Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Daniel Sarnow

SACK Handling for CMT-SCTP in the FreeBSD Kernel

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Sarnow

SACK Handling for CMT-SCTP in the FreeBSD Kernel

Bachelor Thesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Franz-Josef Korf

Eingereicht am: 21. März 2017

Daniel Sarnow

Thema der Arbeit

SACK Handling for CMT-SCTP in the FreeBSD Kernel

Stichworte

Concurrent Multipath Transfer, Stream Control Transmission Protocol, SACK Handling, Ungleiche Pfade, Asymmetrische Pfade, Performance Analyse

Kurzzusammenfassung

In den letzten Jahren hat die Anzahl der mobilen Geräte, wie zum Beispiel Smartphones, die über mehrere Netzwerkschnittstellen verfügen, drastisch zugenommen. Als Folge dieser Enwicklungen hat der Wunsch zugenommen alle verfügbaren Netzwerkschnittstellen für die Übertragung von Applikationsdaten zu nutzen. Die gleichzeitige Nutzung aller verfügbarer Netzwerkschnittstellen für die Datenübertragung kann nicht nur den Durchsatz erhöhen, sondern auch einen besseren Schutz gegen Netzwerkausfälle gewährleisten.

Concurrent Multipath Transfer (CMT) ist eine Erweiterung des Transportprotokolls SCTP, welches die gleichzeitige Nutzung aller Netzwerkschnittstellen erlaubt.

Sollten sich die Eigenschaften eines Kommunikationspfades (wie zum Beispiel Bandbreite, Delay oder Fehlerrate) jedoch ändern, kann der Durchsatz in einem ungleichen Multi-Path Szenario stark abfallen. Es kann sogar so weit kommen, dass der Durchsatz nur noch einen Bruchteil des möglichen Durchsatzes erreicht, als wenn nur der beste Pfad für die Applikationsdatenübertragung genutzt werden würde.

Da mobile Geräte oft in einer Umgebung genutzt werden, in der ungleiche Kommunikationspfade nicht selten sind, ist die effiziente Nutzung von CMT sehr entscheidend. Des Weiteren wird die erste von drei Design Regeln, welche von [1] vorgeschlagen wurden, verletzt. Die erste Design Regel besagt, dass ein Multi-Path Flow wenigstens einen so guten Durchsatz erreichen soll, wie ein Single-Path Flow über den besten Pfad.

In dieser Bachelorarbeit werden die Herausforderungen, die durch Load-Sharing mit CMT-SCTP über ungleiche Pfade entstehen, dargestellt. Die Gründe für die verminderte Leistung werden diskutiert und Lösungskonzepte werden angeboten[1].

Für die Untersuchung von CMT-SCTP wird ein Testbed genutzt, welches größtenteils eine voll kontrollierbare Umgebung schafft. Das Testbed erlaubt es den Datenverkehr zwischen

[1] The Ergebnisse dieser Arbeit werden über folgendes GitHub Repository zur Verfügung gestellt: https://github.com/DanSar/ba_sctp

den kommunizierenden Endpunkten bzgl. der Linkeigenschaften zu verändern. Des Weiteren werden Router Statistiken, Mitschnitte von Netzwerkverkehr und Zeitmessungen von Kernelroutinen zur Analyse genutzt.

Die Analyse der aktuellen Implementierung zeigt, dass die Herausforderungen nicht nur mit der CPU Leistung zusammenhängen. Somit reicht es nicht nur Lösungskonzepte anzubieten, die algorithmisch Effizeienzprobleme lösen ohne die SCTP Spezifikation zu ändern.

Die Bachelorarbeit nutzt die bei der IETF als Referenzimplementierung genutzte SCTP Implementierung in FreeBSD.

Daniel Sarnow

Title of the paper
SACK Handling for CMT-SCTP in the FreeBSD Kernel

Keywords
Concurrent Multipath Transfer, Stream Control Transmission Protocol, SACK handling, Dissimilar Paths, Asymmetric Paths, Performance Analysis

Abstract
Over the last few years the amount of mobile devices with multiple network interfaces, such as smartphones, has increased dramatically. As a result of this development the urge to use *all* available network interfaces has emerged. The simultaneous use of network interfaces allows an increase in application data throughput and improves resilience to network failure.

Concurrent Multipath Transfer (CMT) is an extension to the transport protocol SCTP that allows the simultaneous use of multiple network paths between two peers.

However, if the characteristics (i.e. bandwidth, delay and error rate) of these paths change to be more dissimilar the overall throughput can suffer greatly. It can even reach the point where the throughput plummets to only a fraction of the possible throughput that would be achieved if only the best network path would be used to transfer user data.

Since mobile devices often operate in an dissimilar path environment an efficient usage of those paths by CMT is crucial. Moreover, it violates the first of three design goals proposed by [1] which states that a multi-path flow should perform at least as well as a single-path flow on

the best path.

In this bachelor thesis, the challenges of load sharing with CMT-SCTP over dissimilar paths focusing on bandwidth, as described in the previous paragraph, are demonstrated. The cause for these issues is identified and solutions are proposed[2].

To demonstrate and identify this problem a testbed is used in order to have full control over the test environment. The testbed allows to shape traffic on two disjoint paths between two peers. In addition router statistics, network traces and time measurements of internal SCTP routines are utilized to identify the cause.

The analysis of the current implementation shows that the issues are *not* solely CPU related and therefore cannot only be solved algorithmically without altering the SCTP specification.

This thesis focuses on the implementation of SCTP in FreeBSD since it is used as the reference implementation for standardization at the IETF.

[2]The results of this thesis have also been made available on the following GitHub repository: https://github.com/DanSar/ba_sctp

Contents

List of Tables

List of Figures

Listings

1. Introduction

This chapter described the motivation of this thesis, shorty illustrates the challenges that come with CMT-SCTP in a dissimilar path scenario, defines the goals of this work and finally gives a short overview of the thesis' structure.

1.1. Motivation

At the beginning of the Internet age the Transmission Control Protocol (TCP) [2] was standardized at the Internet Engineering Task Force (IETF) [3]. Until this day it has been one of the most important and widely used Tranport Protocols, but the Internet constantly changes which also calls for further development of Transport Layer Protocols.

Especially over the last years the amount of mobile devices with multiple network interfaces, such as smartphones, has increased dramatically. As a result of this development the urge to use *all* available network interfaces has emerged. The simultaneous use of network interfaces allows an increase in application data throughput and improves resilience to network failure.

As it is stated in "Towards the future Internet" by [4], the Stream Control Transmission Protocol (SCTP) [5] could be a suitable successor of TCP, because it overcomes many TCP challenges such as synchronization attacks and it provides a better resilience to network failure.

SCTP is a modern general-purpose, datagram-oriented and reliable Transport Layer Protocol. It supports the use of multiple network interfaces per endpoint, denoted as Multi-Homing. Additionally, the application is informed of message boundaries and can configure ordered or unordered delivery for user messages individually. The extension Concurrent Multipath Transfer (CMT) even allows the simultaneous use of multiple network paths between two endpoints.

However, if the characteristics (i.e. bandwidth, delay and error rate) of these paths change to be more dissimilar the overall throughput can suffer greatly. It can even reach the point where

the throughput plummets to only a fraction of the possible throughput that would be achieved if only the best network path would be used to transfer user data.
Since mobile devices often operate in a dissimilar path environment an efficient usage of those paths by CMT is crucial. Moreover, it violates the first of three design goals proposed by [1] which states that a multi-path flow should perform at least as well as a single-path flow on the best path.

Figure 1.1 illustrates the issue of reduced throughput in a dissimilar path scenario as it was described in the previous paragraph. The measurement that is depicted in green shows the application payload throughput in different dissimilar scenarios. The measurement is performed in a testbed environment with two endpoint that each have two network interfaces. The endpoints can communicate with each other via two disjoint paths: The Northern Path and the Southern Path. The Northern Path is fixed at 10 Mbit/s and the Southern Path varies.

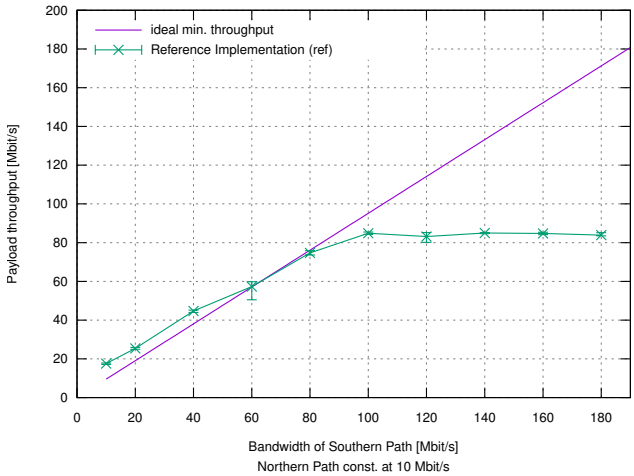

Figure 1.1.: Throughput measurement of CMT-SCTP over dissimilar paths

The minimum throughput that is demanded by the first design rule in a multi-path flow scenario is depicted in purple. It can be observed that CMT-SCTP only safely manages to achieve this goal if the Southern Path is set to less than 60 Mbit/s.

The results presented in Figure 1.1 will be used in the following chapters to identify the issues that lead to the reduced throughput and proposes based on these finding possible solutions.

1.2. Goals

In this bachelor thesis, the challenges of load sharing with CMT-SCTP over dissimilar paths – focusing on bandwidth – are demonstrated. Furthermore, challenges that come with CMT-SCTP in a dissimilar path scenario are identified and solutions that might overcome the described issues are provided, if possible.

1.3. Organization

The thesis is organized as follows:
Chapter 2 – Related Work – provides a short overview of the scientific work related to the development of CMT-SCTP.

Chapter 3 – Basics – provides an overview of the Stream Control Transmission Protocol (SCTP) by comparing it to its relative TCP. Furthermore, the mechanisms used for reliable data transfer are discussed in more detail.

Chapter 4 – Test Environment – describes the test environment and procedures that are applied to the measurements performed in this thesis. Furthermore, network measurement and analysis tools are presented that were used to help identify the issues.

Chapter 5 – Problem of Efficiency – discusses the challenges of the SACK handling algorithm with a focus on efficiency issues. An approach is proposed and evaluated.

Chapter 6 – Problem of Protocol – discusses the challenges of CMT-SCTP in a dissimilar path scenario focusing on protocol related issues. Ideas for approaches are presented and evaluated.

Chapter 7 – Conclusion – summarizes the findings of this thesis and provides an outlook for future work.

2. Related Work

The Internet Draft [6] is a collection of scientific work and papers that are closely related to the development of load sharing for the Stream Control Transmission Protocol (SCTP). The name of SCTP's load sharing extension is also denoted as Concurrent Multipath Transfer for SCTP (CMT-SCTP). [7] first introduced the Concurrent Multipath Transfer (CMT) extension.

The CMT-SCTP approach seems to be rather straightforward at first, because SCTP already offers services like Multi-Homing. With Multi-Homing multiple network interfaces can be used in an association.

However, load sharing with CMT-SCTP leads to some non-trivial challenges. First scientific work focused on CMT-SCTP in similar path environments, e.g. see [7]. Though it became soon apparent that further issues arise in dissimilar environments, e.g. see [8] and [9].

These challenges will be further discusses in the chapters 3, 5 and 6.

3. Basics

This chapter provides a brief discussion of the Stream Control Transmission Protocol (SCTP) as it is defined in [5]. In the following section 3.1 SCTP is compared it to its well-known relative the Transmission Control Protocol (TCP) which is defined by [2]. Furthermore, the basic data transfer mechanism – that is used between two communicating endpoints – as well as some important protocol mechanisms are described.

After the brief discussion of SCTP the section 3.2 introduces the Concurrent Multipath Transfer for SCTP (CMT-SCTP) extension as it is proposed by [6]. As the name implies this extension enables SCTP to concurrently use all available network paths for data transmission.

3.1. Stream Control Transmission Protocol (SCTP)

The Stream Control Transmission Protocol (SCTP) [5] is like TCP [2], UDP [10] and DCCP [11] a member of the Transport Layer Protocols (see [12], section 1.1.3.). The figure 3.1 shows the SCTP Protocol in the IP Reference Model as proposed by [12].

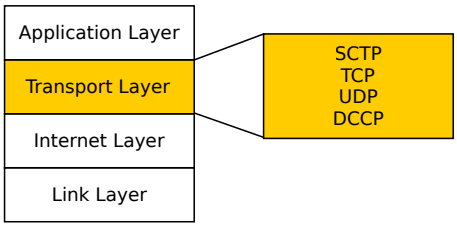

Figure 3.1.: SCTP in the IP Reference Model

Comparison to TCP

Transport protocols provide an end-to-end data transfer and the ability to address an application via port numbers, enhanced by Flow and Congestion Control mechanisms.

Congestion Control mechanisms protect the network from network congestion. If no Congestion Control mechanisms are applied, network congestion can lead to a Congestion Collapse. Congestion Collapse limits or even prevents any useful throughput [13]. A Flow Control mechanism protects the receiver's buffer from overflows that can occur if too much data is sent to the receiver [13].

Some common properties of transport protocols are listed in the table 3.1 in order to compare the services of SCTP as defined in [5] and of TCP as defined in [2] and [14].

Property	TCP	SCTP
Connection-Oriented	yes	yes
Message-Oriented	no	yes
Flow-/Congestion Control	yes	yes
Reliable Transport	yes	config.
Ordered Transfer	yes	config.

Table 3.1.: Comparison between SCTP and TCP

Both protocols are connection-oriented and therefore have to go through a setup phase in order to establish an end-to-end connection between the communicating endpoints. SCTP uses a four-way handshake – and not a three-way handshake like TCP – to protect against synchronization attacks (see [5], section 1.5.1.). Once a connection is established application data can be exchanged. After all application data is transmitted the connection can be gracefully shutdown by a tear-down procedure.

SCTP uses a message-oriented data transportation service. Therefore, it not only transmits the message to the receiving endpoint, but also provides information about the message boundaries. The transmission of message boundaries is also known as conversation of message boundaries (see [15], section 1.2.).

TCP on the other hand will handle all data from the application as a stream of Bytes. Furthermore, TCP ensures that the stream – that was pass down from the application – will reach the receiver's application in the same order of sequence. TCP does not preserve user message boundaries, because TCP only receives a byte stream. Hence, the application has to manage

the message boundaries.

Both protocols have the same goals with the Congestion Control mechanisms. The goals of Congestion Control are to protect the network from congestion – or even congestion collapse – and to ensure fairness between both protocols and communication flows. A communication flow – i.e. flow – is a host-to-host communication path that can be identified by a distinct combination of source and destination address, port numbers and used transport protocol. This combination is also denoted as a 5-tupel (see [16], section 14.1.). If flow fairness is not established network resources can not be distributed equally and the network can become unstable.

Both Congestion Control mechanisms use mainly two algorithms to regulate the amount of data that can be injected into the network. These algorithms are called *slow start* and *congestion avoidance*. The congestion window *cwnd* defines the number of Bytes that can be in flight at any time. The *cwnd* therefore limits the amount of data the sender can inject into the network. Both algorithms increase the *cwnd*, but using different preconditions (for SCTP see [5], section 7.2.). The slow start threshold *ssthresh* is used to determine whether the *slow start* algorithm or the *congestion avoidance* algorithm is used as it is defined in 3.1.

$$cc\_algorithm = \begin{cases} slow\_start, & \text{if } cwnd <= ssthresh \\ congestion\_avoidance, & \text{else} \end{cases} \tag{3.1}$$

The often used underlying Internet Protocol (IP) as defined by [17] only supports best effort delivery. In case of TCP, the protocol must ensure that all application data will be transferred to the receiver's application in ordered sequence and with no duplicates. In contrast to TCP, SCTP is more flexible when it comes to reliable transport. In SCTP the service can be configured as reliable or ordered.

Another advantage of SCTP is the support of Multi-Homing. If an endpoint holds multiple IP addresses, which is normally the case when having multiple network interfaces (NICs), Multi-Homing enables SCTP to use multiple IP addresses in order to increase reliability.

A possible SCTP setup is presented in figure 3.2. The figure shows two endpoints Node A and Node B. Both endpoints have two network interfaces, and thus at least one IP address per

interface.

In this example each endpoint has two TCP-like connections. These TCP-like connections are defined by a 4-tupel that consists of source address, destination address, source port and destination port (see [16], section 14.1.). These TCP-like connections are denoted as communication paths and are unidirectional flow.

From the application's view the individual communication paths are not visible. The communication paths are abstracted by a bidirectional end-to-end connection that is called association. The communication service is therefore handled by SCTP internally. This also includes the scheduling of user messages to the communication paths.

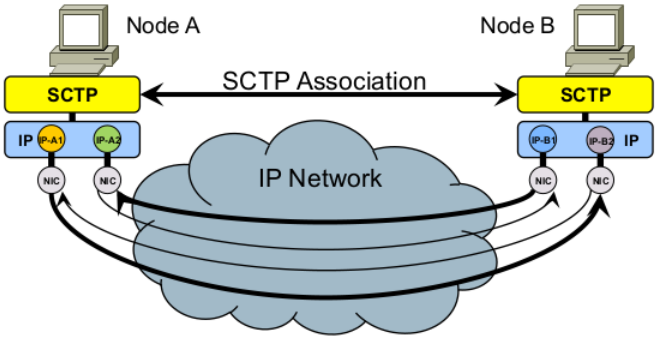

Figure 3.2.: Example of an SCTP Association

The bold paths in figure 3.2 are denoted as primary paths. Only primary paths are used to transmit application data, all other paths are only fallbacks or can be used to retransmit data in case of loss. Each SCTP endpoint has only one primary path per association. If data fails to be transmitted over the primary path a new primary path will be chosen from the remaining communication paths.

Another service of SCTP is the flexible TLV design, as exemplified in figure 3.3. In [15] TLV is described as a data structure format that is always composed of three fields: Type, Length and Value. The figure 3.3 shows an IP packet with an IP header that is depicted in green. Its payload, the SCTP packet consists of a common header, which is depicted in yellow, and a set of chunks. The common header holds information like source and destination port.

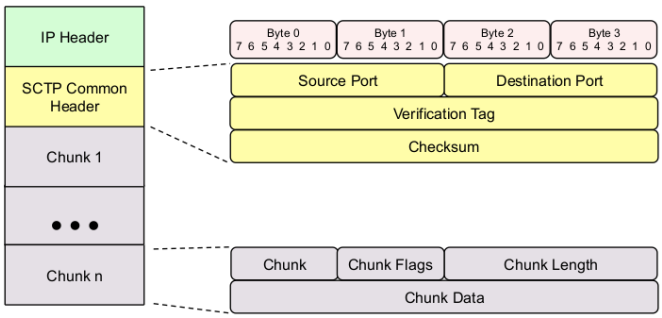

Figure 3.3.: Structure of a SCTP Packet

An SCTP packet can consist of various control and DATA chunks. However, the transmission of chunks that hold control information take precedence over the transmission of DATA chunks.

Each chunk consists of a chunk header and chunk data. The chunk header follows the Type Length Value (TLV) format. A chunk that carries control information is e.g. a selective acknowledgement (SACK chunk or simply SACK). A chunk that carries application data is denoted as a DATA chunk. A DATA chunk has to carry exactly one user message or a fragment of a user message (see [15], ch. 3.2.5.). A chunk can either be comprised of control information or application data.

Data Transfer in SCTP

Once the association between two communicating endpoints is established the normal bidirectional data transfer can start. This section describes the process of sending and receiving user messages and will focus on a reliable transport scenario which is compatible to a TCP scenario.

In order to send application data the sending endpoint needs to encapsulate the user messages in DATA chunks.

If the payload fits in an IP packet (minus the SCTP common and DATA chunk header) only a DATA chunk header is required. The DATA chunk header holds information like the Transmission Sequence Number (TSN). The TSN is very similar to the sequence number used in TCP. The TSN is assigned per DATA chunk (see [5], section 1.3.) and not per Byte like it is done in TCP (see [2], section 3.3.). The payload needs to be fragmented and encapsulated in multiple DATA chunks if it is larger than the allowed payload size of an IP packet (minus the

SCTP common and DATA chunk header).

In order to ensure that all payload has been transferred an acknowledgments is needed. An acknowledgment is represented by a SACK chunk, i.e. SACK.

The SACK chunk holds the following information (compare [5], sec. 3.3.4.), as presented in figure 3.4:

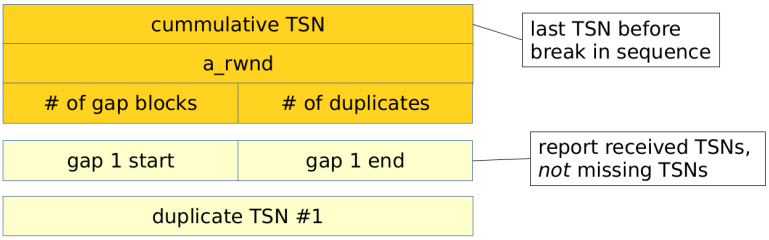

Figure 3.4.: Structure of a SACK Chunk

- Cumulative TSN *cumack*
  The cumulative TSN *cumack* acknowledges all TSNs that were received in order. It is therefore the TSN of the last received DATA chunk that is part of the ordered sequence. The TSNs that are part of the *cumack* can be removed from the send buffer, because the receiver passed these DATA chunks to the application and will not request a retransmission in the future.

- Advertised receiver window *a_rwnd*
  The advertised receiver window *a_rwnd* is included in the SACK to inform the sender about the remaining receive buffer space. The sending endpoint is only allowed to send new DATA chunks if there is enough buffer space left in the receive buffer. This mechanism, denoted as Flow Control, protects the receiver from being flooded by too many DATA chunks. Nevertheless, the sender is allowed to retransmit DATA chunks even if the receive buffer is already used up.

- Gap-Ack Blocks
  DATA chunks that were received out-of-order are also reported in a SACK. Each Gap-Ack Block consists of a start and an end address. The addressing is relative to the *cumack*. Each Gap-Ack Block therefore informs the sender of a range of TSNs that have been received, but are not part of the *cumack*. These TSNs are selectively acknowledged.

In comparison to the *cumack*, these DATA chunks cannot be passed to the application because there are still some DATA chunks missing that need to be received first in order to complete the sequence. In some cases the receiver even needs to request a retransmission of DATA chunks, that were already received once.

The reason for this is as follows: The receive buffer can hold e.g. three DATA chunks. The *cumack* is at TSN #38. The receive buffer holds the TSNs #40 - #42. Once the TSN #39 arrives the receiver needs to drop one of the queued TSNs, e.g. TSN #42, in order to store TSN #39. The receiver wants to store the TSN #39 because this TSN will complete the sequence of TSNs #39 to #41. The receiver can then pass all DATA chunks that are in the receive buffer to the application and send a SACK with an advanced *cumack* and no Gap-Ack Blocks.

The TSN #42 is not selectively acknowledged in the current SACK, because it was dropped from the receive buffer. However, it was selectively acknowledged in a previous SACK and therefore the sending endpoint knows that TSN #42 needs to be retransmitted.

- Duplicates
  This field contains the number of TSNs that have been received more than once. A list of duplicate TSNs is appended after the Gap-Ack Blocks. Duplicate TSNs do not need to be processed by the sending endpoint as described in [5] section 6.2. and they are not as of today.

A Data chunk, i.e. TSN, that has been sent to the receiving endpoint, but has not yet been acknowledged is referred to as an outstanding TSN or one that is still in flight.

An example of a SACK chunk, that needs to be processed by the sending endpoint, is presented in figure 3.5. Further examples and detailed explanations of the SACK handling are provided by [15] in chapter 5.

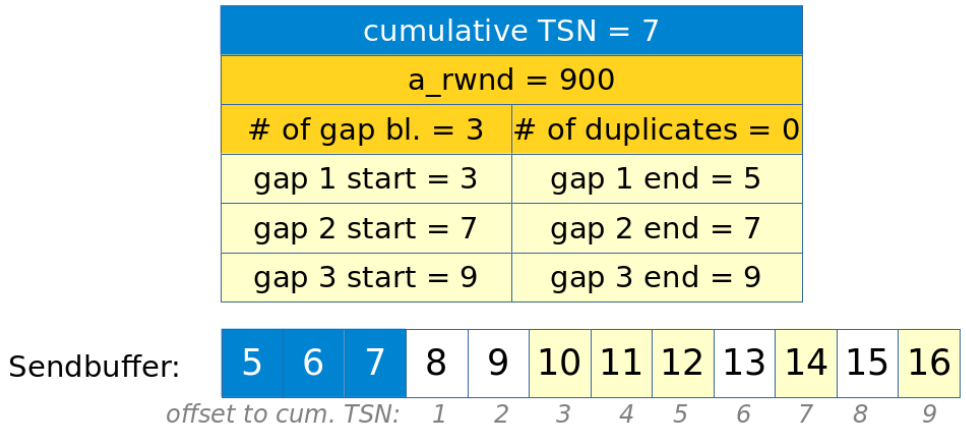

Figure 3.5.: Example of a SACK Chunk

The sending endpoint can obtain the following information from the SACK (fig. 3.5):

- Cumulative TSN *cumack*
  The cumulative TSN *cumack* has moved to TSN #7. The sender can therefore remove the TSN #5 - 7 from the send buffer (highlighted in blue) and make room for new DATA chunks, e.g TSN #17 - 19 could be added to the send buffer.

- Advertised Receiver Window (a_rwnd)
  The *a_rwnd* tells the sender that the receiving endpoint has 900 bytes of free receive buffer space left.

- Gap-Ack Blocks
  The SACK also contains three Gap-Ack Blocks. Through the Gap-Ack Blocks the sender learns that the TSNs #10 - 12, #14 and #16 have been received by the receiving endpoint (highlighted in yellow). Additionally, the sender is indirectly informed that the TSNs #8 - 9, #13 and #14 are still outstanding and might be possibly missing.

If a DATA chnuk got lost in the network, e.g. it was dropped by a router, it needs to be retransmitted. There are two mechanisms available to detect missing DATA chunks:

- Retransmission Timeout
  In SCTP retransmission timers (*T3-rtx*) are used to keep track of the lowest still outstanding TSN. If a RTO (Retransmission Timeout) occurs all outstanding DATA chunks

on that path will be marked for retransmission. A retransmission timer is restarted if the lowest outstanding TSN is cumulatively acknowledged and there are still DATA chunks in flight, otherwise it will be stopped. The slow start threshold *ssthresh* and the congestion window *cwnd* are modified as described in 3.2 (see [5], section 7.2.3.).

$$\begin{aligned} ssthresh &= max(\frac{cwnd}{2}, 4 * MTU) \\ cwnd &= 1 * MTU \end{aligned} \tag{3.2}$$

- Fast Retransmission
  Fast Retransmission is another mechanism to detect loss. Each time a SACK arrives a miss indication counter is incremented for possibly missing TSNs. If a TSN is reported as missing for the third time it is marked for retransmission and the slow start threshold *ssthresh* and congestion window *cwnd* are modified as described in 3.3 (see [5], section 7.2.3.).

$$\begin{aligned} ssthresh &= max(\frac{cwnd}{2}, 4 * MTU) \\ cwnd &= ssthresh \end{aligned} \tag{3.3}$$

In both scenarios DATA chunks will be retransmitted in order to fill the gaps in the sequence. In addition, the congestion control variables *ssthresh* and *cwnd* will be reduced as described in 3.2 and 3.3. In both scenarios the sender will enter the *slow start* phase again, because the *cwnd* <= *ssthresh*.

In case of a Fast Retransmission the sender will only remain in the slow start phase until the next SACK arrives that increases the *cwnd*. After a single increase of the *cwnd* the path will enter the *congestion avoidance* phase again.

An RTO on the other hand will reset the *cwnd* to one MTU, which will cause the affected path to go through the whole *slow start* phase again. An RTO is therefore more severe than a Fast Retransmission.

3.2. Concurrent Multipath Transfer (CMT-SCTP)

Concurrent Multipath Transfer (CMT) is an extension to SCTP that allows to transmit application data concurrently over all available paths in order to improve throughput by making further use of SCTP'S Multi-Homing feature [6]. CMT-SCTP is not yet a standardized extension of SCTP, but the Internet Draft [6] summarizes the work that already exists on CMT-SCTP.

As in section 3.1 described, each SCTP endpoint declares one path as the primary path and only transmits application data over this path. This proceeding has the advantage that congestion control mechanisms can be used that are based on a TCP Congestion Control mechanisms. Furthermore they can be considered TCP-friendly as stated by [18].

The term TCP-friendly describes the goal of a fair competition for bandwidth with other TCP flows [19]. In [19] fairness is described as: "A flow is 'reasonably fair' if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same conditions". Another more strict definition of TCP-friendly is provided by [20]: A TCP-friendly flow is responsive to congestion notification and it uses no more bandwidth than a conforming TCP connection running under comparable conditions.

The simplest implementation of a congestion control algorithm for CMT-SCTP (CMT-CC) is to transfer application data over all available paths and to apply the congestion control algorithm that is used by SCTP, independently on each path. This congestion control algorithm will be denoted as *cmt* in this thesis.

The problem with this approach is that fairness towards other TCP flows cannot be guaranteed in any environment, especially not one like the Internet. The reason for this is due to the possibility of shared bottlenecks. The figure 3.6 illustrates this problem.

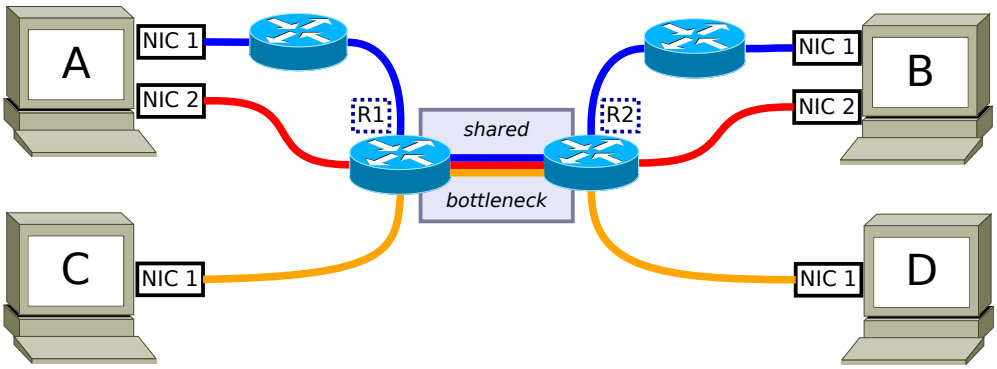

Figure 3.6.: Example of a shared bottleneck

In the example (figure 3.6) the endpoints A and B use CMT-SCTP and the endpoints C and D use TCP. A shared bottleneck exists between the routers R1 and R2. If A and B use the congestion control *cmt*, as described above, they will get about $2/3$ of the available bandwidth where as the TCP flow will only get $1/3$. According to [19]'s definition of TCP-friendly this would still be considered 'reasonably fair', but not if [20]'s definition of TCP-friendliness is applied. This could become even worse if more paths would use the same link between the two routers R1 and R2.

Due to these complications the principle of Resource Pooling was proposed by [21] and a set of design rules has been established in [1] for TCP-friendly CMT Congestion Control Algorithms. The design rules are described in the following:

1. *Improve Throughput*
   A multipath flow should perform at least as well as a single-path flow would on the best of the paths available to it.

   The rule *Improve Throughput* has already been discussed in section 1.1.

2. *Do Not Harm*
   A multipath flow should not take up any more capacity on any one of its paths than if it was a single path flow using only that route. This guarantees that it will not unduly harm other flows.

   The rule *Do Not Harm* attacks the problem of shared bottlenecks as exemplified in figure 3.6.

3. *Balance Congestion*
   A multi-path flow should move as much traffic as possible off its most-congested paths.

   The rule *Balance Congestion* gives advice on packet scheduling with the goal to balance network congestion.

Out-of-order delivery is not very uncommon in a CMT-SCTP scenario. Some user messages may overtake others, because they are e.g. scheduled on a path with a "better" Quality of Service (QoS). Therefore, the gaps in a SACK have not emerged due to loss, but because of varying QoS on the paths.

This leads to the issue that Fast Retransmissions are continuously triggered even though no DATA chunks were lost. Therefore, in CMT-SCTP the Fast Retransmission algorithms only increments the miss indication counter if on the same path has been a higher TSN newly acknowledged (HTNA). This adaption is known as Split Fast Retransmissions and is proposed by [7].

Another issue emerges once DATA chunks are retransmitted. If a DATA chunk is marked for retransmission it is rescheduled on the possibly best path. The newly chosen path may differ from the path the DATA chunk was originally transmitted on. This can become problematic if the TSN range of the newly chosen path is smaller than the retransmitted TSN. Once the retransmitted TSN is selectively acknowledged all DATA chunks with a lower TSN might be marked for retransmission by the Fast Retransmission algorithm. In order to prevent bursts of retransmissions Smart Fast Retransmissions are introduced by [8]. Smart Fast Retransmissions exclude DATA chunks that were scheduled on a different path by the Fast Retransmission algorithm.

The described alterations to the Fast Retransmission algorithm are a fixed part of the CMT-SCTP implementation in the FreeBSD kernel. Other features can be switched on and off via system controls (sysctl). The table 3.2 gives an overview of CMT-SCTP specific system controls that are of interest for this thesis and will be introduced in more detail in chapter 5 and 6.

System Control	Description
net.inet.sctp.cmt_on_off	Enables the CMT extension and selects the CMT-CC
net.inet.sctp.cmt_use_dac	Enabled Delayed Acknowledgments for CMT (DAC) Algorithm
net.inet.sctp.buffer_splitting	Enables Send-/Receive-Buffer Splitting
net.inet.sctp.nrsack_enable	Enables the use of NR-SACKs

Table 3.2.: Important SCTP System Controls

4. Test Environment

In order to evaluate the performance of CMT-SCTP in various dissimilar scenarios a fully-controllable environment is needed. This chapter describes the setup of this environment.

Furthermore, it introduces the network performance and analysis tools, that are used to perform the measurements or analysis of the behavior of CMT-SCTP, such as NetPerfMeter, DTrace and Wireshark.

4.1. Testbed

The testbed presented in figure 4.1 consists of two computers and two traffic shaping software routers (soft-routers), which will be referred to as North, East, South and West in this thesis. Two disjoint paths are configured: The Northern Path (depicted in blue) and the Southern Path (depicted in red).

The computers and soft-routers are FreeBSD-based and use single-board computers of the type PC Engines Alix apu2c4, more details on the computers can be found at [22]. A more specific schematic of the testbed can be found in appendix A.1.

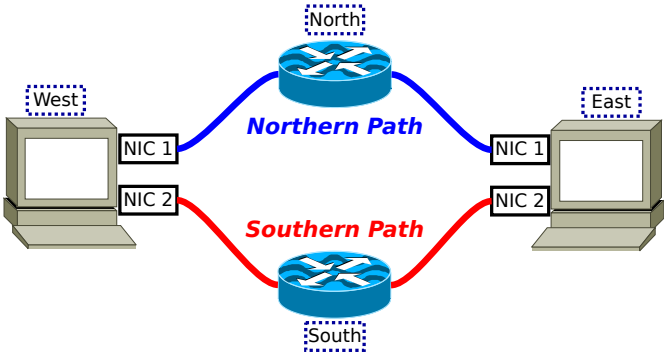

Figure 4.1.: Testbed Setup

The computers East and West act as the communicating endpoints. Both endpoints use the network performance tool NetPerfMeter to generate the traffic and to measure the resulting bandwidth and loss. The NetPerfMeter tool will be described in more detail in subsection 4.2.

The computers North and South act as traffic-shaping routers. They use Dummynet in order to apply certain Quality of Service (QoS) characteristics.

In [23] the IETF's understanding of QoS is described as "a set of service requirements to be met by the network while transporting a flow". Since this thesis focuses on the services provided by Transport Protocols the following QoS characteristics are of interest: Bit rate, delay and packet loss.

Dummynet

Dummynet is an extension to the packet filtering infrastructure IPFW (IP FireWall) [24] of the FreeBSD kernel. It controls traffic that goes through a network interface by applying certain QoS characteristics such as bandwidth and queue size limitations, usage of different queue types, and the emulation of delays and losses [25]. IPFW will be automatically loaded to the kernel, where as Dummynet must be explicitly loaded into the kernel. This can be achieved via the shell command `kldload dummynet` or statically added to the kernel at the time of kernel compilation.

There are mainly two queueing algorithms in dummynet that are of interest for this thesis. One is called Tail Drop as described in [20] and the other is called Random Early Detection (RED) as described in [26].

1. Tail Drop:
   Packets are dropped if they cannot be buffered anymore, i.e. the queue limit (number of slots) is reached – otherwise they are stored in the queue and forwarded when they reach the head of the queue.

2. Random Early Detection (RED):
   Packets are dropped or marked on statistical probabilities. If the routers buffer is almost empty the probability that packets are queued – and not dropped or marked – is very high. As the queue grows, i.e. the buffer fills up, the probability that an incoming packet is dropped or marked rises.

Queue scheduling algorithms are a second part of mechanisms used by routers to process packets. Queue scheduling algorithms are used to manage the allocation of bandwidth among flows. They determine which packet is to be send out next and this is not necessarily the packet that first arrived at the router as described by [20]. Since, they are no multiple flows per path in the measurement scenarios performed in this thesis queue scheduling algorithms do not need to be taken into consideration.

4.2. Analysis Tools

NetPerfMeter

NetPerfMeter [27], i.e. Network Performance Meter, is an open source tool for network performance evaluation for transport protocols such as TCP and SCTP over IPv4 and IPv6. It also supports CMT-SCTP.

NetPerfMeter is one of a few tool-chains that support SCTP and especially the parametrisation of CMT-SCTP. It has already been used in numerous scientific works, like [28] and [29], that deal with the evaluation of CMT-SCTP related topics. Therefore, it is also used as a measuring tool in this thesis.

In order to perform a network measurement with NetPerfMeter one endpoint, e.g. East, has to act as the sink (server mode) and waits for a connection from a source (see listing 4.1).

```
$ netperfmeter <port>
```

Listing 4.1: NetPerfMeter: Sink (Server)

The other endpoint, eg. West, acts as the source (client mode) and specifies the parameters for the measurement.

A simple example would be to transmit data from source to sink using the transport protocol SCTP. In order to measure the links capabilities a saturated flow, ie. send as much data as possible, is used. This can be done by calling the command from listing 4.2.

```
$ netperfmeter <passive side>:<port> \
   -sctp const0:const1452:const0:const0
```

Listing 4.2: NetPerfMeter: Source (Client), Simple Example

In listing 4.2 the source connects to the sink by passing the IP address and port of the server as a first argument to NetPerfMeter. The next argument specifies that the transport protocol SCTP will be used. The flow is further specified:

outgoing_frame_rate:outgoing_frame_size:incoming_frame_size:incoming_frame_rate:option

The outgoing frame rate is set to 0 Frames/s in order to configure the saturated flow and the outgoing frame rate is set to 1452 Bytes/Frame.

In order to change the scenario from listing 4.2 to a CMT-SCTP scenario with the congestion control algorithm *cmt* only the flow specification has to be altered by appending the option `cmt` (see listing 4.3).

```
$ netperfmeter <passive side>:<port> \
   -local=<local addresses> -runtime=300 -scalar=title.sca \
   -sctp const0:const1452:const0:const0:cmt=cmt
```

Listing 4.3: NetPerfMeter: Source (Client)

Besides the alteration to the flow specification, listing 4.3 shows some addition parameters:

- `local`: Specify local IP adresses that should be used.
  This settings is important in order to prevent the use of the external interfaces (igb0), see the more specific schematic of the testbed in appendix A.1.
- `runtime`: Specify the time for the measurement in seconds.
- `scalar`: Specify an output file that store the results of the evaluation, such as total amount of transmitted data, average bandwidth and lost packets.

DTrace

DTrace is short for Dynamic Tracing and is a performance analysis and troubleshooting tool that can not only analyze user-land programs, but also the operating system kernel and device drivers. DTrace is included and ready to use in the FreeBSD Operating System.

DTrace allows to modify the kernel and user processes dynamically to record additional data through so called probes. Like stated in [30]: "A probe is a location or an activity to which DTrace can bind a request to perform a set of actions, like recording a stack trace, a timestamp, or the argument to a function. Probes are like programmable sensors scattered all over the system".

DTrace is used for this thesis in three ways:

1. To get a global overview of the CPU usage at the time of a CMT-SCTP Measurement, e.g. the hotkernel analysis [31]
2. Analysis of a certain kernel level function
3. Protocol-specific analysis

Wireshark

Wireshark [32] is an open source network packet analyzer tool that is used to examine the traffic between the communicating endpoints in order to analyze the behavior of CMT-SCTP.

With Wireshark network dumps can be analyzed that were captured by tools like TCPDump [33]. In order to capture all packets that reach the network interface controller (NIC), and not just the ones that are addressed to one of its own addresses, the NIC needs to support the promiscuous mode.

In order to use a larger variety of features in Wireshark it is desirable that the whole network traffic is combined in one network dump. Wireshark is e.g. able to mark and count retransmissions of a DATA chunk, especially if they were retransmitted over an alternate path. Therefore, port mirroring is used to be able to record the whole traffic from the Northern and Southern Path in a single network dump.

Port mirroring is a mechanism that allows to record all packets for a given list of source ports. Therefore, one port of the switch is configured as a mirror port (sometimes also referred to as monitor port). Further information can be found at [34], section 4-5. In this thesis the mirror port records all packets that are send or received from the endpoint West on both paths. Detailed port mapping information are available in appendix A.1. The monitor system can then connect to the configured mirror port and capture the arriving traffic with TCPDump.

Furthermore, it is profitable that the network dumps are already on the target PC, because the size of the network dumps grows rather fast and can easily reach a couple GiB during a test scenario.

However, the ability of Port Mirroring is limited if the amount of traffic that is directed to the mirror port is greater than the mirror port ability to process it. If e.g. a Gigabit-Interface is configured to be a mirror port it can only process Gigabit traffic. If more traffic arrives only part of it can be recorded. All measurements performed in this thesis are far from reaching this border, hence port mirroring can be used without deficits.

4.3. Measurement Procedure

All measurements that will be presented in this thesis comply with the following measurement procedure:

- The measurement runtime for each measurement run is set to 300 seconds.
  Numerous measurements have shown that 300 seconds are a sufficient time span so that the resulting average throughput is not affected by any start-up time. Start-up time in this context refers to the time span SCTP needs until it steadily reaches the maximum throughput.
- A total of five measurement runs were conducted for one measurement point.
  This should ensure that any outliers are captured and the amount of scatter per measurement point can be identified.
- The result of one measurement point is presented in the figures as follows:
  - The median is calculated out of the five measurement runs.
  - The lowest and highest measurement result is also displayed as an vertical error line.

- If a chart contains a line without points, the line resembles ideal calculated values.
- If a chart contains a line with data points, the line is the result of a measurement.

In order to calculate an ideal theoretical payload throughput, known sizes of payload and overhead can be used:

- The size of the Ethernet frame is 1526 Bytes.
  The standard MTU for Ethernet is 1500 Bytes as defined in [35]. Adding to the MTU the preamble (8 Bytes), the source and destination MAC address (2x6 Bytes), the ethertype (2 Bytes) and the CRC (4 Bytes) results in 1526 Bytes per Ethernet packet.
- Each SCTP packet has a payload of 1452 Bytes [1].
  A payload size of 1542 Bytes results from the MTU of 1500 Bytes minus the IP header (20 Bytes) and SCTP common and data chunk headers (28 Bytes).

The ideal, i.e. theoretically possible, payload throughput for a multi-path scenario can be calculated with the equation 4.1.

$$\sum_{i=1}^{num\_paths} bandwidth_i * protocol\_efficiency \tag{4.1}$$

The protocol efficiency can be calculated with the equation 4.2.

$$protocol\_efficiency = \frac{payload\_size}{frame\_size} = \frac{1452B}{1526B} \approx 95.15\% \tag{4.2}$$

[1] Only if a single user message, i.e. DATA chunk, is included in the SCTP packet.

The following list describes the default configuration for measurement runs:

- The bandwidth of the Northern Path is fixed at 10 Mbit/s. The bandwidth of the Southern Path is variable.
- Both endpoint use the standard send- and receive buffer spaces that are set by default in FreeBSD to 1864135 Bytes.
- The delay of both paths is set to 1 ms.
- No Packet Loss Rate (PLR) is applied.
- The Tail Drop queuing algorithm is used. The queue is configured to have 100 slots, which is the maximum setting.
- The size of a user message is set to 1452 Bytes, as described earlier.
- Only the system control net.inet.sctp.cmt_on_off is configured to use the congestion control *cmt*.
- All traffic sent is ordered and reliable.

If it is not stated otherwise the default configuration for measurement runs is applied.

The measurement configuration described above was chosen to reduce the affects created by other factors – like small router queues – and rather focus on the dissimilarity of bandwidth and its effects first.

5. Problem of Efficiency

The reduced throughout of CMT-SCTP in a dissimilar scenarios – see figure 1.1 – lead to the question if the reduced throughput can be related to the efficiency of the implementation.

Therefore the following discussion focuses on efficiency problems in a dissimilar scenario that could lead to a reduction in throughput. An approach is presented and evaluated.

5.1. Analysis

In order to determine if the measurement results from figure 1.1 are caused by an efficiency problem the following steps are taken:

1. Repetition of the Measurement on different Hardware
2. Analysis of Network Traffic with Wireshark
3. Analysis of SCTP Kernel Functions with DTrace
4. Analysis of SCTP Kernel Source Code

Each step is discussed in detail the section 5.1.1 to 5.1.4.

5.1.1. Repetition of the Measurement on different Hardware

During the measurement series for figure 1.1 an increasing CPU load could be observed once the Northern and Southern Paths became more dissimilar.

Therefore, a second series of measurements is done on another testbed environment that follows the exact same setup as described in chapter 4, but uses different hardware for the endpoints. The endpoints from the primary testbed use a AMD Embedded G series GX-412TC processor with a CPU frequency of 1 GHz [22]. The endpoints from the secondary testbed on the other hand use a Intel quad core i5-4690 Processor with a CPU frequency of 3.5 GHz [36].

The results of the second measurement series are shown in figure 5.1, too.

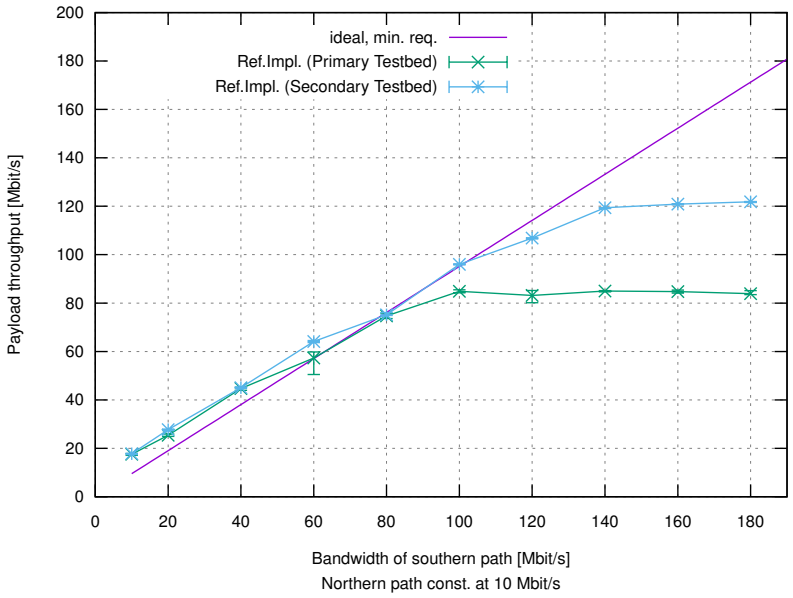

Figure 5.1.: Comparison of two testbeds with different hardware

The measurements performed for figure 5.1 and figure 1.1 used the default measurement configuration like it is described in section 4.3.

The endpoints from secondary testbed show a higher payload throughput than the endpoints from the primary testbed. Since the only alteration to the testbed environment is different hardware for the endpoints, the measurement results suggest that the reduced throughput could be connected to a CPU limitation.

Therefore, it can be concluded that a more efficient implementation that reduces the computation time will also increase the throughput on low performance hardware.

5.1.2. Analysis of Network Traffic with Wireshark

Network traces for two scenarios are recorded: One in a scenario where both paths are configured to 50 Mbit/s and the other were the Northern Path is set to 10 Mbit/s and the Southern

Path is set to 100 Mbit/s.

The results show that the number of selectively acknowledged TSNs greatly increases in a dissimilar scenario. Figure 5.2 shows a histogram of selectively acknowledged TSNs.

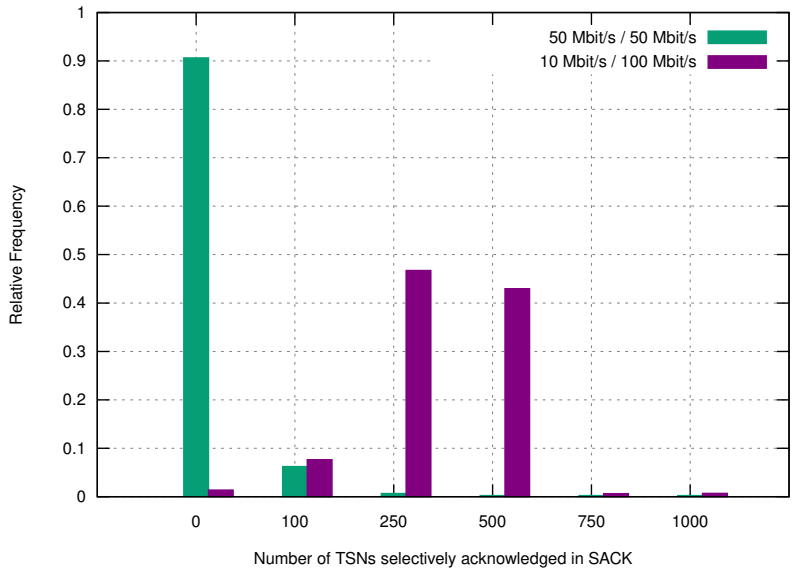

Figure 5.2.: Histogram of the number of selectively acknowledged TSNs

Wireshark triggers a warning every time the number of selectively acknowledged TSNs is greater than 100. In the scenario with similar paths 90 % of SACKs reported less that 100 selectively acknowledged TSNs. In the dissimilar scenario only about 1.5 % did not trigger this warning. Hence, about 98.5 % of SACKs reported had more than 100 TSNs in the Gap-Ack Blocks.

Furthermore, the figure 5.2 shows that close to 90 % of the SACK chunks selectively acknowledged between 251 and 750 TSNs. Thus, about 90 % of SACKs exceed the trigger level of the Wireshark warning by at least 2.5 times.

The network traces also revealed that the advertised receiver window *a_rwnd* never shrank close to zero Bytes. Therefore, a receiver buffer shortage on the receiver's side did *not* forbid

the sender from sending new data chunks.

In summary, the growing number of selectively acknowledged TSNs can be identified as a reason for the reduced throughput, once the dissimilarity between the Northern and Southern Path grows.

5.1.3. Analysis of SCTP Kernel Functions

The findings from the previous subsections indicate that the reduced throughput originates from a great increase in the number of Gap-Ack Blocks. And thus also increases the computation time of processing a SACK chunk on the sender's side.

In this subsection the performance analysis and troubleshooting framework DTrace is used to analyze the kernel routines, using the same test setup as described in subsection 5.1.2.

Hotkernel Analysis

The hotkernel analysis samples over a specified time period kernel-level functions and identifies the amount of CPU time each function used [31]. The so-called hottest function uses the most CPU time during the time span of the measurement.

The figure 5.3 shows the three hottest kernel-functions in two test scenarios. Idle routines were not included in the column chart.

In order to get a reference for a test scenario with dissimilar paths (depicted in purple) the analysis was first done in a CMT-SCTP scenario with similar paths (depicted in green).

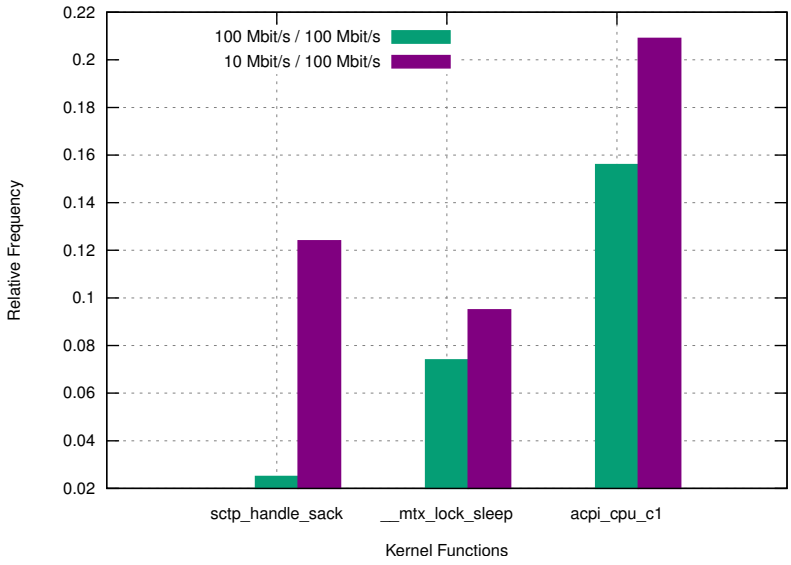

Figure 5.3.: DTrace hotkernel analysis

In both scenarios the kernel-level function *sctp_handle_sack* appears among the top three (hottest) kernel-level functions.

In the scenario with similar paths (depicted in green) the kernel-level function *sctp_handle_sack* used about 2.5 % of the CPU time during the test period. In the scenario with dissimilar paths (depicted in pruple) the CPU time used by *sctp_handle_sack* increased from 2.5 % to about 12.5 %, which means that the function occupied the CPU five times more than in the scenario with similar paths.

The hotkernel analysis reveals that a cause for the reduced throughput in a scenario with dissimilar paths seems to be strongly connected to the kernel-level function *sctp_handle_sack*.

Kernel-level Function *sctp_handle_sack*

Figure 5.4 displays a histogram of the execution time of the kernel-level function *sctp_handle_sack* in a scenario with similar paths (green columns) and one with dissimilar paths (purple columns).

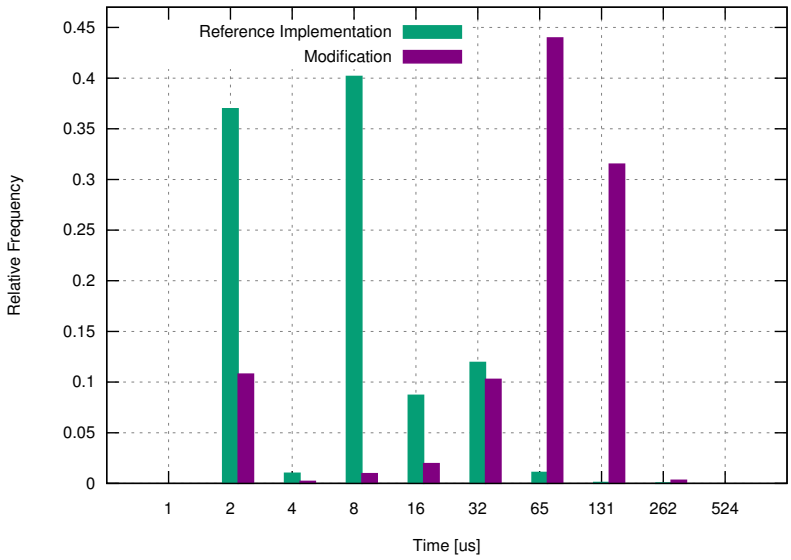

Figure 5.4.: Histogram of the time spent in the kernel-level fuction *sctp_handle_sack*

As expected from the results of the DTrace hotkernel analysis the average execution time of the function *sctp_handle_sack* in a scenario with similar paths is less than the execution time of the same function in a scenario with dissimilar paths.

More interestingly, about 80 % of the function's execution time in a scenario with similar paths is greater 1 µs and less than 8 µs. The same clustering of about 80 % is also present in the scenario with dissimilar paths, but with an execution time greater 32 µs and less than 131 µs.

This type of clustering suggests that the distribution of the execution time of the function *sctp_handle_sack* simply shifts upwards. As expected the function *sctp_handle_sack* is responsible for processing the incoming SACKs. Therefore, it further supports the thesis that the increase in the number of TSNs that are reported in the Gap-Ack Blocks are responsible for the reduced throughput, due to an increase in computation time.

5.1.4. Analysis of SCTP Kernel Source Code

In this subsection the kernel-level function *sctp_handle_sack* is examined in more detail and as a result the Flow Chart (Figure 5.5) was created.

The Flow Chart (Figure 5.5) gives a brief overview of the function *sctp_handle_sack* that resides in the module `netinet/sctp_indata`.

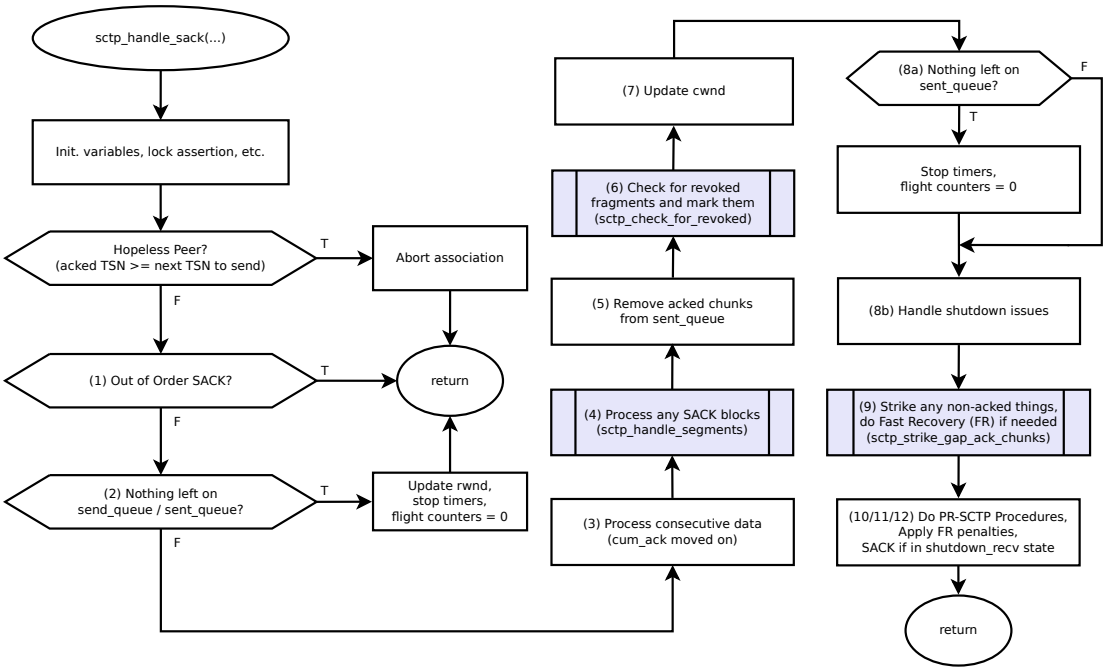

Figure 5.5.: Flow Chart of the kernel-level function *sctp_handle_sack*

Each time a SACK with Gap-Ack Blocks is processed the routines 3 to 6 and 9 are called. The *sent_queue* is a buffer that holds DATA chunks that are sent to the receiving endpoint. DATA chunks can either be removed from the *sent_queue* if they are part of the cumulative acknowledgement or if they were NR-selectively acknowledged[1]. NR-selectively acknowledged TSNs are also part of the Gap Report and are introduced in more detail in chapter 6.

[1]PR-SCTP DATA chunks are out of focus in this thesis.

In the case of a dissimilar scenario the following static functions (colored process blocks) loop through most of the elements from the *sent_queue* that were sent to the receiving endpoint:

- *sctp_handle_segments* (4),
- *sctp_check_for_revoked* (6) and
- *sctp_strike_gap_ack_blocks* (9)

All three functions loop through the *sent_queue* starting with the first element and continue until they reach their stop criterion.

The function *sctp_handle_segments* processes the NR-/Gap-Ack Blocks. It marks DATA chunks from the NR-/Gap-Ack Blocks as newly selectively acknowledged, decreases the flightsize counters and updates the congestion window if necessary.

The function *sctp_handle_segments* iterates through the send_queue until it has processed the last NR-/Gap-Ack Block from the arrived SACK. For performance reasons it is very crucial that the Gap Report should be sorted by the TSN in ascending order. If the gap report is not sorted as described the function *sctp_handle_segments* will take even longer to process all Gap-Ack Blocks, especially in a scenario of dissimilar paths. Every time the algorithm detects a Gap-Ack Block that is not in-order it has to start the loop from the beginning again. The worst performance can be observed if the Gap Report is sorted by TSN in descending order.

In a scenario of dissimilar paths the Gap Report can easily reach a couple of hundred TSNs as the analysis of the network traces showed. With an increased amount of TSNs that are reported in the Gap-Ack Blocks the function *sctp_handle_segments* has to iterate over more elements from the *sent_queue* and thus needs more time for computation.

If a DATA chunk is not selectively acknowledged by the current SACK, but was selectively acknowledged in a previous one, it is revoked by the receiving endpoint. The DATA chunk was removed from the receiver's buffer. A DATA chunk that is revoked needs to be resend to the receiving endpoint. The function *sctp_check_for_revoked* iterates over the *sent_queue* and marks every chunk for retransmission that was not marked by the function *sctp_handle_segments* as newly acknowledged.

To mark all suitable chunks the function has to iterate over all elements in the *sent_queue* as long as the element's TSN is smaller than the highest TSN newly acknowledged (HTNA). All TSN smaller the HTNA are possible candidates for revoked DATA chunks.

In a dissimilar scenario the cumulative acknowledgement advances only slowly. Thus, the function *sctp_check_for_revoked* needs more time to mark all revoked data chunks, because more DATA chunks are queued in the *sent_queue*.

If a SACK with a Gap Report is received a counter is incremented for the TSNs that are possibly missing in order to determine if a DATA chunk needs to be retransmitted (Fast Retransmission algorithm). Possibly missing DATA chunks are those that are not part of the SACK and are prior to the higest TSN newly acknowledged. The function *sctp_strike_gap_ack_blocks* increments the miss indication counter accordingly, marks the DATA chunks with three miss indications for retransmission and also triggers Fast Recovery.

As described in the previous paragraph the stop criterion is the higest TSN newly acknowledged (HTNA).

With an increased number of Gap-Ack Blocks, the function *sctp_strike_gap_ack_blocks* also needs an increased time for computation, because it has to iterate over the *sent_queue* until it reaches the HTNA.

5.2. Approach

The SACK handling procedure described in the previous section (sec. 5.1.4) is rather ineffective, because all three functions iterate over most of the sent DATA chunks from the *sent_queue*, which is not necessary.

The functions *sctp_check_for_revoked* and *sctp_strike_gap_ack_blocks* are only interested in DATA chunks, i.e. TSNs, that were not processed by the function *sctp_handle_segments*.

The DATA chunk's sent status tracks if the DATA chunk was acknowledged, newly acknowledged, revoked or is scheduled for resend. The most important sent status start with «SCTP_DATAGRAM_», this prefix will be omitted in this thesis for simplicity reasons. The table 5.1 gives an overview of the most important sent status. It is important to notice that a sent state that equals SENT can be incremented three times to equal RESEND. This mechanism is used by the function *sctp_strike_gap_ack_blocks* to mark DATA chunks for retransmission.

Macro	Value	Description
UNSENT	0	Data chunk has not been sent yet
SENT	1	Data chunk has been sent, but not yet acknowledged
RESEND1	2	Used as a miss indication counter for Fast Retransmission
RESEND2	3	Used as a miss indication counter for Fast Retransmission
RESEND	4	Data chunk is scheduled for retransmission
ACKED	10010	Chunk was acknowledged
MARKED	20010	Chunk was newly acknowledged
NR_ACKED	40010	Chunk was nr-acknowledged

Table 5.1.: SCTP Datagram Sent States

The function *sctp_handle_segments* marks TSNs of the Gap Report by setting the sent status to MARKED.

The function *sctp_check_for_revoked* changes all DATA chunks that are in the state MARKED to the state ACKED. This switch is necessary in order to distinguish between newly acknowledged TSNs and TSNs that were previously acknowledged but are not anymore, because they are revoked.

TSNs who's sent status equals ACKED are not part of the current SACK, because they were not touched by the function *sctp_handle_segments*. Hence, they are revoked by the receiving endpoint.

The described procedure of the function *sctp_check_for_revoked* is illustrated in the Flow Chart 5.6. For a more detailed Flow Chart see appendix A.3.

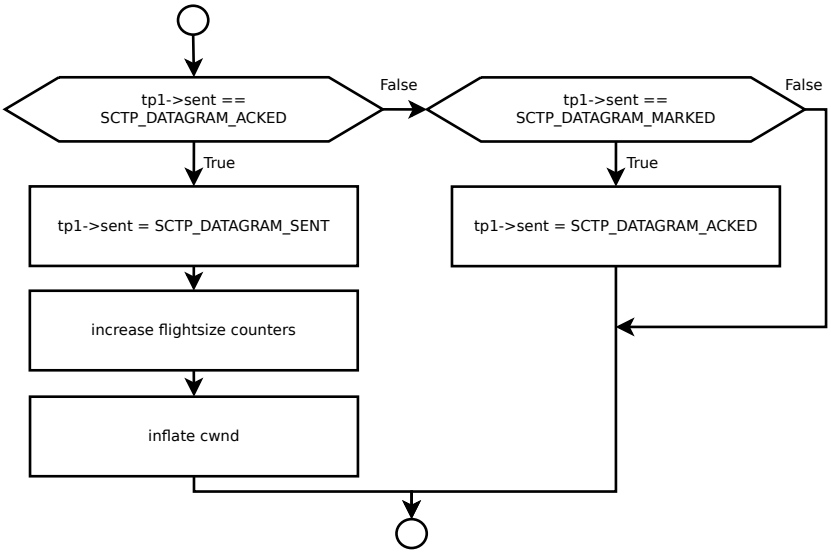

Figure 5.6.: Extract Flow Chart of the function *sctp_check_for_revoked*

The variable `tp1` is a pointer to an element from the *sent_queue*. If the sent status *sent* is equal to ACKED the data chunk was revoked and needs to be sent again.

As illustrated in figure 5.6 the function *sctp_check_for_revoked* really only modifies DATA chunks that are marked ACKED. Therefore, in order to limit the number of elements an array of pointers to DATA chunks is used. This array consists of TSNs that have not been modified by the function *sctp_handle_segments* and are lower than the highest TSN newly acknowledged. Since the function *sctp_handle_segments* iterates over all DATA chunks less than the HTNA it can build up the array.

The function *sctp_strike_gap_ack_blocks* not only processes TSNs who's sent status is equal to ACKED. The code listing 5.1 shows an extract of the functions code. The code extract is located inside the loop that iterates over the *sent_queue*. For a simplified Flow Chart of the function *sctp_strike_gap_ack_blocks* see appendix A.4.

```
1         if (tp1->sent >= SCTP_DATAGRAM_RESEND) {
2             /* either a RESEND, ACKED, or MARKED */
3             /* skip */
4             if (tp1->sent == SCTP_FORWARD_TSN_SKIP) {
5                 /* Continue strikin FWD-TSN chunks */
6                 tp1->rec.data.fwd_tsn_cnt++;
7             }
8             continue;
9         }
```

Listing 5.1: Skipped TSNs from *sent_queue* in *sctp_strike_gap_ack_blocks*

The algorithm will not process a given element and continue with the next element of the *sent_queue* if the sent status is greater or equal to RESEND. Therefore only elements are further processed that have not been acknowledged or are already scheduled for retransmission.

So the function *sctp_strike_gap_ack_blocks* can also make use of the same array of pointers that is prepared by the function *sctp_handle_segments* if it is extended by the data chunks that are equal the sent status SCTP_FORWARD_TSN_SKIP.

In order to keep the evaluation process of the approach simple and flexible a system control is created. The system control *net.inet.sctp.cmt_eff* allows to switch between the reference and efficiency modified implementation. If *net.inet.sctp.cmt_eff* is set to zero the reference implementation is used otherwise the modified implementation is chosen. The use of a system control switch allows to automate measurements. The boolean expressions that have to be evaluated – each time the function *sctp_handle_sack* is called – to decide which implementation to use do not affect the performance imperceptibly.

The array *outstanding_tsn_arr* is used as a data structure to store the outstanding TSNs that might be processed by the functions *sctp_check_for_revoked* and *sctp_strike_gap_ack_blocks*. At initialization of the SCTP association structure an initial amount of memory is allocated for the array. As the number of outstanding TSNs grows the size of the array might also have to be increased. In order to limit the maximum size of the array the system control *net.inet.sctp.outstanding_arr_max_size* is specified. If the number of outstanding TSNs is greater than the maximum specified array size only the first TSNs that still fit in the array will be store. Measurements have shown that a limitation of the array's size does not reduce the throughput if it is not too restricting. The following parameters proved suitable for the measurement scope in this thesis: The initial array size is set to 256 TSNs and the default

maximum array size to 1024 TSNs. The additional memory space that is needed per association amounts to a maximum of $1024 * 8Byte$, because each pointer to a DATA chunk is 8 Byte on a 64-Bit architecture.

5.3. Evaluation

This section presents the results of the evaluation of the suggested alterations that were applied to the implementation of the SACK handling as described in the previous section (sec. 5.2).

The figure 5.7 presents the results of the measurement comparing the modified implementation (mod_eff) to the reference implementation (ref). The measurement used the same test parameter as described in section 4.3.

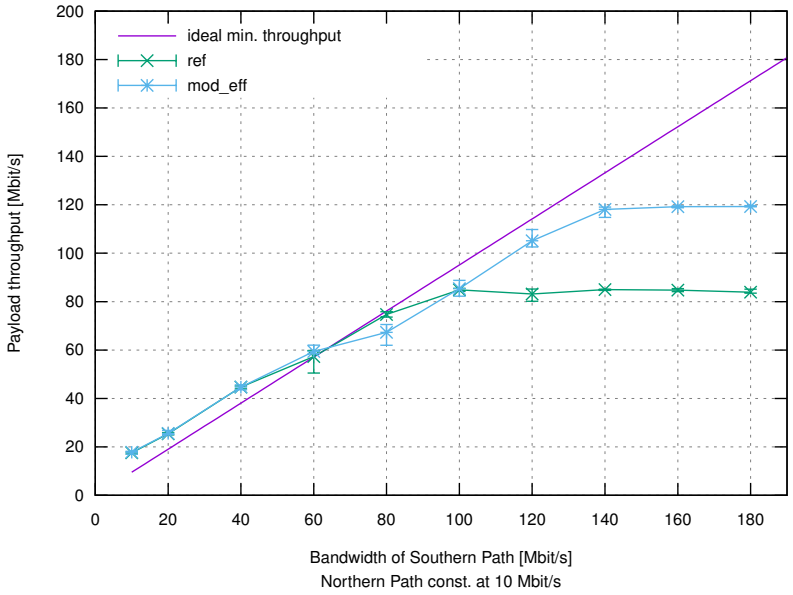

Figure 5.7.: Throughput measurement comparing the reference implementation with efficiency modified implementation

The measurement results show that the modified implementation increases the payload throughput by up to 29 % in comparison to the reference implementation. However, the modified implementation performs worse than the reference implementation at around 80 Mbit/s. The

reason for this behavior will be discussed in subsection 6.1.2. With an increasing degree of similarity of the Northern and Southern Path the performance of the two implementation become nearly the same.

In Figure 5.8 the modified implementation and the reference implementation are compared by their average computation time using the DTrace hotkernel analysis.

The average computation time for both implementation are obtained in a scenario where both paths are configured to 50 Mbit/s (green columns) and another were the Nothern Path is set to 10 Mbit/s and the Southern Path is set to 140 Mbit/s (purple colums).

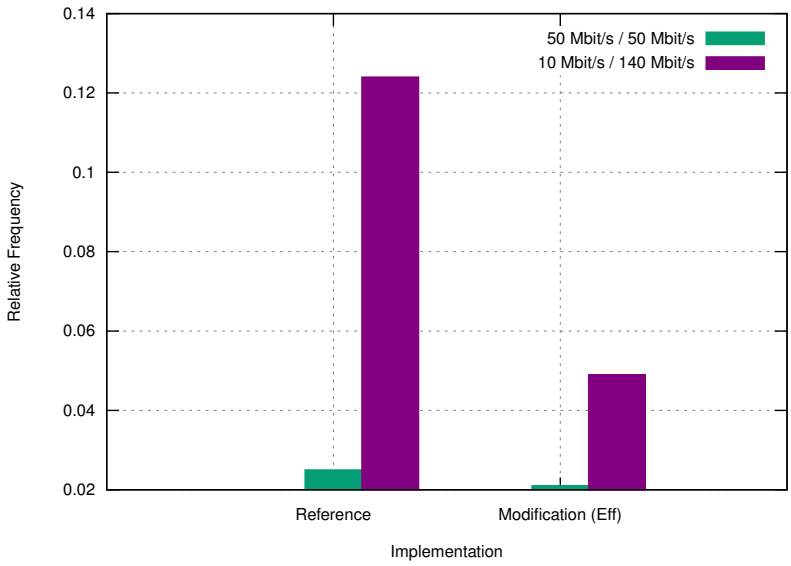

Figure 5.8.: CPU Time of sctp_handle_sack

Through the improvements of efficiency the CPU Usage of the function *sctp_hanlde_sack* could be reduced to only 4.9%, which is about 2.5 times less than the function used in the reference implementation.

Since the alterations proposed in section 5.1 are only efficiency related and do not change the specified behavior, the modified implementation performs as well as the reference implemen-

tation in a scenario of similar paths. In such a scenario the CPU usage time is about 50% less that the reference implementation would need.

The performance of the of modified implementation can be further improved by making use of Buffer Splitting [29] and the DAC algorithm [7].

It is stated in the SCTP specification [5] section 6.2. that the receiver should use the delayed acknowledgement algorithm as proposed by the TCP Congestion Control [37] in section 4.2. The timer for delayed acknowledgments is set for SCTP by the system control variable *net.inet.sctp.delayed_sack_time* to 200 ms. This means that a SACK chunk is send at latest after 200 ms if there is no other reason to send a SACK earlier. But once a gap in the TSN sequence is discovered the receiver should send a SACK chunk immediately and continue to send SACK chunks each time a new DATA chunk arrives until the gap is closed and the cumulative acknowledgement has advanced. This increased rate of SACKs should ensure that Fast Retransmissions are triggered and the gap is closed soon.

In a non-CMT SCTP scenario, where the majority of DATA chunks is only transmitted over a single path such an approach, as described in the previous paragraph, is very crucial because a gap in the TSN sequence can cause the continuous transmissions of user messages to stall. In order to keep the cumulative acknowledgment moving the gaps need to be filled fast.

In a scenario of dissimilar paths this approach can lead to a flood of SACK chunks. The receiver also notices gaps in the TSN sequence, but these gaps are not caused by DATA chunks that actually possibly went missing, but by the way that DATA chunks are scheduled on the communication paths. DATA chunks are scheduled in SCTP by the round-robin scheduling algorithm as described in [38].

To prevent an increased amount of SACKs the Delayed Acknowledgment for CMT (DAC) algorithm ignores the rule that a SACK should be send immediately once a out-of-order DATA chunk is received as proposed by [7].

Another problem that arises in a scenario of dissimilar paths is the problem of Sender and Receiver Buffer Blocking. All DATA chunks who's TSNs are greater than the cumulative TSN have to reside in the send buffer. They can only be removed from the send buffer if they are part of the cumulative acknowledgment. In a scenario of dissimilar paths the send buffer can fill up quickly. This happens because of the way DATA chunks are scheduled on the paths.

The round-robin scheduling algorithm distributed the DATA chunks equally on the available paths which leads to multiple gaps in the TSN sequence if the paths are dissimilar. These gaps can prevent the cumulative acknowledgment to advance at a needed pace that would be necessary to fulfill the demands of the paths to use them to their full capacity. In other words: Not enough DATA chunks can be removed from the send buffer in order to add new DATA chunks that can then be transmitted. The same problem also applies to the receiver side. There it is possible that arriving DATA chunks cannot be stored in the receive buffer and would have to be dropped, because the receive buffer is already filled up by DATA chunks.

A solution for this problem is proposed by [29] and is called Buffer Splitting. Buffer Splitting is a mechanism that tries to provide a metric for per-path usage of send buffer space that should prevent one path from using too much buffer space and therefore limits other paths from sending out new DATA chunks.

An improvement by the use of Buffer Splitting and the DAC algorithm can especially be observed when the Southern Path is configured in between 80 and 120 Mbit/s. Once the dissimilarities of the paths further grow the positive effect of these two mechanism decreases rapidly.

The Figure 5.9 displays the effect of Buffer Splitting and the DAC algorithm on the two implementations.

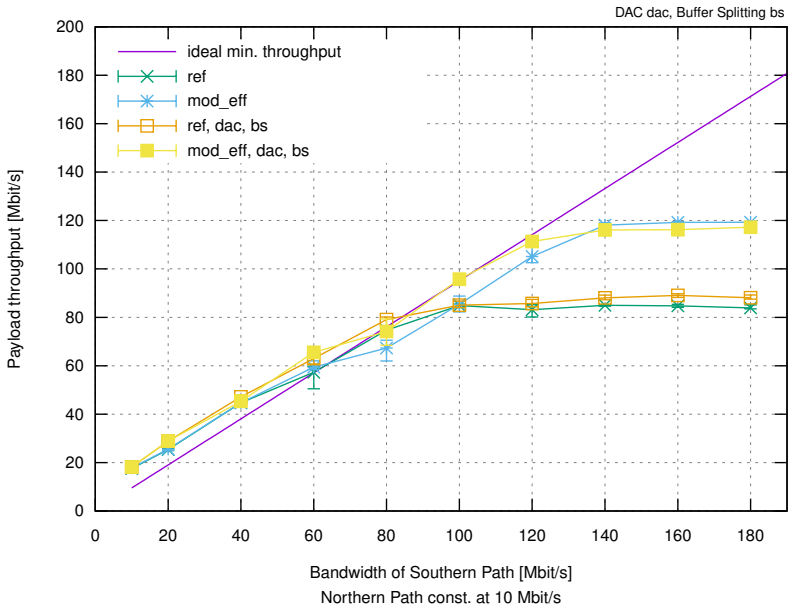

Figure 5.9.: Throughput measurement using DAC and Buffer Splitting

The results presented in the previous Figure 5.9 show that the modified implementation increases the payload throughput and can fulfill the requirements of the first design rule *Improve Throughput* up to a dissimilarity of 10 to almost 120 Mbit/s. But the desired theoretical payload throughput for a CMT scenario is not reached.

The measurements performed so far only looked at a dissimilarity concerning bandwidth. The delay on both paths was set to 1 ms and the packet loss rate (PLR) was set to zero.

In an environment like the Internet such a delay and packet loss rate is rather unlikely. The results of a measurement would be more realistic, if a certain amount of delay and packet loss would be applied to the paths.

If one or even both of these parameters are increased the payload throughput in a dissimilar scenario will decrease even more. In such a case the requirement that is demanded by the first design rule *Improve Throughput* cannot be met anymore.

In the following figure (figure 5.10) a greater delay of 10ms, 25 ms and 50 ms is evenly applied to both paths.

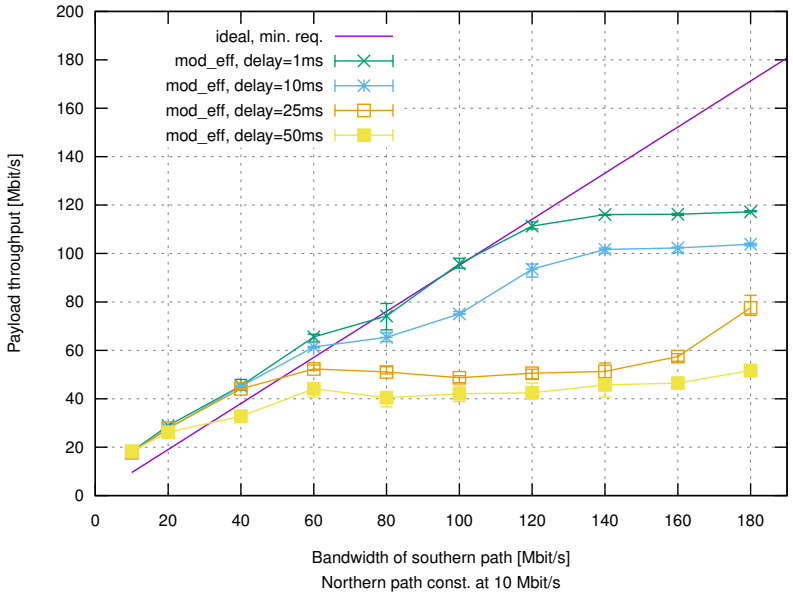

Figure 5.10.: Throughput measurement using DAC and Buffer Splitting varying the delay on both paths

The results presented in figure 5.10 show that if the delay is increased on both paths the payload throughput plummets to the ground.

If the delay on a path is increased the DATA chunks will take a longer time to reach the receiver's side and the acknowledgments will also take longer on the way back. In order to fully utilize the link's capacity the sender needs to inject a certain amount of data into the network. If the sender is not able to do that it has to stop and wait for the corresponding acknowledgment. Thus, the path can not be kept busy and it cannot be used to its full capacity which leads to a reduced overall throughput.

The maximum amount of data that is in the network at any given time, i.e. data that has been injected into the network but has not yet been acknowledged, can be calculated with the bandwidth-delay-product (BDP, see equation 5.1) [39].

$$BDP = bandwidth * RTT \tag{5.1}$$

If the delay increases on a path the BDP will also increase, which means that more data can reside in the network circuit. An example is given in 5.2 and 5.3.

$$BDP_{N.2} = 10Mbit/s * 2ms = 20,000Bit = 2,500Byte \tag{5.2}$$

$$BDP_{N.50} = 10Mbit/s * 50ms = 500,000Bit = 62,500Byte \tag{5.3}$$

The sending endpoint therefore has to inject 25 times more data into the network in order to keep the link busy if the delay is increased from 1 ms to 25 ms. The increased link's capacity becomes even more severe if the send buffer is already used to its capacity. This results in the sending endpoint being unable to satisfy the demands of the link due to send buffer blocking, which then results in a reduced throughput.

The following Figures 5.11 and 5.12 present the impact of packet losses and different queuing algorithms on the resulting payload throughput in a scenario of dissimilar paths.

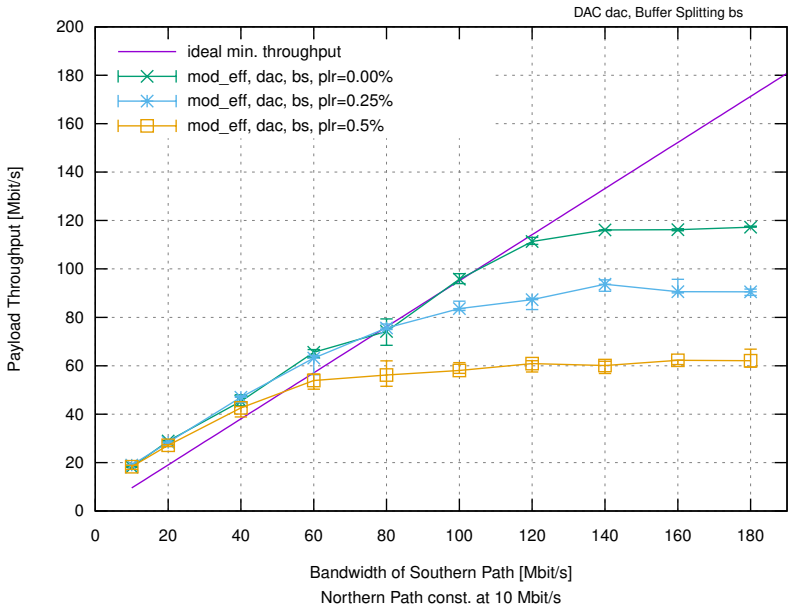

Figure 5.11.: Throughput measurement using DAC and Buffer Splitting varying the PLR on both paths

It can be observed from Figure 5.11 that a packet loss rate (PLR) of 0.25 % has already a noticeable impact on the resulting throughput.

The packets that regularly get lost – or in the test case were dropped by Dummynet – trigger Fast Retransmissions. Fast Retransmissions get triggered because there are higher TSNs that were newly acknowledged (HTNA) and therefore the miss indication counter is incremented for the lost packets. Once a Fast Retransmission is triggered the congestion window *cwnd* is also reduced. In the measurements the slow start threshold *ssthresh* and therefore also the *cwnd* were always set to $cwnd/2$. If packet loss happens constantly the *cwnd* cannot grow linearly until a certain limit is reached but it is constantly reduced by a factor of two. Since packet loss is an indication for the Congestion Control mechanism to detect network congestion the PLR artificially limits the link's capacity.

The use of a different queuing algorithm can increase the overall payload throughput. In contrast to the Tail-Drop approach Active Queue Management (AQM) algorithms, as proposed

by [40], start dropping packets already before the queues capacity is reached. This results in a smaller average queue size and therefore a greater capacity to cope with bursts. In Figure 5.12 the former used Tail Drop queuing algorithm is replaced by the RED queuing algorithm as it is introduced by [26]. The RED queuing algorithm is an AQM algorithm. It can be configured by four parameters: A lower threshold, an upper threshold and two relative parameters that are used to calculate the probability to drop incoming packets. If the average queue size is less than the lower threshold (e.g. 20) the packet is always queued. If it is greater than the upper threshold the packet is always dropped. If it is in between the lower and upper threshold the packet is dropped on calculated probability. The configurations used in figure 5.12 are based on suggestions made in [26].

It can be observed that a suitable configuration can increase the overall payload throughput as it is the case for RED queue #1. On the other hand, a less well configured RED queue, e.g. RED queue #2, can decrease the overall payload throughput.

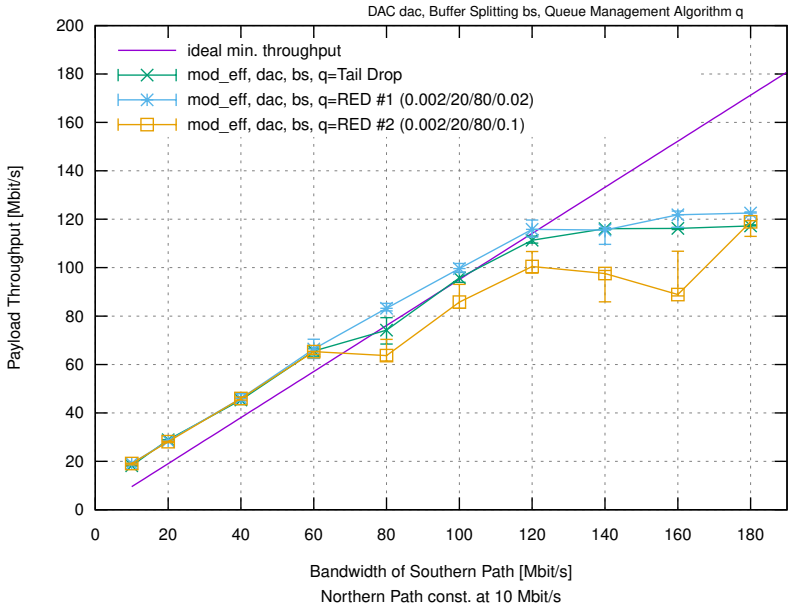

Figure 5.12.: Throughput measurement using DAC and Buffer Splitting varying the queue management algorithm on both paths

The use of AQM algorithms is generally encouraged by [20], but not especially the use of the RED queuing management algorithm as it was in the past. [20] states that the RED queuing management algorithm can be rather efficient with an appropriate configuration, but to predict the suitable set of parameters dynamically has been found difficult. Therefore, [20] recommends that AQM algorithms should be self-tuning and called for further research in this direction in July 2015.

In conclusion: A more efficient implementation alone cannot solve the problem of the reduced throughput in a scenario of dissimilar paths. A more efficient implementation alone cannot fulfill the requirements set by the first design rule *Improve Throughput*, especially not in case of increasing delay and packet losses. A more efficient implementation can only make it less worse.

6. Problem of Protocol

The previous chapter 5 shows that the throughput issues that come with dissimilar paths cannot be solved by a more efficient implementation alone.

Summary of the findings from chapter 5:
It can be observed that the number of Gap-Ack Blocks greatly increases in a dissimilar scenario. The increase of Gap-Ack Blocks is caused by the the DATA chunk's scheduling algorithm and the fact that the paths are dissimilar. The increase of Gap-Ack Blocks leads to an increase in the CPU usage of the kernel-level function *sctp_handle_sack* and thus decreases the overall payload throughput. The SACK handling algorithm was optimized. Though, the increased throughput was not enough to meet the requirements of the first design rule for CMT-CC as proposed by [1].

However, the evaluation of the efficiency optimized implementation already gives some indication towards further reasons for the reduced throughput that can be observed in a dissimilar scenario.

The following sections will therefore focus on protocol related problems that might influence the overall payload throughput.

6.1. Analysis

At first, this section will focus on the countermeasures that have already been taken (see [6], [7], [8], [9], [41], [29]) in order to reduce the problems that arise if CMT-SCTP is used over dissimilar paths. Starting with the DAC algorithm and Buffer Splitting feature which have already been introduced in section 5.3.

6.1.1. Buffer Blocking Issues

The measurement series from Figure 5.9 compares the effect of the features Buffer Splitting and Delayed Acknowledgment for CMT (DAC) to the reference and modified implementation. The results illustrate that the challenges that come with dissimilar paths are not only performance related but also protocol related.

It can be observed that Buffer Blocking issues are already visible once the Southern Path is greater than 60 Mbit/s. From this point on the throughput increases and finally outperforms the reference implementation. However, the requirements of the first design rule *Improve Throughput* cannot be met.

The features Buffer Splitting and DAC further improve throughput – reaching the minimum required throughput up to almost 120 Mbit/s – enabling the modified implementation to barely meet the first design rule's requirements. The measurements further demonstrate that even if Buffer Splitting is enabled it cannot increase the payload throughput to the theoretical possible level if the Southern Path is greater than 60 Mbit/s.

In order to reduce the issues that come with Buffer Blocking the only possible solution is to reduce the number of DATA chunk that are queued in the *sent_queue*. A reduction of queued DATA chunks can be achieved in two ways:

One approach would be to reduce the number of queued DATA chunks by making sure that the cumulative acknowledgment moves more regularly and thus frees up buffer space that can then be used for new DATA chunks. Furthermore, it would reduce the average number of TSNs that are reported in the Gap Report.

An alternative approach would be to use a different scheduling algorithm that minimizes the probability of scheduling-induced gaps in the TSN sequence. DATA chunk scheduling will not be further discussed here, because it is not a focus of this thesis.

6.1.2. Timer-based Retransmissions

Another issue arises when comparing network traces from the measurement series performed in Figure 5.12. Figure 5.12 demonstrates that a properly configured RED queuing algorithm

can increase throughput in comparison to the Tail Drop queuing algorithm.

In case of the Tail Drop measurements simultaneously occurring idle times on all paths can be observed. This phenomenon is not present in case of the properly configured RED queue measurements. The observed simultaneously occurring idle times are connected to timer-based retransmission.

Normally, once a packet gets lost in the network, e.g is dropped by the router, the Fast Retransmission algorithm triggers. It schedules the lost packet for retransmission after it was reported missing for three times. However, if a packet is scheduled for retransmission by the Fast Retransmission algorithm it is excluded from being processed by the Fast Retransmission algorithm again. In order to guarantee that those packets are skipped the flag *no_fr_allowed* from the DATA chunk's management structure is set.

The next time a SACK is processed and the function *sctp_stike_gap_ack_chunks* is called all DATA chunks that are marked by the flag *no_fr_allowed* are skipped, as illustrated in listing 6.2. The code extract from listing 6.2 is part of the for loop that iterates over all DATA chunks. The variable *tp1* is a pointer to the current DATA chunk element in the *sent_queue*.

```
1        if (tp1->no_fr_allowed) {
2            /* this one had a timeout or something */
3            continue;
4        }
```

Listing 6.1: Retransmitted TSN Skip

If such a retransmitted DATA chunk is dropped by the router *again* the loss will not be noticed by the Fast Retransmission algorithm, because the DATA chunk will be skipped. Therefore, the sender has to wait until a timer-based retransmission is triggered. In the testbed environment the timer is always set to the minimum retransmission timeout (RTO) as it is defined by the system control *net.inet.sctp.rto_min*. In FreeBSD the system control *net.inet.sctp.rto_min* is set to 1000 ms by default.

The described scenario also occurs if paths are rather similar. The difference in case of dissimilar paths is that the send buffer is already much more utilized than it is in a similar path scenario. The reason for this is that the cumulative acknowledgment moves more slowly and thus more

DATA chunks have to be queued over longer periods of time.

If a retransmitted DATA chunk is dropped by the router the resulting gap cannot be filled until this DATA chunk is resend by a timer-based retransmission which takes one second to trigger. This results in a huge Send Buffer Blocking issue because for about one second no new DATA chunks can be send. The last SACK before the cumulative acknowledgment moves forward only consists of one huge Gap-Ack Block with over 1000 selectively acknowledged TSNs.

To mark all DATA chunks that need to be excluded from the Fast Retransmission algorithm is part of the SCTP specification (see [5], subsection 7.2.4.). [5] states that all DATA chunks being fast retransmitted should be marked and thus ineligible for a subsequent fast retransmission. Another reason for marking DATA chunks is the Smart Fast Retransmission algorithm as proposed by [8]. The reason for Smart Fast Retransmissions is to prevent bursts of Fast Retransmissions, if a DATA chunk is scheduled for retransmission on a path that differs from its initial path.

However, the retransmitted TSNs that use the same path as they used initially can be processed by the Fast Retransmission algorithm again, because Fast Retransmission bursts cannot occur. In the current implementation the flag *no_fr_allowed* is set regardless of whether a the path changes.

6.1.3. Non Renegable Selective Acknowledgment (NR-SACK)

So far this thesis has only focused on ordered and reliable data transfer. As already described in chapter 3, SCTP can also be configured to send unordered data. If only unordered transfer is used the same challenges can be identified as they have been discussed in the previous chapter 5 and sections 6.1.1 and 6.1.2.

It does not matter if the unordered or ordered data service is used as long as the traffic is reliable the receiving endpoint needs to acknowledge every single DATA chunk. As a consequence of reliable transfer all DATA chunk that are not part of the cumulative acknowledgment need to reside in the send buffer until they are cumulatively acknowledged. All DATA chunks that have been selectively acknowledged can be revoked by the receiving endpoint and then need to be retransmitted. The discarding of previously selectively acknowledged DATA chunks is

also denoted as reneging.

When it comes to unordered data traffic it is not necessary to keep those DATA chunks in the send buffer once they have been delivered to the receiving endpoint. These DATA chunks can be passed directly from the receiver to the application. The reason for this is that they are unordered and therefore no sequence needs to be maintained.

Therefore, the SACK chunk is extended by another type of Gap-Ack Block. This extension is denoted as Non Renegable Selective Acknowledgment (NR-SACK) and proposed by [41]. As the name implies the NR-SACK chunk enables the receiving endpoint to report TSNs that are selectively acknowledged but also not renegable. Thus, a nr-selectively acknowledged DATA chunk can be removed from the sending endpoint's buffer, because the receiving endpoint has already passed it to the application layer and will never revoked the DATA chunk.

The Figure 6.1 displays the structure of a NR-SACK chunk.

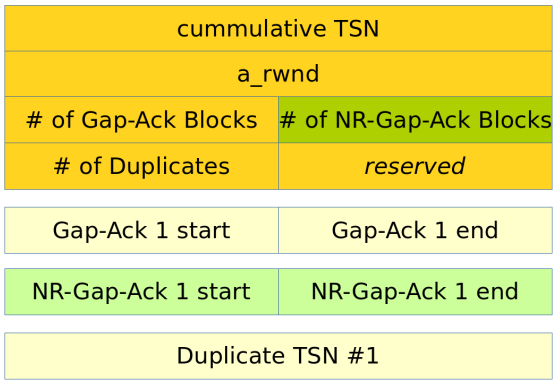

Figure 6.1.: Structure of a NR-SACK chunk

In comparison to a SACK chunk the NR-SACK carries the following additional information (highlighted in green): A two byte field that holds the number of reported NR-Gap-Ack Blocks. Furthermore, the actual NR-Gap-Ack Blocks are inserted after the Gap-Ack-Blocks and before the report of duplicate TSNs.

Figure 6.2 presents a measurement using unordered transfer with NR-SACKs, Buffer Splitting and DAC enabled. It can be observed that the maximum ideal throughput can be achieved if the send buffer is configured between 200 KB and 400 KB.

The DTrace probes for SCTP show that in the case of ideal send buffer sizes no Fast Retransmissions or time-based retransmissions were triggered. However, if the send buffer size grows the number of Fast Retransmissions and time-based retransmissions also grows. In the test environment Fast Retransmissions can only appear if packets get dropped by the soft routers North or South. A retransmission of any kind always decreases the *cwnd* and *ssthresh* which then can lead to a reduced utilization of the affected communication path.

Therefore, the range of ideal send buffer size has to be large enough to almost fully utilize the network paths at all time. On the other hand, if the buffer is too small the paths cannot be used to their full capacity. If the buffer is too large it not only triggers retransmissions, but the function *sctp_handle_sack* also need more time to compute the SACKs.

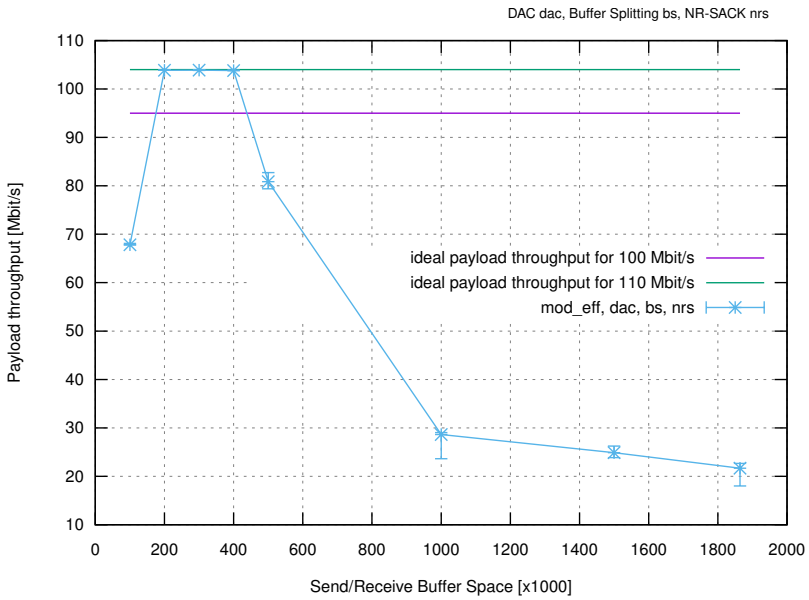

Figure 6.2.: Throughput Measurement using NR-SACKs with different send buffer sizes

The following Figure 6.3 displays measurement series with varying send buffer sizes and different delays on both paths. The first measurement (depicted in green) is performed with an ideal buffer size (compare Figure 6.2) and a delay of 1 ms. This configuration is able to fulfill the requirements of the first design rule *Improve Throughput* up to 120 Mbit/s.

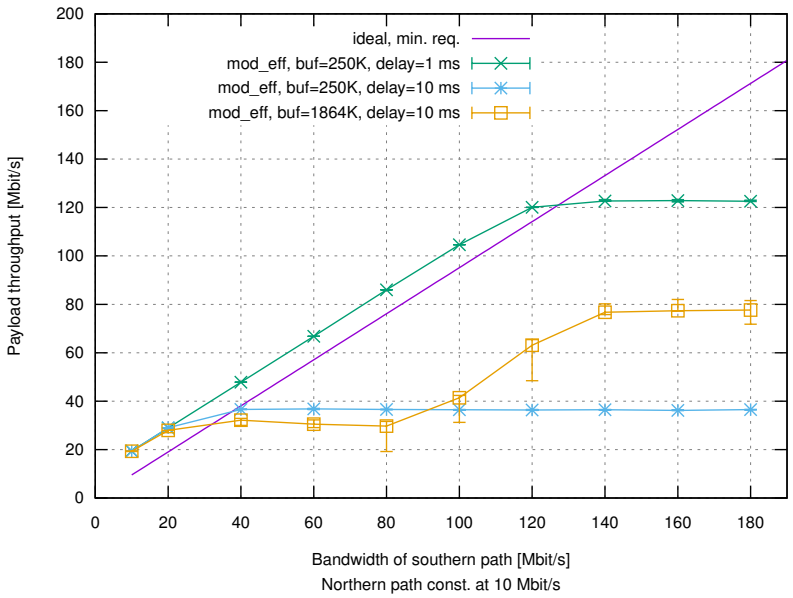

Figure 6.3.: Throughput Measurement using NR-SACKs with varying delay

However, in the second measurement series (depicted in blue) the delay is increased to 10 ms. In this case the configuration performs less well and the first design rule can only be met in a rather similar case. It is probably possible to find a buffer size that is suitable for such a case, but in a more realistic environment – like the Internet – the characteristics of a path can change. Moreover, as the third measurement (depicted in yellow) shows a different buffer size (the FreeBSD default buffer size) can show non-linear behavior. The other measurements can be approximates by a linear function that at a certain point turns into a constant function. The third measurement however shows a different behavior. Therefore, it is less likely to find a buffer size that works for a vast range of dissimilar scenarios as they can appear in the Internet.

6.2. Approach

In the previous section 6.1 the challenges that come with a dissimilar path scenario were discussed. Therefore the following approaches are presented:

1. Flag no_fr_allowed in Fast Retransmission algorithm
2. SACK Window – limits the number of TSNs in Gap Reports
3. Chunk Rescheduling Algorithm – A modified approach

Due to time regulations for this thesis the approaches 2 and 3 should rather be viewed as an idea and not an approach that is discussed to its fullest.

6.2.1. Flag no_fr_allowed in Fast Retransmission algorithm

As described in the previous subsection 6.1.2, a setting of the flag *no_fr_allowed* can cause all communication paths to stall if such a DATA chunk gets lost in the network, e.g. is dropped by a router.

There are two possible approaches for the described issue: The first would be to use a self-tuning Active Queue Management (AQM) algorithm that reduces the risk of dropped packet bursts. However, this approach is not discussed further because it is not a focus of this thesis. The second approach would be to allow DATA chunks that are retransmitted over the same path to be processed by the Fast Retransmission algorithm again. In the following the second approach is discussed.

The flag *no_fr_allowed* is set in the function *sctp_strike_gap_ack_blocks* , that schedules TSNs for retransmission using the Fast Retransmission algorithm. In the current implementation the flag is set before the function *sctp_find_alternate_net* is called. This function selects the currently best path and returns it to the caller. Only if the returned alternate path differs from the original path the flag *no_fr_allowed* needs to be set. Since, there is some additional work to do if the path changes a suitable branch already exists.

The modifications that need to be taken are displayed in listing 6.2. After *sctp_find_alternate_net* selects the best path it is stored in *alt*. The path that the DATA chunk *tp1* was originally send over is stored in *tp1->whoTo*. Therefore, the flag *no_fr_allowed* can be set in line 3 of listing 6.2.

```
if (alt != tp1->whoTo) {
    /* set flag, because path has changed */
    tp1->no_fr_allowed = 1;

    /* yes, there is an alternate. */
    sctp_free_remote_addr(tp1->whoTo);

    /* sa_ignore FREED_MEMORY */
    tp1->whoTo = alt;
    atomic_add_int(&alt->ref_count, 1);
}
```

Listing 6.2: Set *no_fr_allowed* in *sctp_strike_gap_ack_blocks*

6.2.2. SACK Window

With the efficiency modifications proposed in 5.3 the average CPU Usage time can be reduced from 12.5 % to 5 %. However, in a similar path scenario the function *sctp_handle_sack* only has a CPU usage time of 2.5 %. Therefore, in order to further decrease the computation time of *sctp_handle_sack* the number of reported TSNs in the Gap-Ack Blocks will be limited. This approach will be denoted as SACK Window *swnd*.

A maximum size for the Gap Report can guarantee that the CPU usage of the function *sctp_handle_sack* also has an upper limit. Section 5.2 already shows that a maximum size of the array *outstanding_tsn_arr* does not lead to a reduction of throughput if the configured maximum size is not too small. Actually, an upper limit for the number of Gap-Ack Blocks – not number of selectively acknowledged TSNs – already exists in the SCTP implementation. Since, a SACK cannot be split into two SACK chunks the number of Gap-Ack Blocks is limited by the allowed payload size. The allowed payload size is the MTU minus the SCTP common and SACK chunk header. If the MTU is 1500 Bytes the allowed payload size is 1452 Bytes. Each Gap-Ack Block has a size of $2 * 2Bytes = 4Bytes$. Therefore, a total of 363 Gap-Ack Blocks can be reported in a SACK.

Simply limiting the number of Gap-Ack Blocks is not a solution, because each Gap-Ack Block can report multiple TSNs. In the worst case a SACK consists of one Gap-Ack Block but selectively acknowledges several hundred TSNs.

An example for this issue is described in subsection 6.1.2: In the case a of a timer-based retransmission, the TSN that was dropped again by the router caused a gap in the TSN sequence that could could not be filled for about one second. This led to SACKs that over time reported less Gap-Ack Blocks but more selectively acknowledged TSNs. The last SACK that was processed before the cumulative acknowledgment moved forward only consisted of one Gap-Ack Block but reported over 1000 TSNs.

Therefore, the number of selectively acknowledged TSNs need to be limited and not the number of Gap-Ack Blocks. The SACK window *swnd* is defined as follows in equation 6.1:

$$last\_reported\_tsn = cumack + swnd \tag{6.1}$$

This approach can guarantee that the function *sctp_handle_sack* only has to iterate over a predefined amount of DATA chunks. If the number of selectively acknowledged TSNs would be counted with *swnd* the resulting computation time of *sctp_handle_sack* could differ, because possibly missing TSNs are not included. For this reason the *swnd* is defined as an offset from the cumulative acknowledgment *cumack*.

In order to limit the number of selectively acknowledged TSNs the mechanism that limits the number of Gap-Ack Blocks can be used and modified. The listing 6.3 displays the mentioned code section. The SACK chunk is basically just a huge chunk of allocated memory that is set step by step. The variable *gap_descriptor* points to the address of the next Gap-Ack Block that can be filled. The data structure *struct sctp_gap_ack_block* represents a Gap-Ack Block and consists of two members: *start* and *end*. The variable *limit* is a pointer to the end of the allocated memory for a SACK chunk.

```
1 if (((caddr_t)gap_descriptor + sizeof(struct sctp_gap_ack_block))
2     > limit) {
3     /* no more room */
4     limit_reached = 1;
5     break;
6 }
```

Listing 6.3: Assembly of a SACK chunk – stop criterion for Gap-Ack Blocks

In order to limit the number of selectively acknowledged TSNs the boolean expression displayed in listing 6.3 line 1 is modified to also stop adding new Gap-Ack Blocks if a certain *swnd*

threshold is exceeded.

Listing 6.4 displays the modifications. The system control *net.inet.sctp.cmt_swnd* defines if the limitation should be enabled and how many selectively acknowledged TSNs are allowed to be reported in a SACK. If the branch is executed the last Gap-Ack Block might need to be modified if more TSNs are reported than configured by the the SACK window *swnd*.

```
1  // swnd = SCTP_BASE_SYSCTL(sctp_cmt_swnd) &&
2  //        num_tsn_in_gap_blocks >= SCTP_BASE_SYSCTL(sctp_cmt_swnd);
3  if ((((caddr_t)gap_descriptor + sizeof(struct sctp_gap_ack_block))
4      > limit) || swnd ) {
5      if (num_tsn_in_gap_blocks > SCTP_BASE_SYSCTL(sctp_cmt_swnd)) {
6          // modify last Gap-Ack Block
7          gap_descriptor--;
8          gap_descriptor->end = htons(SCTP_BASE_SYSCTL(sctp_cmt_swnd));
9          gap_descriptor++;
10     }
11
12     /* no more room */
13     limit_reached = 1;
14     break;
15 }
```

Listing 6.4: Assembly of a SACK chunk – modified stop criterion

6.2.3. Chunk Rescheduling

As already identified in subsection 6.1.3 the difference between unordered and ordered transfer is as follows: In case of ordered transfer an user message can only be passed to the application if all previous user messages have been received. For unordered transfer user messages can be passed to the application as soon as they arrive at the receiving endpoint.

NR-SACKs can only be successfully applied to unordered transfer. In a ordered scenario a modified NR-SACK algorithm could reduce the problem of Send Buffer Blocking if the receiver guarantees that it takes responsibility for received user messages. However, a modified NR-SACK approach could not reduce the Receive Buffer Blocking issue, because the receiver has to buffer the received user messages until the user messages are in sequence. Thus, the blocking

issue only shifts from the sending endpoint to the receiving endpoint.

A different approach is Chunk Rescheduling as proposed by [8] and [29]. The Chunk Rescheduling algorithms approximates the level of send or receive buffer blocking and if a certain threshold is exceeded a user message is resend in order to fill a gap. The goal is to detect signs of Buffer Blocking early and react by retransmitting the TSN that causes the gap. As a result the gap is filled and the cumulative acknowledgment can advance.

In the original approach only the lowest possibly missing TSN is retransmitted. In the modified approach all possibly missing TSNs on the affected path prior to the first selectively acknowledged TSN are retransmitted. This seemed to produce more stable results.

Furthermore, the affected path is not excluded from Chunk Rescheduling for one round trip time (RTT). This part of the algorithm was not implemented because of a lack of time for the implementation.

6.3. Evaluation

This chapter evaluates the proposed approaches from the previous section 6.2.

6.3.1. Flag no_fr_allowed in Fast Retransmission algorithm

Figure 6.4 displays three measurements: The first measurement (depicted in green) shows the results of the reference implementation. The second measurement (depicted in blue) shows the results of the efficiency modified implementation. The last measurement (depicted in yellow) shows the results of the proposed modification to the *no_fr_allowed* flag.

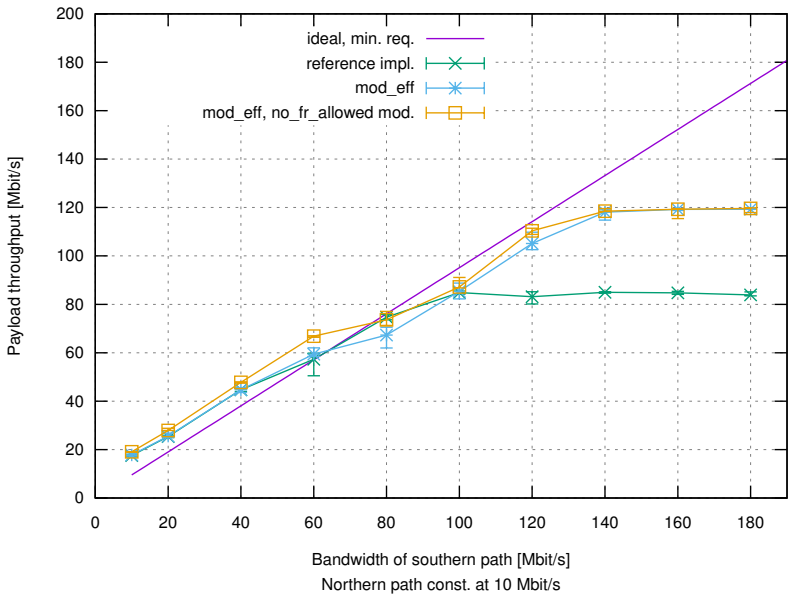

Figure 6.4.: Throughput measurement with no_fr_allowed modification

An increased throughput can be observed if the Southern Path is set between 40 Mbit/s and 80 Mbit/s. At 60 Mbit/s the *no_fr_allowed* modified implementation reaches the maximum possible throughput. The efficiency modified implementation increases the throughput over almost the whole measurement range, only at 80 Mbit/s it performs worse. The *no_fr_allowed* modified implementation manages to perform at least as good as the reference implementation at 80 Mbit/s.

It is still possible that a timer-based retransmission can stall all communication paths. In such a case the TSN is resend over a different path, gets dropped by the router and can only be resend once a retransmission timeout occurs. However, the results from Figure 6.4 show that it is beneficial to include retransmitted TSN in the Fast Retransmission algorithm if they are resend over the same path.

Further improvements can be observed, as Figure 6.5 presents, if the DAC algorithm and Buffer Splitting is enabled for both implementations.

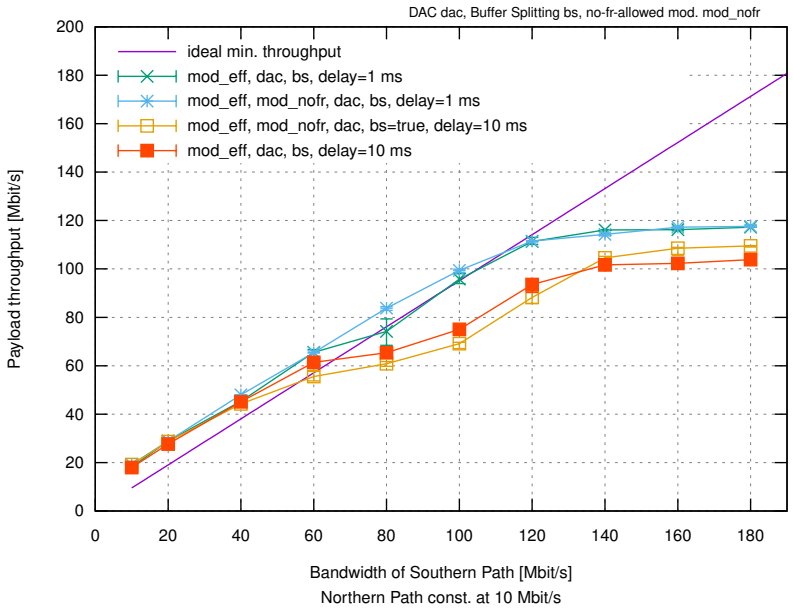

Figure 6.5.: Throughput measurement with no_fr_allowed mod. using DAC and Buffer Splitting

With DAC and Buffer Splitting enabled the *no_fr_allowed* modified implementation manages to further improve the throughput from 80 Mbit/s to 100 Mbit/s. Up to 80 Mbit/s the maximum theoretically possible throughput is reached.

However, the *no_fr_allowed* modified implementation is still greatly affected by an increase in delay as illustrated by the yellow measurement series. To compare the yellow measurement series to the efficiency modified implementation the red measurement is also added to Figure 6.5.

6.3.2. SACK Window

Figure 6.6 displays the results from the measurements using different SACK Window *swnd* sizes in comparison to the efficiency modified implementation. It can be observed that with an increase in *swnd* the overall throughput also increases.

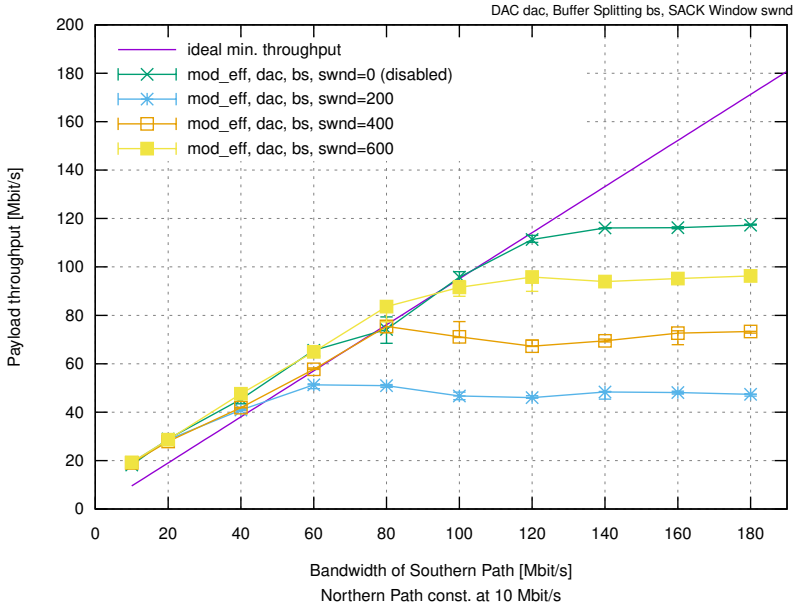

Figure 6.6.: Throughput measurement with varying SACK Windows

If the *swnd* is set to 400 the CPU usage of about 2.5 % is about the same as in a similar path scenario. However, only *swnd* sizes close too 1000 were able to reproduce results – in greater dissimilar scenarios – that are close to the efficiency modified implementation. Therefore, just limiting the number of selectively acknowledged TSNs by using the proposed *swnd* is not sufficient.

The reason for this is that the *flightsize* counter is only partly modified. A *flighsize* counter keeps track of the amount of Bytes that are outstanding on a path. The sending endpoint is only allowed to send out new DATA chunks if the *flightsize* is less than the *cwnd*.

Measurements have shown that if the *swnd* is configured too small, e.g. 200, the sending endpoint is not allowed to send new DATA chunks, because in about 90 % of the time the *flightsize* is greater than the *cwnd*.

If only the first 200 TSNs are reported in the Gap-Ack Blocks the function *sctp_handle_segments* only processes these TSNs and decreases the *flighsize* counter accordingly. To the sending endpoint it appears as if some TSNs arrived at the receiving endpoint, but all TSNs greater the

cumack + swnd are still in flight. If these TSNs are still outstanding the *flightsize* counter is not decreased for those TSNs. This then results in an artificially higher *flightsize* counter and thus also decreases the ability to send out new DATA chunks.

Therefore, the idea emerged to process the *sent_queue* piecewise. In this extended approach all Gap-Ack Blocks are reported in a SACK by the receiving endpoint, but the sending side performs the limitations. This alternate approach always processes the TSNs, that are part of the *swnd* and additionally process TSNs of a second window that constantly moves through the rest of the *sent_queue*. An example is illustrated in Figure 6.7. Figure 6.7 displays a *sent_queue* that is logically divided into two parts. The first part, the swnd, is processed every time the function *sctp_handle_sack* is called. For the remaining part each window 1 to 4 is only processed every fourth time the function *sctp_handle_sack* is called.

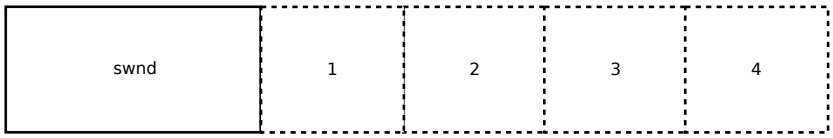

Figure 6.7.: Alternate SACK Window approach

The purpose of the second window – 1 to 4 in the example in Figure 6.7 – is only to decrease the *flightsize* counters and therefore provide a more accurate count of outstanding TSNs.

With this alteration it is possible to increase the overall throughput, but it was never able to outperform the efficiency modified implementation. Furthermore, a maximum computation time of the function *sctp_handle_sack* cannot be guaranteed any more because a fifth iteration step would be needed if more DATA chunks would be queued.

Another possible solution for this issue would be to artificially decrease the *flightsize* counter for the TSNs that were not reported, but were received at the receiving endpoint. This could be done by an additional field in the SACK chunk. However, artificially decreasing the *flightsize* counter has to be done carefully, because it is an important part of the Congestion Control mechanism. If the receiving endpoint also reports the amount of Bytes received after the *swnd* the sender has to be able to decide for which TSN it needs to decrease the *flightsize* and for which it was artificially decreased.

6.3.3. Chunk Rescheduling

Figure 6.8 presents the results of the chunk rescheduling measurement: One measurement (depicted in blue) is done with a delay of 1 ms and the other (depicted in yellow) at 10 ms.

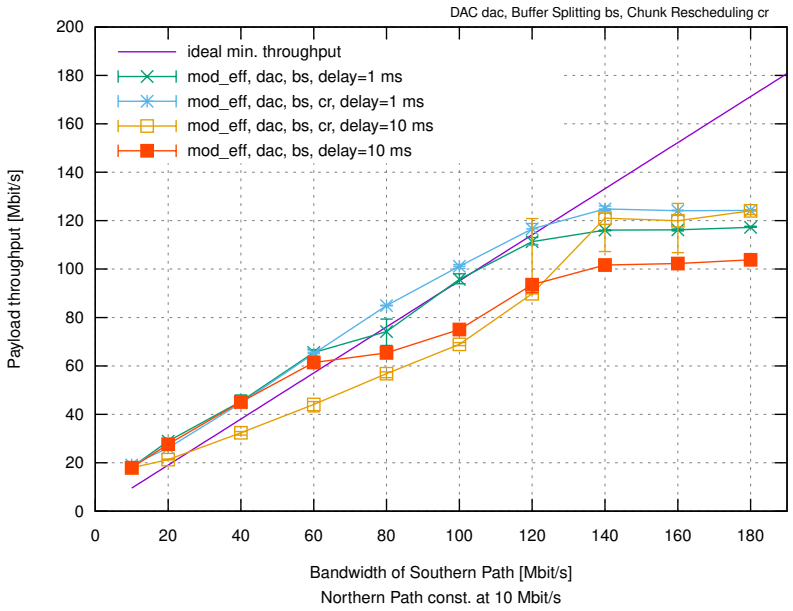

Figure 6.8.: Throughput measurement using the modified Chunk Rescheduling algorithm

In case of the 1 ms delay measurement the Chunk Rescheduling algorithm is able to outperform the efficiency modified implementation, especially if the Southern path is set to 140 Mbit/s or more.

If the delay on both paths is increased from 1 ms to 10 ms the measurement that uses Chunk Rescheduling (depicted in yellow) is able to outperform the efficiency modified implementation in the case of greater dissimilarity. At lower dissimilarity levels the Chunk Rescheduling approach performs worse than the previous efficiency optimized approach.

However, if the Southern Path is set to 100 Mbit/s 68.1 % of the SACKs selectively acknowledged less than 100 TSNs. The remaining SACKs include no Gap-Ack Blocks at all. About 50 % more TSNs are retransmitted due to the Chunk Rescheduling algorithm, but they only make up

3.46 % of the total data transferred.

Therefore, Chunk Rescheduling helped to reduce the average number of selectively acknowledged TSNs. Through the retransmission of TSNs, that might hold back the cumulative acknowledgment, the Chunk Rescheduling algorithm was able to make the cumulative acknowledgment move more often and thus it reduced the time that DATA chunk spent in the *sent_queue*.

For future work it might be worth trying to combine the an improved SACK Window approach with Chunk Rescheduling. This could help to limit the CPU Usage of the function *sctp_handle_sack* , but also make sure that the Buffer Blocking issue is reduced by the Chunk Rescheduling algorithm.

7. Conclusion

The goal of this thesis was to identify issues that arise if CMT-SCTP is used in a dissimilar path environment and – if possible – propose solutions that help to reduce the issues with the goal to fulfill the requirements of the first desgin rule *Improve Throughput* as defined by [1].

With a more efficient implementation of the SACK handling algorithm it is possible to increase the throughput in a dissimilar multi-flow scenario. Depending on the degree of dissimilarity the throughput could be increased by up to 29 % in the test environment. However, it is not enough to meet the first desgin rule's requirements, especially not if a greater delay was added to the Northern and Southern Path.

Therefore, it could be concluded that the issues of a reduced throughput in a dissimilar multi-flow scenario cannot be solely solved algorithmically.

The protocol related discussion revealed that a great issue is Send and Receive Buffer Blocking that is caused by the SCTP's DATA chunk scheduling algorithm and the fact that the paths are dissimilar. The Chunk Rescheduling algorithm tries to identify signs of Send and Receive Buffer Blocking and reduces it by resending DATA chunks that cause gaps. The approach tries to ensure that the cumulative acknowledgment constantly moves and thus releases buffer space in order to send new DATA chunks. However, in case of greater dissimilarity and increased delay all approaches struggle to fulfill the first desgin rule's requirements.

A different approach to strike the Buffer Blocking issues would be to implement a different DATA chunk scheduling algorithm that tries to schedule DATA chunks so that the possibility of scheduling-induced gaps are minimized. Furthermore, the SACK window approach in conjunction with Chunk Rescheduling might further improve the throughput, if the issues with the artificially to high *flighsize* counters can be solved.

The challenges can become even greater if other Congestion Control algorithms CMT-CCs than *cmt* – that was used in this thesis – are used as it is illustrated in Figure A.2.

A. Appendix

A.1. Testbed

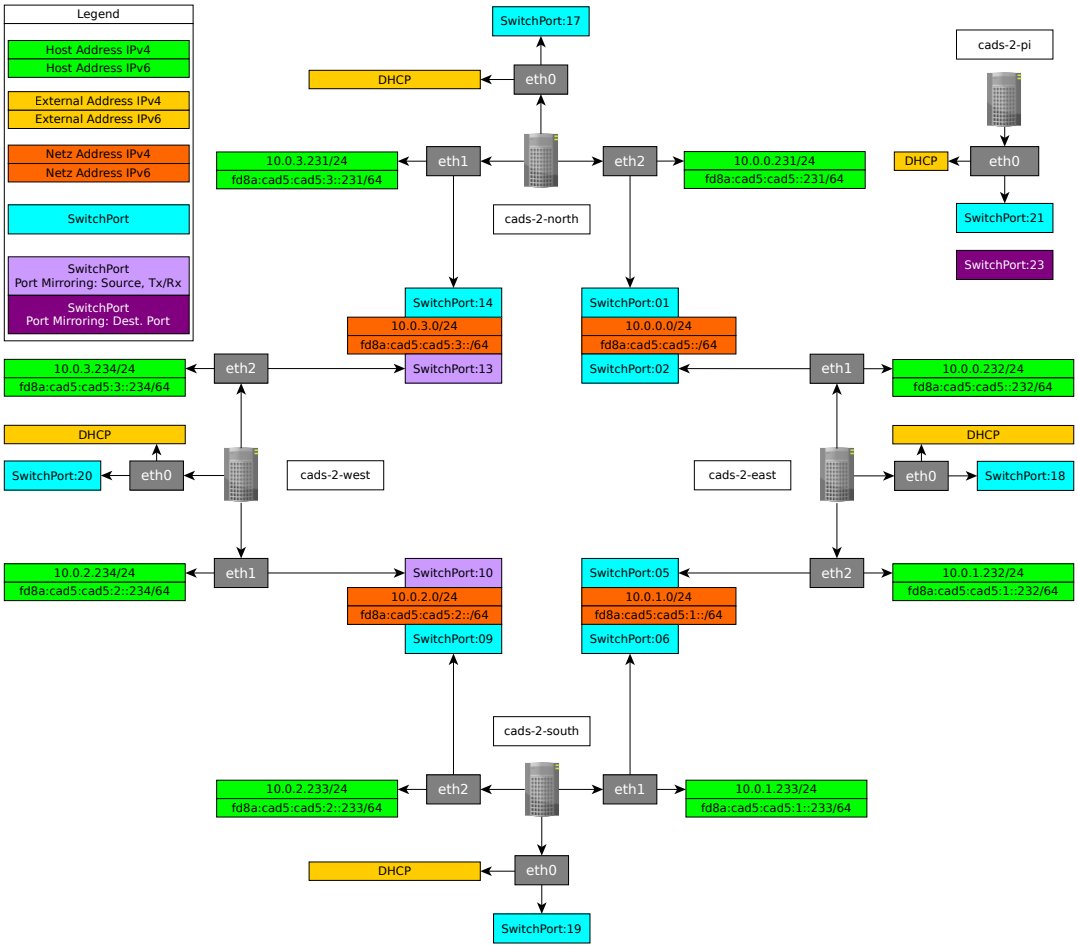

Figure A.1.: Detailed Testbed Setup

A.2. Additional Measurements

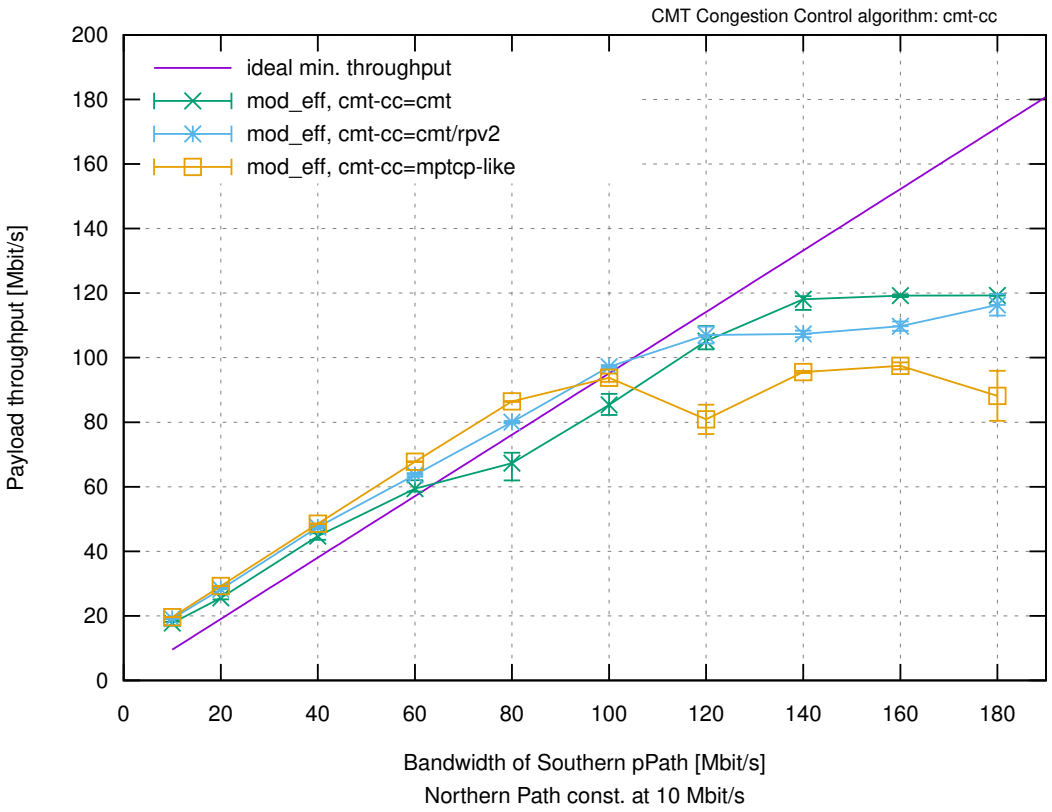

Figure A.2.: Throughput measurement using the efficiency modified implementation proposed in section 5.2 with different CMT Congestion Control algorithms

A.3. Additional Work

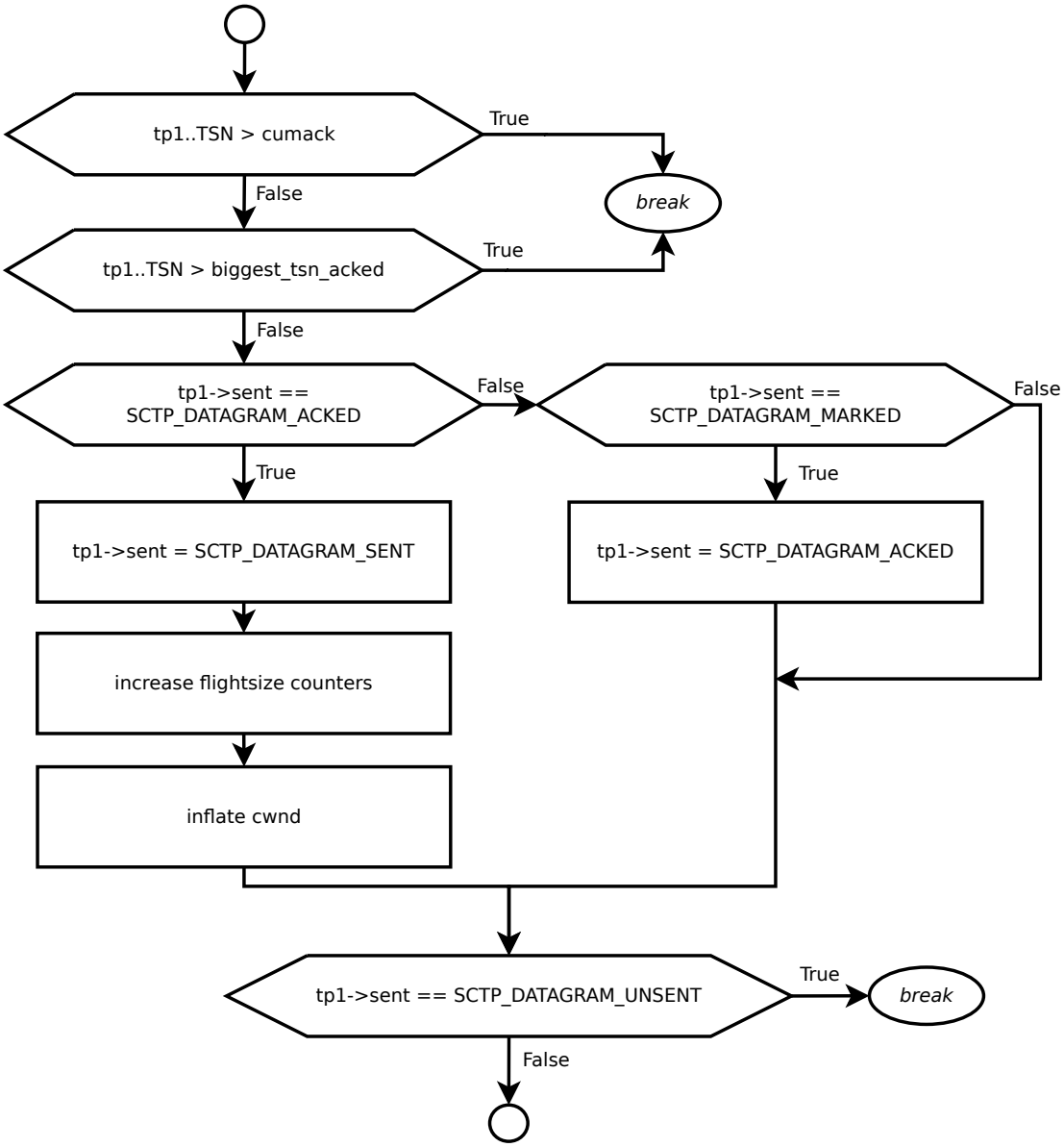

Figure A.3.: Extract from *sctp_check_for_revoked* : Inside the loop that iterates over all DATA chunks from the *sent_queue*

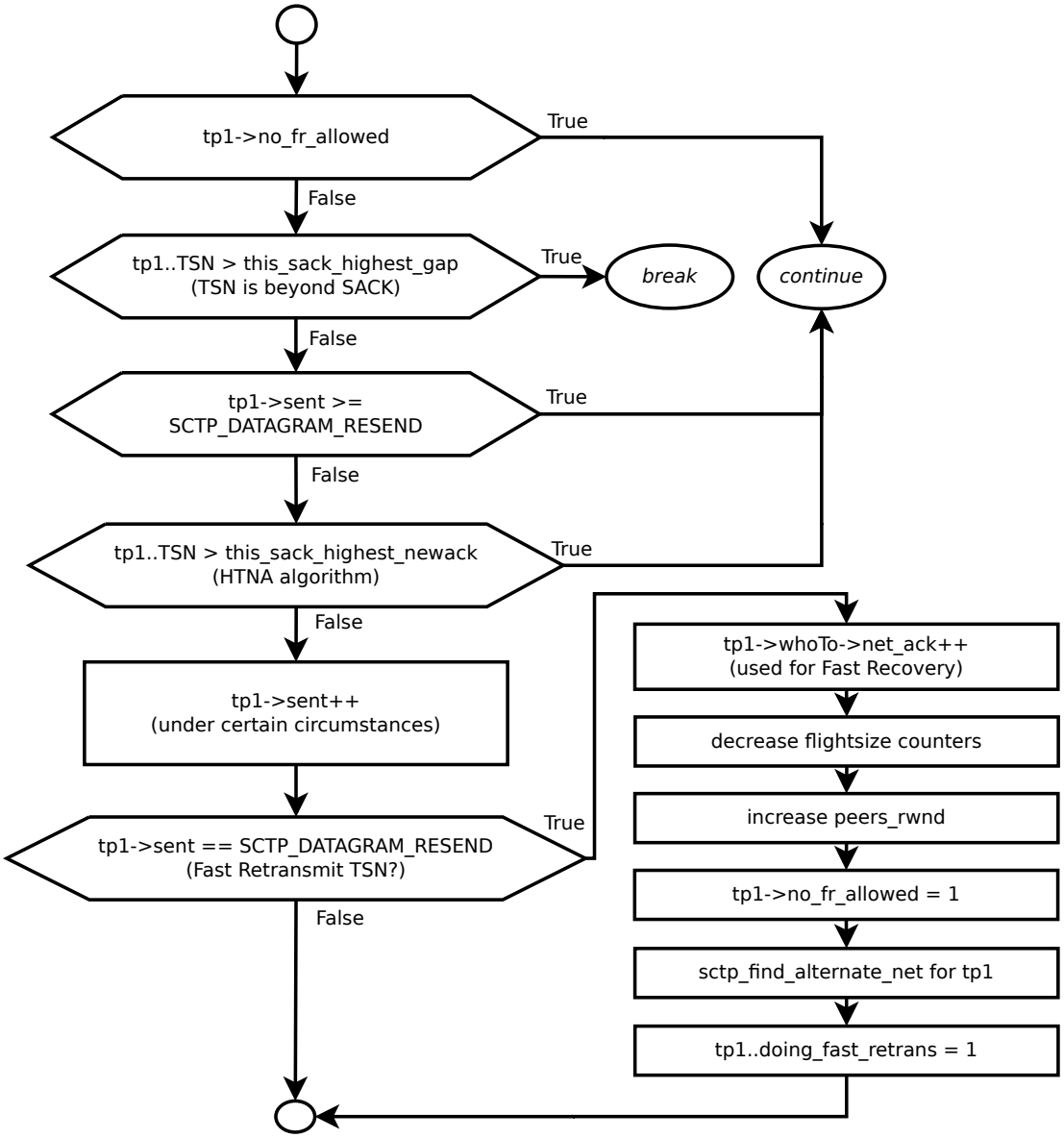

Figure A.4.: Extract from *sctp_strike_gap_ack_blocks* : Inside the loop that iterates over all DATA chunks from the *sent_queue* (strongly simplified)

Bibliography

[1] Costin Raiciu, Damon Wischik, and Mark Handley. "Practical Congestion Control for Multipath Transport Protocols". In: *University College London, London/United Kingdom, Tech. Rep* (2009).

[2] Jon Postel. *Transmission Control Protocol*. RFC 793. IETF, 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt (visited on 01/18/2017).

[3] IETF. *Internet Engineering Task Force (IETF)*. 2016. URL: https://www.ietf.org/ (visited on 02/28/2017).

[4] Thomas Dreibholz and Erwin P Rathgeb. "Towards the Future Internet–An Overview of Challenges and Solutions in Research and Standardization". In: *Proceedings of the 2nd GI/ITG KuVS Workshop on the Future Internet, Karlsruhe/Germany*. 2008.

[5] R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. IETF, 2007. URL: http://www.rfc-editor.org/rfc/rfc4960.txt (visited on 01/18/2017).

[6] Paul Amer et al. *Load Sharing for the Stream Control Transmission Protocol (SCTP)*. Internet-Draft draft-tuexen-tsvwg-sctp-multipath-13. IETF Secretariat, 2016. URL: http://www.ietf.org/internet-drafts/draft-tuexen-tsvwg-sctp-multipath-13.txt (visited on 01/18/2017).

[7] Janardhan R Iyengar, Paul D Amer, and Randall Stewart. "Concurrent Multipath Transfer using SCTP multihoming over independent end-to-end Paths". In: *IEEE/ACM Transactions on networking* 14.5 (2006), pp. 951–964.

[8] Thomas Dreibholz et al. "On the use of Concurrent Multipath Transfer over Asymmetric Paths". In: *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*. IEEE. 2010, pp. 1–6.

[9] Hakim Adhari et al. "Evaluation of Concurrent Multipath Transfer over dissimilar Paths". In: *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*. IEEE. 2011, pp. 708–714.

[10] J. Postel. *User Datagram Protocol*. RFC 768. IETF, 1980. URL: http://www.rfc-editor.org/rfc/rfc768.txt (visited on 02/23/2017).

[11] E. Kohler, M. Handley, and S. Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340. IETF, 2006. URL: http://www.rfc-editor.org/rfc/rfc4340.txt (visited on 01/18/2017).

[12] Robert Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. IETF, 1989. URL: http://www.rfc-editor.org/rfc/rfc1122.txt (visited on 01/18/2017).

[13] S. Floyd. *Congestion Control Principles*. RFC 2914. IETF, 2000. URL: https://tools.ietf.org/html/rfc2914 (visited on 01/18/2017).

[14] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. RFC 5681. IETF, 2009. URL: http://www.rfc-editor.org/rfc/rfc5681.txt (visited on 01/18/2017).

[15] Randeall R. Stewart and Qiaobing Xie. *Stream Control Transmission Protocol (SCTP) - A Reference Guide*. Addison-Wesley, 2002. ISBN: 978-0-201-72186-7.

[16] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2015.

[17] Jon Postel. *Internet Protocol*. RFC 791. IETF, 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt (visited on 02/23/2017).

[18] Armando L Caro Jr et al. "Congestion control: SCTP vs TCP". In: *Protocol Engineering Lab, Computer and Information Sciences, University of Delaware* (2003).

[19] S. Floyd et al. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 5348. IETF, 2008. URL: http://www.rfc-editor.org/rfc/rfc5348.txt (visited on 01/18/2017).

[20] F. Baker and G. Fairhurst. *IETF Recommendations Regarding Active Queue Management*. RFC 7567. IETF, 2015. URL: https://tools.ietf.org/html/rfc7567 (visited on 02/28/2017).

[21] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. "The Resource Pooling Principle". In: *ACM SIGCOMM Computer Communication Review* 38.5 (2008), pp. 47–52.

[22] PC Engines GmbH. *APU Alix (APU3C4)*. 2016. URL: http://www.pcengines.ch/apu2c4.htm (visited on 01/20/2017).

[23] Janusz Gozdecki, Andrzej Jajszczyk, and Rafal Stankiewicz. "Quality of service terminology in IP networks". In: *IEEE communications magazine* 41.3 (2003), pp. 153–159.

[24] Ugen J. S. Antsilevich et al. *ipfw(8)*. 2016. URL: https://www.freebsd.org/cgi/man.cgi?query=ipfw (visited on 01/18/2017).

[25] Luigi Rizzo. *dummynet(4)*. 2016. URL: https://www.freebsd.org/cgi/man.cgi?query=dummynet (visited on 01/18/2017).

[26] Sally Floyd and Van Jacobson. "Random Early Detection Gateways for Congestion Avoidance". In: *IEEE/ACM Transactions on Networking (ToN)* 1.4 (1993), pp. 397–413.

[27] Thomas Dreibholz. *NetPerfMeter*. 2017. URL: https://github.com/dreibh/netperfmeter (visited on 01/18/2017).

[28] Thomas Dreibholz et al. "Evaluation of A New Multipath Congestion Control Scheme using the NetPerfMeter Tool-Chain". In: *Proceedings of the 19th IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. Sept. 2011, pp. 1–6.

[29] Thomas Dreibholz. "Evaluation and Optimisation of Multi-Path Transport using the Stream Control Transmission Protocol". Habilitation Treatise. University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, Mar. 2012.

[30] Sun Microsystems and Joyent Inc. *Dynamic Tracing Guide*. 2017. URL: http://dtrace.org/guide/chp-intro.html (visited on 01/18/2017).

[31] Brendan Gregg. *Hotkernel Analysis*. 2016. URL: http://www.brendangregg.com/DTrace/hotkernel (visited on 01/18/2017).

[32] Wireshark Foundation. *Wireshark - Network Protocol Analyzer*. 2016. URL: https://www.wireshark.org/ (visited on 02/23/2017).

[33] Tcpdump. *Tcpdump/Libpcap Public Repository*. 2016. URL: http://www.tcpdump.org (visited on 01/18/2017).

[34] Hewlett Packard Enterprise. *HP 1820 Switches – Management and Configuration Guide*. 2017. URL: http://h20566.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=7687976&docId=emr_na-c04622710 (visited on 02/27/2017).

[35] C. Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. RFC 894. IETF, 1984. URL: https://tools.ietf.org/rfc/rfc894.txt (visited on 01/18/2017).

[36] Intel Corporation. *Intel Core i5 4690 Specification*. 2016. URL: http://ark.intel.com/products/80810/Intel-Core-i5-4690-Processor-6M-Cache-up-to-3_90-GHz (visited on 02/23/2017).

[37] M. Allman, V. Paxson, and W. Stevens. *TCP Congestion Control*. RFC 2581. IETF, 1999. URL: https://tools.ietf.org/html/rfc2581 (visited on 01/18/2017).

[38] Thomas Dreibholz et al. "Transmission Scheduling Optimizations for Concurrent Multipath Transfer". In: *Proceedings of the 8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*. Vol. 8. 2010.

[39] W. Richard Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols*. Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN: 0-201-63346-9.

[40] Bob Braden et al. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC 2309. IETF, 1998. URL: http://www.rfc-editor.org/rfc/rfc2309.txt (visited on 01/18/2017).

[41] Preethi Natarajan et al. "Non-renegable selective acknowledgments (NR-SACKs) for SCTP". In: *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*. IEEE. 2008, pp. 187–196.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 21. März 2017 Daniel Sarnow