN - firstTSN].retransmissions = 0;
          datachunks[TSN - firstTSN].sacks = 0;
          datachunks[TSN - firstTSN].cumulative_retrtime = 0;

245       printf("␣TSN:␣%u\n", (TSN-firstTSN));

          first_datachunk = 0;
        }

250     /* Handle retransmission */
        else if (TSN < expectedTSN){

          retr_time = calc_timediff(&datachunks[TSN - firstTSN].timestamp,
                                    &header->ts);
255
          if(retr_time > 0){

            if(firstInPacket) printf("␣RETRANSMISSION:\n");
            datachunks[TSN - firstTSN].retransmissions++;
260
```

```
            if(datachunks[TSN - firstTSN].retransmissions > ↩
                ↪max_retransmissions){
              max_retransmissions = datachunks[TSN - firstTSN].↩
                  ↪retransmissions;
            }

            datachunks[TSN - firstTSN].cumulative_retrtime += retr_time;

            printf("\t R%d %u (%.3fms)",
                   datachunks[TSN - firstTSN].retransmissions,
                   (TSN-firstTSN),
                   datachunks[TSN - firstTSN].cumulative_retrtime);

            /* Call routine based on if we are testing the original SCTP
               or modified SCTP.
            */

            if(!TEST_MODIFIED_SCTP){
              handle_chunk_retransmission(firstInPacket,
                                          retr_time, (TSN-firstTSN),
                                          header, &isRTO);
            }
            else{
              handle_chunk_mod_retransmission(firstInPacket,
                                              retr_time, (TSN-firstTSN),
                                              header, &isRTO);
            }
            if(firstInPacket) firstInPacket = 0;
          }
          else{
            // Timestamp failure, throw packet
            break;
          }

        }

        /* A new data chunk, reset retransmission statistics */
        else{
          printf(" TSN: %u\n", (TSN-firstTSN));
          datachunks[TSN - firstTSN].retransmissions = 0;
          datachunks[TSN - firstTSN].sacks = 0;
          datachunks[TSN - firstTSN].cumulative_retrtime = 0;
          expectedTSN = TSN + 1;
        }

        /* Set timestamp of when data chunk is transmitted */
        datachunks[TSN - firstTSN].timestamp.tv_sec = header->ts.tv_sec;
        datachunks[TSN - firstTSN].timestamp.tv_usec = header->ts.tv_usec;
      }

      else if(chunkID == SCTP_SELECTIVE_ACK){

        const struct sctpSelectiveAck *sack;
        const struct sctpSelectiveFrag *gapblock;
        u_int16_t numberOfdesc, startBlock, endBlock;
        u_int32_t gapStart, gapEnd;
        int blockNo, j;


        /* Need to start retransmission statistics with a data chunk */
        if(first_datachunk){
          break;
```

```
      }

      /* Extract SACK chunk*/
      sack=(const struct sctpSelectiveAck*)(chunkDescPtr+1);

      /* Get number of gap ack blocks */
      numberOfdesc = EXTRACT_16BITS(&sack->numberOfdesc);

      printf("␣SACK␣CUM␣ACK:␣%u␣#GAP␣ACKs:␣%u␣",
             (EXTRACT_32BITS(&sack->highestConseqTSN)-firstTSN),
             numberOfdesc);

      if(numberOfdesc > 0) {
        /* Print gap ack blocks */
        printf("GAP␣ACKED:␣");

        gapblock = (const struct sctpSelectiveFrag *)((const struct ↩
            ↪sctpSelectiveAck *) sack+1);
        gapStart = (EXTRACT_32BITS(&sack->highestConseqTSN)-firstTSN) + 1;

        for(blockNo = 0; blockNo < numberOfdesc; blockNo++){

          startBlock = EXTRACT_16BITS(&gapblock->fragmentStart)
            + (EXTRACT_32BITS(&sack->highestConseqTSN)-firstTSN);

          endBlock = EXTRACT_16BITS(&gapblock->fragmentEnd)
            + (EXTRACT_32BITS(&sack->highestConseqTSN)-firstTSN);

          for(j = gapStart; j < endBlock; j++){
            /* Update SACK missing report of data chunk */
            datachunks[j].sacks++;
          }
          gapStart = endBlock+1;

          printf("%u-%u␣", startBlock, endBlock);
          gapblock++;
        }
      }

      printf("\n");
      break;
    }

    else{

      /* Print other chunk types */

      if(chunkID == 1) printf("␣␣INIT\n");
      else if(chunkID == 2) printf("␣␣INIT␣ACK\n");
      else if(chunkID == 4) printf("␣␣HEARTBEAT\n");
      else if(chunkID == 5) printf("␣␣HEARTBEAT␣ACK\n");
      else if(chunkID == 6) printf("␣␣ABORT\n");
      else if(chunkID == 7) printf("␣␣SHUTDOWN\n");
      else if(chunkID == 8) printf("␣␣SHUTDOWN␣ACK\n");
      else if(chunkID == 9) printf("␣␣ERROR\n");
      else if(chunkID == 10) printf("␣␣COOKIE␣ECHO\n");
      else if(chunkID == 11) printf("␣␣COOKIE␣ACK\n");
      else if(chunkID == 12) printf("␣␣ECNE\n");
      else if(chunkID == 13) printf("␣␣CWR\n");
      else if(chunkID == 14) printf("␣␣SHUTDOWN␣COMPLETE\n");
      else printf("\n");
      break;
```

```
      }

      /* Chunk length must be a multiple of 4 bytes,
         max alignment = 3 bytes */

      align = chunkLength % 4;
      if (align != 0){
        align = 4 - align;
      }
      /* Find pointer to next chunk in packet */
      nextChunk = (const void *) (chunkEnd + align);
      chunkDescPtr = (struct sctpChunkDesc *)((u_char*)  nextChunk);
    }

    fflush(stdout);
  }


  /* Reset retransmission statistics */
  void reset_retr(){

    int i = 0, j =0;

    for(i=0; i< 5; i++){
      for(j=1; j<MAX_RETRANSMISSIONS; j++){
        retr[i][j].avg_cumulative_retrtime = 0;
        retr[i][j].min_cumulative_retrtime = 0;
        retr[i][j].max_cumulative_retrtime = 0;
      }
    }
  }

  /* Print retransmission statistics */
  void print_retr(){

    int i = 1;

    if(TEST_MODIFIED_SCTP){
      printf("\n MODIFIED ");
    }
    else{
      printf("\n");
    }

    printf("SCTP RETRANSMISSION STATS SORTED BY RETRANSMISSION NR AND TYPE:\n↩
        ↪");

    for(i = 1; i <= max_retransmissions; i++){

      printf("\n%d. RETRANSMISSION:\n\n", i);
      printf("RTO                (%d): min = %.1f ms, max = %.1f ms, avg = %.↩
          ↪1f ms\n",
             retr[1][i].retransmissions,
             retr[1][i].min_cumulative_retrtime,
             retr[1][i].max_cumulative_retrtime,
             retr[1][i].avg_cumulative_retrtime);

      printf("Fr                 (%d): min = %.1f ms, max = %.1f ms, avg = %.↩
          ↪1f ms\n",
             retr[2][i].retransmissions,
             retr[2][i].min_cumulative_retrtime,
             retr[2][i].max_cumulative_retrtime,
```

```
            retr[2][i].avg_cumulative_retrtime);

      printf("B: Reported missing (%d): min = %.1f ms, max = %.1f ms, avg = %.1f ms\n",
            retr[3][i].retransmissions,
            retr[3][i].min_cumulative_retrtime,
            retr[3][i].max_cumulative_retrtime,
            retr[3][i].avg_cumulative_retrtime);
      printf("B: In flight        (%d): min = %.1f ms, max = %.1f ms, avg = %.1f ms\n",
            retr[4][i].retransmissions,
            retr[4][i].min_cumulative_retrtime,
            retr[4][i].max_cumulative_retrtime,
            retr[4][i].avg_cumulative_retrtime);

      printf("FR + RTO:           (%d): min = %.1f ms, max = %.1f ms, avg = %.1f ms\n",
            retr[0][i].retransmissions,
            retr[0][i].min_cumulative_retrtime,
            retr[0][i].max_cumulative_retrtime,
            retr[0][i].avg_cumulative_retrtime);
    }

  }

  int main(int argc, char *argv[]){

    /* Check if first packet to set global time */
    int first = 1;

    if(argc != 2){
      fprintf(stderr, "Usage: %s <SCTP pcap tracefile>\n", argv[0]);
      exit(1);
    }

    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* file = pcap_open_offline(argv[1], errbuf);

    if(file == NULL){
      fprintf(stderr, "Couldn't open tracefile '%s'\n", argv[1]);
      exit(1);
    }
    /* Reset retransmission statistics */
    reset_retr();

    struct pcap_pkthdr h;
    const u_char *data;

    /* Sniff each packet in pcap tracefile: */
    do {
      data = (const u_char *)pcap_next(file, &h);
      if(data == NULL){
        pcap_perror(file, "\nNo more data on file\n");
      }
      else{
        if(first){

          /* Store the time of the first packet as global time */

          global_time.tv_sec = h.ts.tv_sec;
          global_time.tv_usec = h.ts.tv_usec;
          first = 0;
```

```
500         }
            sniff_packet(&h, data); /* Sniff packet */
          }
        }
        while(data != NULL);
505
        /* Print retransmission statistics to screen */
        print_retr();

        /* Close pcap file */
510     pcap_close(file);

        return 0;

    }
```

B.2 sctp_trace.h

```
1   #define SIZE_ETHERNET 14
    #define MAX_CHUNKS 1000000
    #define MAX_RETRANSMISSIONS 100
    #define SACK_THRESHOLD 4
5
    /* Are we testing the modified SCTP protocol for thin streams? */
    #define TEST_MODIFIED_SCTP 0

    #define SCTP_DATA               0x00  /* Chunk ID for data chunk */
10  #define SCTP_SELECTIVE_ACK      0x03  /* Chunk ID for SACK chunk */

    #define EXTRACT_16BITS(p) \
            ((u_int16_t)ntohs(*(const u_int16_t *)(p)))
    #define EXTRACT_32BITS(p) \
15          ((u_int32_t)ntohl(*(const u_int32_t *)(p)))

    /* IP header */
    struct sniff_ip {
      u_char ip_vhl;/* version << 4 | header length >> 2 */
20    u_char ip_tos;/* type of service */
      u_short ip_len;/* total length */
      u_short ip_id;/* identification */
      u_short ip_off;/* fragment offset field */
    #define IP_RF 0x8000/* reserved fragment flag */
25  #define IP_DF 0x4000/* dont fragment flag */
    #define IP_MF 0x2000/* more fragments flag */
    #define IP_OFFMASK 0x1fff/* mask for fragmenting bits */
      u_char ip_ttl;/* time to live */
      u_char ip_p;/* protocol */
30    u_short ip_sum;/* checksum */
      struct in_addr ip_src,ip_dst; /* source and dest address */
    };

    #define IP_V(ip)        (((ip)->ip_vhl & 0xf0) >> 4)
35  #define IP_HL(ip)       ((ip)->ip_vhl & 0x0f)

    /* Retransmission statistics for a specific retransmission */
    struct retr{
```

```
  double avg_cumulative_retrtime;
  double min_cumulative_retrtime;
  double max_cumulative_retrtime;
  int retransmissions;
};

/* Holds retransmission information for a data chunk*/
struct datachunk{

  /*TSN of this DATA chunk */
  u_int32_t TSN;

  /* Number of retransmissions of this DATA chunk */
  int retransmissions;

  /* Number of SACKs reporting this chunk to be lost */
  int sacks;

  /* The cumulative retransmission time of this DATA chunk */
  double cumulative_retrtime;

  /* Timestamp of when DATA chunk is transmitted */
  struct timeval timestamp;
};

/* SCTP packet header */
struct sctpHeader{
  u_int16_t source;
  u_int16_t destination;
  u_int32_t verificationTag;
  u_int32_t adler32;
};

/* SCTP Chunk header*/
struct sctpChunkDesc{
  u_int8_t chunkID;
  u_int8_t chunkFlg;
  u_int16_t chunkLength;
};

/* SCTP Data Chunk header */
struct sctpDataPart{
  u_int32_t TSN;
  u_int16_t streamId;
  u_int16_t sequence;
  u_int32_t payloadtype;
};

/* SCTP SACK chunk header*/
struct sctpSelectiveAck{
  u_int32_t highestConseqTSN;
  u_int32_t updatedRwnd;
  u_int16_t numberOfdesc;
  u_int16_t numDupTsns;
};

/* SACK Gap ack block */
struct sctpSelectiveFrag{
  u_int16_t fragmentStart;
  u_int16_t fragmentEnd;
};
```