Beneficial gradual deployment of SCTP

A diploma thesis submitted to the
Institute of Computer Science,
University of Innsbruck

for the degree of
Diplom-Ingenieur in Computer Science

presented by
Bakk.techn. Florian Niederbacher

supervised by
Dr. Michael Welzl

February 2010
To my parents
In a perfect world software has no bugs, following specifications would never lead you astray and all specifications have one and only one interpretation.

Alas! ... software does have bugs, specifications are not perfect and almost everybody has a different interpretation.

So we must carry out improvements furthermore!

(Florian Niederbacher)
Abstract

The Internet-wide deployment of new protocols such as the Stream Control Transmission Protocol (SCTP) is a difficult matter. The employment of SCTP would be beneficial in many cases, but it is a major challenge to enable applications to use the new protocol at both ends, client and server. The prevalence of the Transmission Control Protocol (TCP), which has been established as the dominant transport protocol in today’s Internet, therefore needs to be broken.

This master thesis discusses existing solutions and presents a technique for gradually deploying SCTP alongside TCP in two steps. First, it investigates the advantages and disadvantages of replacing a TCP connection with an SCTP association. Second, it demonstrates a new approach which let existing and new applications transparently use SCTP. The goals were to gain information about the benefits of SCTP’s multi-streaming feature and to explore the portability of existing TCP-based applications to SCTP.

Practical measurements show that transparent mapping of TCP connections onto SCTP streams is possible and that in certain cases a speedup can be achieved. The results are gained by experimental analysis in a typical small office or home (SOHO) network. Documented benefits and open issues give the reader an outlook on problems that can arise in such environments, as well as a good insight into SCTP multi-streaming application development. This makes this thesis a valuable guide for programmers who consider implementing SCTP based software in such a manner for a productive use in communication systems.
# Contents

## Abstract v

## Contents viii

## List of Figures x

## List of Tables xi

1 **Introduction** 1

2 **Background and Analysis of SCTP** 5

   2.1 **Computer Networks and Layer Models** 6
      2.1.1 The Internet Protocol Suite 7

   2.2 **The Stream Control Transmission Protocol** 8
      2.2.1 Basics and RFC Tracks of TCP and SCTP 9

   2.3 **Comparison of SCTP with TCP and UDP** 11
      2.3.1 The Packet Structure 15
      2.3.2 The Initiation Procedure 23
      2.3.3 The Shutdown Procedure 25
      2.3.4 Data Transfer - SACK, Flow and Congestion Control 27
      2.3.5 Features of SCTP - Multi-homing and Multi-streaming 33

   2.4 **SCTP Extensions** 39
      2.4.1 Partial Reliability Extension 39
      2.4.2 Dynamic Address Reconfiguration Extension 39
      2.4.3 Extensions in Draft 40

   2.5 **Summary and Conclusion** 41

3 **Motivation, Design and Implementation** 43

   3.1 **Preliminary Investigation** 44
      3.1.1 Available SCTP Applications 44
      3.1.2 TCP to SCTP Protocol Translator Software 45

   3.2 **Motivation** 47
      3.2.1 Intentions and Related Questions 48

   3.3 **The First Steps** 50
      3.3.1 A Simple Data Transfer 50
      3.3.2 Dual Setup of TCP and SCTP 52
      3.3.3 Parallel Data Transmission with Multi-Streaming 53

   3.4 **Practical Ideas for Implementations** 57
      3.4.1 First Proposal: SCTP encapsulation 57
## List of Figures

2.1 The Internet protocol suite - layer model ........................................... 7  
2.2 SCTP feature combination ............................................................... 8  
2.3 TCP RFCs ......................................................................................... 10  
2.4 SCTP RFCs ....................................................................................... 11  
2.5 PDU composition on different layers .................................................... 16  
2.6 Composition of TCP and SCTP packets with IPv4 in Ethernet networks .... 17  
2.7 General packet composition of UDP, TCP and SCTP ............................ 18  
2.8 Simple SCTP packet with one data chunk vs. chunk composition .......... 20  
2.9 SCTP chunk composition: DATA and INIT ........................................... 21  
2.10 Comparison of the initiation procedure of TCP and SCTP .................... 23  
2.11 Comparison of the termination procedure of TCP and SCTP ............... 26  
2.12 Acknowledgment procedures: (1) simple ACK and (2) delayed ACK .... 28  
2.13 Delayed SACK in SCTP .................................................................... 28  
2.14 Five different SACK cases of SCTP .................................................... 29  
2.15 Flow control mechanism - sliding window ........................................... 30  
2.16 Standard congestion control algorithms .............................................. 31  
2.17 Multi-homing .................................................................................... 34  
2.18 Single TCP and parallel TCP connection vs. SCTP multi-streaming .... 35  
2.19 Data transmission with multi-streaming and shared transmission control values .................................................................................. 36  
2.20 Detailed view of an SCTP association with multi-streaming ............... 37  
3.1 General overview of different operating system levels and where the mapping tools are attached (1 = withsctp, 2 = shim layer) ................................. 46  
3.2 Intended transfer behavior by a protocol translation with multi-streaming 49  
3.3 Thought process for considering various data transmission modes ........ 50  
3.4 Considerations for a dual setup of TCP and SCTP ............................... 52  
3.5 Differences between parallel transfer modes in SCTP: (a) parallel associations, (b) single association with multi-streaming ........................................ 53  
3.6 A wrong approach of SCTP multi-streaming for parallel data transfers 54  
3.7 Solution for parallel transmissions with SCTP’s multi-streaming .......... 55  
3.8 One-to-many style of SCTP sockets ...................................................... 56  
3.9 SCTP encapsulation ............................................................................ 57  
3.10 General overview of different applications with processes/threads and the corresponding socket access ................................................................. 59  
3.11 Possible application scenario after an applied protocol modification with SCTP multi-streaming ................................................................. 59  
3.12 Implementation model for a protocol mapping in user space ................ 60  
3.13 Proposal for a connection manager gateway (CMG) .......................... 62  
3.14 Standard function sequence for a client/server application .................. 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.15</td>
<td>Thread architecture of the implemented gateway server application</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Network testbed used for the practical measurements</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Acknowledgment procedures: TCP ACK vs. SCTP’s delayed SACK in slow start</td>
<td>81</td>
</tr>
<tr>
<td>4.3</td>
<td>Iperf measurements with default TCP and SCTP settings</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>Iperf measurements with a similar configuration for TCP and SCTP</td>
<td>88</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison of single file transfers with TCP and SCTP</td>
<td>90</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of single file transfers with TCP and SCTP with optimizations</td>
<td>92</td>
</tr>
<tr>
<td>4.7</td>
<td>Magnified view of Figure 4.6</td>
<td>93</td>
</tr>
<tr>
<td>4.8</td>
<td>The evolution of cwnd in slow start</td>
<td>93</td>
</tr>
<tr>
<td>4.9</td>
<td>TCP time sequence graph: problems with automatic buffer tuning</td>
<td>94</td>
</tr>
<tr>
<td>4.10</td>
<td>SCTP SACK procedure: delayed vs. no delay</td>
<td>95</td>
</tr>
<tr>
<td>4.11</td>
<td>Time for the data reception with SCTP: slow start vs. open cwnd</td>
<td>97</td>
</tr>
<tr>
<td>4.12</td>
<td>Results of performance measurements with iperf for various configurations</td>
<td>101</td>
</tr>
<tr>
<td>4.13</td>
<td>Time for loading a web page with a single image</td>
<td>105</td>
</tr>
<tr>
<td>4.14</td>
<td>Time for loading a web page with five images of same size</td>
<td>107</td>
</tr>
<tr>
<td>4.15</td>
<td>Analysis of a parallel HTTP request with firebug (TCP and CMG)</td>
<td>107</td>
</tr>
<tr>
<td>4.16</td>
<td>Time required for a serial download of five files with the same size</td>
<td>109</td>
</tr>
<tr>
<td>4.17</td>
<td>Time required for a serial download of five files with the same size over the same connection (connection reuse)</td>
<td>110</td>
</tr>
<tr>
<td>4.18</td>
<td>Time required for a parallel download of five files with the same size.</td>
<td>111</td>
</tr>
<tr>
<td>A.1</td>
<td>SOHO testbed network structure</td>
<td>127</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Feature comparison of SCTP, TCP and UDP ................................. 15
2.2 Basic SCTP chunks ............................................................... 22

4.1 Summary of important testbed network details .......................... 79
4.2 Default settings of buffer sizes and the MSS ............................ 83
4.3 Bandwidth-delay product in a 100 Mbps network for different path delays ................................. 84
4.4 Number of data packets for the file transfer if MSS-sized PDUs are sent ........................................ 90

A.1 Hardware details: Linux router ............................................... 128
A.2 Hardware details: Linux host ............................................... 128
Chapter 1

Introduction

The Stream Control Transmission Protocol (SCTP) is a reliable transport protocol that provides more robustness than TCP, and delivery of data between two endpoints (like TCP) with message boundaries preservation (like UDP). Additionally it has advantageous capabilities, such as multi-homing\(^1\) and multi-streaming\(^2\). These features increase availability and solve the “Head-of-line blocking” problem\(^3\).

Despite the many novelties and benefits of this protocol, its usage in the Internet is rare. While the individual advantages are application dependent, the deployment of new protocols is a general challenge. SCTP, like many other network protocols, suffers from the often discussed “chicken and egg” problem: application developers are not interested in developing SCTP applications because hosts do not provide services supporting SCTP, and system developers do not implement SCTP services because there are no applications available that use SCTP. The problem originates in the TCP/IP layer model. SCTP must assert itself against its predecessors TCP and UDP on the transport layer, and might has to be introduced in such a manner that the benefits of the protocol modifications are perceptible by the end user without putting any constraints to or limiting the functionality of existing applications. The protocol implementation must happen at the service provider as well as end-user side with as little effort as possible, allowing to easily switch between the old (TCP) and the new protocol (SCTP).

There are two possibilities for the practical realization: reprogram the applications with support for various protocols, or modify the operating system functionalities at different layers. Given the multitude of applications and thus application developers, it’s highly unlikely (too much effort) that all of them will start to implement the support for new protocols.

\(^1\) Endpoints use multiple IP addresses that share a single port – called association.
\(^2\) One association can have more streams. Each stream independently provides a specified delivery method (ordered or unordered).
\(^3\) The transfer of packets on a TCP connection can get blocked by the strict packet ordering if a retransmission is necessary.
The most reasonable solution seems to provide a protocol translator somewhere on
the communication path (e.g. operating system, firewall...). A transparent and general
approach is needed, which allows a possible later upgrade or replacement. The intended
modification must be accomplished in an easy manner at both endpoints. An alternative for
switching back to the old protocols needs to be available as well, to prevent that in case of
bad performance the introduction of SCTP yields exactly to the opposite effect and starts
to be counterproductive for increasing the amount of SCTP applications. A few solutions
that meet the aforementioned guidelines already exist. Current software working at a layer
between the application and the operating systems network stack are:

- the “withsctp-tool” included in the Linux SCTP package (LKSCTP) [1]
- the “shim layer” implemented as a kernel module in BSD based operating systems [2]

Both translate network system calls to TCP into corresponding calls to SCTP. This
happens without any modifications to the application source code. However, the application
fields of these solutions are restricted by the operating system and not all advanced features
of SCTP were taken into account. The multi-streaming feature, for example, is neglected
by both approaches and the address configuration for multi-homing is not conveniently
solved either. Further differences between these two translators are the portability and
the general use with respect to the already mentioned support of the old protocol. The
technical differences of both are compared in this thesis, and the main ideas are taken on for
the development of a new tool that uses an additional feature of SCTP – multi-streaming.

A few attempts were made to port this property toward existing applications. In
Chapter 3 it is explained how this can finally be implemented with a middleware solution.
Different implementation ideas are discussed, and for a potentially reasonable approach
a proof of concept implementation was developed. The ultimate goal would have been to
introduce all characteristics of SCTP but this can only happen gradually. It is too complex to
integrate all properties at once. The appearing difficulties and side-effects have still not been
examined in detail, as one will see in the documentation of the experimental measurements.
However, the main problem is the technical feasibility of introducing multi-streaming, for
which a solution is presented.

The benefit from the multi-streaming usage is documented by theoretical evaluations
and practical performance measurements in Chapter 4. In principle, before the new approach
can be considered, it must be studied if SCTP is generally able to transmit data as fast as
TCP. Many performance results from various scientific papers reflect only special cases and
do not hold generally. Moreover, SCTP implementations are still under development and
many experimental results are not up-to-date anymore. It has to be noted that performance
is first of all implementation dependent (protocol implementation in the kernel) and varies
with the operating system. The next difficulty is the general comparison of TCP with SCTP
– one cannot compare apples with pears. The comparison is done with a special configuration
of TCP, which tries to be as similar as possible to SCTP. Clear rules must be defined in advance for the performance evaluation to satisfy the scientific rule of repetition, and adequate test tools must be used for it. Hence the next problem: A variety of measurement tools for TCP exist, while they are quite rare for SCTP. Furthermore, no tool is available, which can perform a data transmission measurement according to the same principles for both protocols and a simple software element had to be developed for the purpose of the experiments presented here. It is capable of logging measurement results and allows variable parameterization. This software is needed to certify that tests are carried out on equal grounds.

Various approaches and ideas for solutions did not prove to be right and did not lead to the desired result. These ideas and the corresponding error sources are documented in this thesis and should be a help to prevent programmers from falling into the same pitfall again. Other revealed problems are protocol gaps in functionality and some probably missing extensions. The corresponding suggestions are given at the end. Existing drafts from where ideas are taken on and other scientific works that probably will have an influence on the further deployment of SCTP are summarized in the related work chapter.

The obtained final results reveal a possible general approach, on how SCTP can be used in the future – the dawn of a possible new Internet-Draft for the implementation of a connection or congestion manager?

The structure of this thesis consists of the following three major parts:

- Chapter 2 contains an introduction and analysis of selected details of SCTP compared to other transport protocols. Special attention has been given to the structural differences of TCP and SCTP.

- In Chapter 3 the motivation for deploying a protocol translator and the background of a contemporary protocol initiation procedure are documented. Based upon that, the evolution of the multi-streaming idea with the different attempts for mapping parallel TCP connections onto an SCTP association is presented. Implementation details of the proof of concept solution as well as the measurement elements are explained in this part, including documentation of the encountered problems during the development.

- Chapter 4 includes results of the preliminary performance test and the final measurements which show the advantages achieved by using the multi-streaming feature. Proposals for improvements and problem solving approaches are also given.

The thesis assumes that the reader has profound knowledge of the transport protocol functionalities. Chapter 2 provides assistance to the reader for the necessary background.
Chapter 2

Background and Analysis of SCTP

The Stream Control Transmission Protocol (SCTP) has a lot in common with the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). General functionalities of these transport protocols are explained, and they are compared with each other in this chapter. The given descriptions offer a help to the reader, so that the presented ideas in this thesis are understandable.

Initially, the computer network concept and the corresponding layer models are presented in Section 2.1. Like TCP and UDP, SCTP belongs to the transport layer of the Internet protocol suite. As a consequence of the continuous development of protocols over time, SCTP combines functionalities of other transport layer protocols and provides advanced features. To highlight the evolution of these protocol properties, various IETF specifications, namely Request for Comments (RFC), are briefly described in Section 2.2. The content of this section, which is illustrated by timelines, is a simple summary of the necessary background and helps doing a faster look-up for RFC specifications that stand in connection with this work. A detailed description of selected transport protocol properties follows in Section 2.3. The various subsections discuss the packet structure such as the packet composition on the different layers, the connection oriented establishment/teardown procedures, the regulated data transfer by the flow/congestion control and the advanced SCTP features. For the sake of completeness, some special SCTP extensions are presented in Section 2.4.
### 2.1 Computer Networks and Layer Models

The term *network* has a widespread meaning and is also used in the computer world. A more precise definition for *computer networks* is the following:

A computer network is an association of various technical electronic components which makes it possible to communicate between individual systems. The communication is carried out via different activity procedures, called protocols, which can be structured by a layer model such as the ISO/OSI (International Organization for Standardization / Open Systems Interconnection) reference model.

Layer models are important for structuring complex computer networks like the Internet. The idea is that almost every complex network structure can be formed by combinations of simple and small structures. The different structures are interpreted as layers. The complete model is built by a hierarchical composition of different layers, where higher layers get access to the functionalities of layers lying underneath. The functionalities are defined by the provided protocol procedures. The principle task of most protocols is to submit *data* (payload). An exception exists for certain information systems whose exclusive function is to set up their own network infrastructure. The procedure for the data exchange is defined by various specified patterns, called *protocol*, used on either endpoints. The ISO/OSI reference model is only one model that defines such a protocol layer composition. Although no computer network shows the ISO/OSI model completely in practice, it is of decisive importance for network engineering. Another one, used fairly often practically, is given by the Internet protocol suite, which is shown in Figure 2.1.

Technical improvements turned the local communications into global communications and this was the dawn of the Internet, whose actual importance increases continually. This implies that in almost the same manner new protocols were deployed to satisfy the Internet-community with the additional requirements for recent applications. Probably the most used and studied network protocols are TCP and IP. However, a number of additional protocols play an important role on the Internet. The Internet itself is a heterogeneous network and consists of many subnetworks, which work quite differently. Subnetworks have only the upper protocol layers in common and the data transmission on the lower protocol layers is handled differently. The Internet is a decentralized net with different services. This global computer network shows a variety of technical differences – thus, the use of a layer model is inevitable. The enormous growth of the Internet over years and the introduced layer composition yields a vast quantity of specifications, services and communication schemes. To guarantee interoperability and scalability general specifications are necessary. The *Internet Engineering Task Force* (IETF) is an open international community concerned with the evolution of the Internet and it produces corresponding technical specification documents – *Request for Comments* (RFC). The IETF Mission Statement itself is documented in such a document – RFC 3935 [3]. Many protocols and layer models are specified by RFCs, which provide the basis for nearly all computer networks.
2.1.1 The Internet Protocol Suite

The model for the Internet protocol suite was designed in the 1970s with funding from DARPA (Defense Advanced Research Projects Agency), an agency of the United States Department of Defense and led to the foundation for ARPANET. ARPANET was the world’s first wide area network and a predecessor of the Internet. To meet the goal of interoperation across the diversity and complexity of the Internet a layer model is used which can be derived from the requirements for internet hosts, specified in RFC 1122 [4] and RFC 1123 [5]. These RFCs define the communication protocol layers as: link layer, network layer, transport layer and application layer.

![Layered model of the Internet protocol suite](image)

**Figure 2.1:** The Internet protocol suite - layer model

Applications (on top of the model) are subject to different requirements on the respective used transport protocol. The widely used transport protocols in networks are TCP, UDP and now also SCTP, due to the combination and proliferation with the Internet Protocol (IP).

UDP is used to implement a message oriented and connectionless service. The protocol is kept very simple (RFC with three pages) and therefore also shows some decisive disadvantages. Packet losses cannot be recognized and tackled by the protocol. Fault treatment routines must be implemented on the application layer, so that duplicated or lost messages can be recognized and a fault treatment procedure can take place. UDP is appropriate for real-time multimedia and other time sensitive data transmissions without a necessary receipt of every message [6].

TCP establishes a bidirectional connection after a predefined scheme between two endpoints. First the initiation procedure takes place, afterwards the data transmission happens and finally the connection is closed again. Errors are recognized and cleared automatically by the protocol on the transport layer. TCP guarantees through this to the upper layer a fault-free assignment of the data and does not work with messages, unlike UDP. Instead
of that, TCP uses a byte stream for the data transport. The combination of IP and TCP is used for the implementation of most common applications which require a reliable and ordered data transmission [7].

SCTP has inherited many properties from UDP and TCP. New functionalities were introduced, e.g. multi-homing and multi-streaming, to enlarge the field of application. The specific use of the new features is a recent scientific topic, and is a main focus of this master thesis.

2.2 The Stream Control Transmission Protocol

The Stream Control Transmission Protocol uses a potentially unreliable connectionless parcel service (e.g. IP [8]) and guarantees reliable, connection-oriented transport. SCTP first was published as RFC2960 [9] in October 2000 and then replaced by RFC4960 [10] in September 2007, again as a proposed standard. SCTP offers ordered delivery, flow control, congestion control and full-duplex data transfers, like TCP. The message oriented mode of operation and unordered delivery option for the data transmission was inherited from UDP. SCTP unites the main properties of these two protocols with new functionalities. The general feature composition is represented in Figure 2.2.

![Figure 2.2: SCTP feature combination](image-url)
A decisive question is: what led to the development of SCTP and not to an additional extension or update of existing protocols? The answer: SCTP was originally conceived for the transport of telephone signals over IP networks. Neither TCP nor UDP could fulfill the needs of telephone carriers. The development of a new transport protocol was initiated and the developers have recognized that SCTP features are generally applicable and useful in computer networks. The consequence was that SCTP turned into an IETF standards-track specification with the purpose of a more general transport protocol.

The major points that brought SCTP to an IETF standardization process were:

- the union of features from approved protocols (such as UDP and TCP)
- the integration of new features for network and application needs
- the general representation and data structure (packet composition) of the protocol for future extensions

These points represent a general guideline for deciding on pros and cons of the introduction of a new protocol. However, all upgrade possibilities for existing protocols should be exhausted first, instead of defining a completely new one. Thus not every needed feature in computer networks implies the development and standardization of a new transport protocol. It is a fact that in general new protocols won’t be easily accepted by IETF. Reasons are, that the introduction of new protocols does not usually solve all the problems and protocols also need an ongoing development. Furthermore, the introduction of protocols (like with SCTP) require more effort than the deployment of extensions, which are therefore usually preferred. The best example is TCP (since 1981 defined in RFCs), where improvements are continually integrated. A structured and detailed historical overview of many additional TCP specifications can be found in the book ”Network congestion control” [11, p.88].

2.2.1 Basics and RFC Tracks of TCP and SCTP

This master thesis does not treat all transport protocol functionalities in full detail, only the most important ones are explained. The presented background helps the reader, so that implementation ideas in Chapter 3 as well as the argumentation in Chapter 4 are understandable. Basic information of the TCP and SCTP protocols are the content of this chapter, which are helpful for a general comparison between the transport protocol structures. Reading RFCs it is the best option to obtain a detailed and complete description. Important RFCs that stand in connection with the thesis are presented in a timeline. The historical development of SCTP and TCP is depicted in the next figures (TCP RFC track in Figure 2.3 and SCTP RFC track in Figure 2.4) with the purpose of showing how the protocol standards have developed, and where many SCTP functionalities originated.
The history of the TCP standard starts in 1981, even though TCP was already mentioned in previous RFCs. TCP was specified because the UDP protocol did not fulfill the requirements for a general data transfer, since major functionalities were missing. Consequently the TCP protocol was developed and it became (and remained until nowadays) the standard protocol in combination with IP for reliable data transmission in networks.

In the course of time improvements were made and the treatment of the detected network congestion problem\(^1\) became important. Different algorithms were and are still developed since congestion control represents a large field of research. The sending behavior of TCP was also revised by different improvements that were done later on. Of special importance are the Nagle algorithm to avoid the Silly Window Syndrome\(^2\) (SWS), the window scaling option to deal with large packets in high speed networks and the calculation of the retransmission timeout\(^3\) (RTO) if packet losses occur. Again, to achieve throughput improvements, enhanced confirmation procedures were deployed, like the delayed and selective acknowledgement of packets (delayed ACK [4] and SACK [12]). A problem with introducing new extensions is that some of them need to be supported by both endpoints, which does not always work, because hosts often use different protocol versions. The consequence is that some improvements can only partly or with difficulty be introduced.

Weak points of the TCP protocol were stated approximately ten years after the standardization at special applications (e.g. telephone call control). This was the reason that a new overall plan was thought about for the transport of data. From this the Multi-Network Datagram Transmission Protocol (MDTP) developed. MDTP was the predecessor of the standardized SCTP protocol [13]. The mentioned points are shown in red in the timeline.

---

1. Low throughput because the network is overloaded due to the many data packets sent.
2. Unfavorable transmission behavior caused by small data segments.
3. Time limit within a received packet needs to be acknowledged otherwise a retransmission occurs.
The development of SCTP is presented in Figure 2.4. There are several possibilities to consolidate the basic understanding of the SCTP protocol. A general introduction to the SCTP basics at a high level is given in RFC 3286 [14], which is a guide for potential users of SCTP, who intend to use it as a general purpose transport protocol. A more detailed introduction to all technical specifications can be obtained from books (e.g. SCTP Reference Guide [13]), the actual SCTP RFC (RFC4960 [10]) and various web pages (e.g. SCTP for Beginners [15]). The RFC timeline overview in Figure 2.4 shows recommended additions that are related to the security in networks (checksum change, Ipsec, TLS,...) and the new specified protocol extensions. From the timeline it can be seen that since SCTP standardization many improvements were integrated for expanding the application field and the usability. Of special interest are the partial reliability extension (RFC 3758 [16]) and the dynamic address reconfiguration (RFC 5061 [17]). Several future specifications of extensions are still drafts (shown in red on the timeline) and these are: a general programming interface (socket API) [18], the behavior of SCTP in NAT environments [19] in combination with UDP encapsulation [20] and the Stream Reset extension [21].

RFCs of TCP and SCTP, contained in the presented timelines, are the basis for the following description of differences between both protocols.

2.3 Comparison of SCTP with TCP and UDP

In this section the different functionalities of SCTP are explained and compared with TCP and UDP. Some parts of special interest are described separately and detailed in the subsections. A compact overview of the comparison is shown at the end in Table 2.1 and summarizes the following description.
Connection oriented transport protocols (TCP and SCTP, unlike UDP) in general have three communication phases and additionally communication steering elements for the data transmission are integrated. The three communication phases are:

- connection establishment phase (confirmed service)
- data transmission phase – data transfer (confirmed or no confirmed) with various possibilities for, e.g., data packet sizing, data delivery, data packet constraints and data processing behavior
- connection release phase (confirmed service)

A comparison between the transport protocols (UDP, TCP and SCTP) based on the above mentioned phases follows.

**Connection establishment phase** SCTP and TCP are connection oriented protocols and therefore always need a setup procedure between the two endpoints. After the completion of the connection setup the data transmission is conducted. The confirmed connection setup procedure is described in detail in Section 2.3.2. UDP is a connectionless transport protocol and has no initiation procedure.

**Data transfer** The data transfer can occur in different modes: reliable, partially reliable or unreliable. A reliable data transmission guarantees the receiver to obtain the sent data completely and without errors (this is carried out by TCP and SCTP), which is realized through confirmation messages sent back by the receiver. Should a data packet loss or an error be detected, a retransmission is performed. The simplest mode is exhibited by UDP, which sends data and does not to guarantee the reception. Partial reliability is a new way how data can be sent and is a special feature introduced by SCTP. The characteristic is to send a data-set where only parts of it need a definite reception by the endpoint. The variable transmission rules are defined on sender side.

**Data packet sizing** The size of data packets on application layer plays an essential role, because it can lead to fragmentation\(^\text{4}\) or bundling\(^\text{5}\) of data blocks on the transport layer. The right behavior depends on the user application and determines if fragmentation or bundling is done by the transport protocol. A bundling of protocol data units (PDU) should be carried out if many little messages (from the view of the application) need to be sent and it helps reducing the data transmission overhead caused by protocol control information (PCI) necessarily included in every packet. According to this, SCTP packets can incorporate more than one application message. Too big application data units require fragmentation due to the limited PDU size of the underlying used link media. The consequence is that one SCTP packet may contains only

\(^{4}\) Subdivision of one packet into smaller packets.

\(^{5}\) Aggregation of smaller packets in a single one.
parts of the application message and the receiver needs to reassemble the application message by the arrived PDUs. TCP uses a byte stream for the transmission and does not care about message sizes on the application layer. The bundling or fragmentation based on byte counting is performed automatically, except for special cases, which are explicitly signaled by the application. UDP itself does not support fragmentation or bundling, but makes use of the provided fragmentation possibility of the underlying network layer.

**Data delivery**  Packets can reach the endpoint in different sequence: ordered or unordered. Ordered delivery implies that the transmission sequence of the different PDUs that form the application message is retained and a delivery occurs in an identical sequence on the receiver side. The unordered delivery of UDP and SCTP allows passing received messages to the application without regard to their arrival order. Nevertheless, a precondition for the delivery of the application message exists. SCTP allows the delivery of the received data to the upper layer only after the reassembly of possible PDU fragments, whereby the original size of the application messages is preserved, as with UDP. Unlike SCTP and UDP, TCP sends the data as a byte stream and preserves no message boundaries. The ordered transfer of TCP is strongly connected with its reliability and PDUs are delivered to the receiver in same sequence as they were sent. A special mode offers SCTP, which allows an unordered packet reception (like UDP), but with a confirmation (like TCP).

**Data packet constraints**  As already mentioned, a message cannot be always sent in a single step and the maximum PDU size varies due to the used physical media and network layer protocol (e.g. by IP version). The maximal packet size on the network layer is referred to as *maximum transfer unit* (MTU) and limits the PDU on the transport layer. TCP and SCTP use the Path MTU discovery algorithm (PMTU) for the determination of the MTU value. The algorithm is specified in RFC 1191 [22].

The fragmentation of messages may occur on the transport layer and on the network layer. A network layer fragmentation is problematic. The problem is the unreliable service provided by IP (used to transfer the transport layer packets) in combination with the possible fragmentation of IP packets (if not explicitly disabled)\(^6\). One failure scenario is that an IP packet needs to be fragmented, which contains only a part of the already fragmented transport layer PDU, and one part gets lost. In this case the complete transport layer packet must be retransmitted once more, even if the other packet fragments reach the endpoint. For this reason, fragmentation at IP layer should always be avoided as far as it is possible. The new version of the IP protocol (IPv6) does no longer allow fragmentation on the network layer.

---

\(^6\) Fragmentation of IPv4 packets should be disabled with the setting of the optional don’t fragment flag. Thus oversized packets on the network path are rejected and not fragmented. The error is reported to the sender by the Internet Control Message Protocol (ICMP).
Data processing behavior Another central point is the control of data flows. If a fast transmitter cooperates with a slow receiver, the data transmission must be sometimes interrupted. Otherwise the receiver would be cluttered with data which it could not process. The control of these interruptions is the task of flow control. Flow control is steered with confirmation messages from the receiver host for which different methods could be used. UDP gives no guarantee for the receipt of the data, and thus no confirmation procedure is used by it.

Not only hosts have to deal with packet processing, but network devices on the transmission path, too. If many TCP or SCTP packets get lost, this can be stated, since the transmitter does not receive an acknowledgment within a predefined time. The sender must assume that the packet was dropped due to overload on network routers. This means the buffer of a router is full and the transfer path is called congested, which can be dissolved only if involved transmitters reduce their transfer rate. RFC 5681 [23] defines four TCP congestion control algorithms to resolve the problem. Furthermore, congestion control is used to ensure a reasonably fair bandwidth sharing. Like TCP, SCTP also implements congestion control mechanisms. UDP does not; it can show an unfair network behavior where the consequences are a throughput reduction of other transmissions on the path and again the network can be overloaded.

Another approach is to detect network congestion by continuous router queue measurements. This helps to discover possible congestion in advance. The advantages of active queue management are discussed in RFC 2309 [24]. Mechanisms that let senders react to the onset of congestion measured in routers, is referred to as “Explicit Congestion Notification” (ECN) – described in RFC3168 [25]. ECN is supported by TCP and SCTP, but is not mandatory.

Connection release phase The existing connection is closed at the end of the data transmission. One host sends a message that the transfer comes to an end and may be closed and no further data will be submitted. TCP and SCTP have different ways to do this, which are described in Section 2.3.5.

The previous descriptions mentioned only general features of the data transmission phase. A closer look at some important differences between TCP and SCTP – for instance, the connection setup/teardown and new features – is done in the following subsections.
<table>
<thead>
<tr>
<th>Protocol Feature</th>
<th>SCTP</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection-oriented</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Half-closed connection state</td>
<td>no</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>Reliable data transfer</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Ordered data delivery</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Unordered data delivery</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Preservation of message boundaries</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Path MTU discovery</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Application PDU fragmentation</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Application PDU bundling</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Selective ACKs</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Flow control</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Congestion control</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ECN capable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Security cookie against SYN flood attacks</td>
<td>yes</td>
<td>no</td>
<td>n/a</td>
</tr>
<tr>
<td>Multi-homing</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Reachability check by heartbeat</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Multi-streaming</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Partial reliable data transfer</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2.1: Feature comparison of SCTP, TCP and UDP

2.3.1 The Packet Structure

The detailed examination of the transport protocols starts in this work with the packet structure and its composition. The data which has to be transmitted crosses different layers. It is passed from the application through different operating system functions to the network stack and finally sent. In this context the following terms are used:

- **PCI**: Protocol Control Information (protocol steering elements)
- **SDU**: Service Data Unit (packet from upper layer)
- **PDU**: Protocol Data Unit (packet/frame on actual layer)

The composition of a packet on any layer is defined as: \( PDU = PCI + SDU \).

Packets are composed and modified on its way through the layers down to the hardware (adapter, cable). The data unit (from the view of every protocol driver) coming from the layer above is described as a service data unit (SDU) of the current working service. The receiving protocol driver supplements the service data unit with its own protocol steering
elements, the protocol control information (PCI). PCI is also known as protocol header or simply called header. The SDU, together with the PCI, forms the complete data unit (still from the view of the working protocol driver) which is committed to the next layer or transmitted by the hardware device. The packet formed by SDU and PCI is described in general as protocol data unit (PDU). A corresponding packet composition of transport protocols (SCTP and TCP) is shown in Figure 2.5. The header composition shows variable sizes (shown in red in the figure) on the transport level, which depend on additional TCP options or SCTP chunks used. The PDU is passed from transport layer down to the network layer. The next layer operates in the same manner. In this way, a protocol encapsulation for the different layers is realized (like Russian nested dolls).

Two essential concepts for the assignment of data between the layers have to be distinguished: frames and packets. The difference between a packet and a frame needs to be understood as follows: all protocols of higher layers put only respective information (PCI) in front of the received SDU. Merely on the link layer, frames are formed, because all protocols of this layer (Ethernet, Token-ring,...) additionally to the PCI add a checksum at the end of the packet, which is called trailer. A data frame is a PDU at the link layer.

Figure 2.5: PDU composition on different layers

7 Specific packet building blocks with additional PCI information.
The PDU composition on the different layers leads to a reduction of the maximal possible data payload. The additional PCI data, added up to the payload, is often referred to as overhead. The overhead increases by the number of used protocols on the different layers. All PCIs contribute to the overhead and can be used to calculate the effective payload data. The frame size is determined by the used physical network media. In the next descriptions and figures it is explained how overhead and maximum payload data on the transport layer can be calculated.

![Diagram of data link header, IP header, TCP header, data, FCS, ethernet frame, maximum size 1518 bytes, 40 bytes, 48 bytes.](image)

**Figure 2.6:** Composition of TCP and SCTP packets with IPv4 in Ethernet networks

The frame size per default (assumed Ethernet\(^8\) is used) is 1518 bytes, of which 18 bytes are reserved for the frame-header and trailer. 1500 bytes remain for the data field, of which 46 bytes must be at least used, so that the minimum frame length of 64 bytes is kept to ensure the collision detection of the media access control (assumed CSMA/CD\(^9\)). Since an IP packet can have a maximum size of 64 kilobytes (specified in RFC 791 [8]), it does not always fit in a single frame. The maximum transfer unit (MTU) is the maximum packet size on the network layer which can be transferred without fragmentation. The MTU value is determined by the Path MTU discovery algorithm, specified in RFC 1191 [22].

The effective payload on the transport layer is limited furthermore by the used network layer protocol (e.g. IPv4 or IPv6). An IPv4 header usually uses 20 bytes and IPv6 [26] 40 bytes. IPv4 offers (most of the times unused) options which can prolong the header by

---

\(^8\) The Institute of Electrical and Electronics Engineers (IEEE) is an international technical professional association for the advancement of technology related to electricity. The IEEE 802 standard defines physical layer services and protocols, e.g. Ethernet – IEEE 802.3.

\(^9\) The term “Carrier Sense Multiple Access/Collision Detection (CSMA/CD)” describes an asynchronous physical media access control used by Ethernet.
up to 60 bytes. Note that the header composition on the transport layer is of variable size, too (shown in red in Figure 2.5). With regard to this it is possible to calculate the maximal transport layer PDU and thus to generate corresponding packets that can be passed to network layer without fragmentation. From this follows the maximal payload value on the transport layer, called maximum segment size (MSS). An overview of a data packet composition with TCP and SCTP, in combination with IPv4 is shown in Figure 2.6 (optimistic assumption – minimal transport and network PCI for data units). Additional information and examples about the calculation of TCP packet sizes can be found in RFC 879 [27].

Figure 2.7: General packet composition of UDP, TCP and SCTP
A data packet is generally a PDU of well-defined form and length and consists of two parts, the header and the payload. The different packet compositions are specified in the corresponding RFCs (IP - RFC791[8], TCP - RFC 793[7], UDP - RFC 768[6] and SCTP - RFC 4960[10]). Of special interest are the packet structures of the transport protocols and the header data fields, which are illustrated in figure 2.7. A comparison of the mentioned transport protocols shows that UDP, and originally also TCP, have a fixed header size. UDP packets always have a header of 8 bytes. The TCP header varies in a range of minimal 20 and maximal 60 bytes. SCTP has a fixed general header size of 12 bytes and uses 4 bytes for at least one required chunk header. Additional SCTP header information is held by SCTP chunks, too. Thus, the header composition is extremely variable in size and this makes an overall header calculation for SCTP more complicated than for TCP or UDP. To know the size of the header is of decisive importance for calculating the maximum segment size as well as the effective data payload. As remainder, the MSS is the maximal payload that can be sent with one packet and the consequence is, the bigger the segments the bigger the possible throughput.

TCP changes only the header size due to the used options. The overhead calculation is therefore quite simple, like for a SCTP packet, if only a single data chunk is used. However, the total SCTP overhead can change fundamentally due to the special packet composition. SCTP packets consist of several parts because of the used "building block" approach. The main component builds the common header with fixed size and content. In addition to that, one or more blocks can be added. The different blocks are referred to as chunks, and they have only a conditional fixed header size of 32 bit (4 bytes) with supplementing parameters called chunk values. The chunk value must always be a multiple of 32 bits (32-bit word boundary), otherwise a padding occurs until the boundary is reached. The number of values depends on the chunk type used.

The SCTP chunk overhead is computed by the chunk-type and the number of inserted chunks as shown in Figure 2.8. A calculation for a SCTP packet with a single data chunk is simple. The data chunk header has a fixed size of 16 bytes (default chunk header of 4 bytes and 12 bytes for PCI). Together with the SCTP header it yields to a total size of 28 bytes. The composition of an SCTP packet with a single data chunk looks like the first packet composition in Figure 2.8.

SCTP offers the possibility of combining several chunks in one SCTP packet. This principle is called chunk bundling. Even chunks of different types can be summarized into one packet. Another way is to put more than one data chunk in a single SCTP packet if the application only transfers small messages\(^\text{10}\). Thus PCI overhead is reduced because the common header overhead of 12 bytes is saved. In comparison with TCP such a packaging method can reduce the overhead if message based communication is mandatory and large frames on the link layer are possible. However, the chunk bundling can also lead to exactly the opposite effect if wrong message sizes are used by the user application.

\(^{10}\) Note that one chunk can contain only one message.
With chunk bundling control chunks can be inserted and transferred together with data chunks. The order at which the chunks are inserted in the SCTP package is rigorous and control chunks are always inserted with high priority. Thus all control chunk processing happens before any user data is handled. In Figure 2.8 the individual parts are graphically represented. SCTP has the peculiarity that all values which are not a multiple of 32 bits (chunk values and data) are padded up to it. The consequence is an additional overhead. The chunk length (value in the header of every chunk) contains only the amount of effective data and padded bits are not included. Again, wrong message sizes on the application layer, so that data padding is required, can yield an unfavorable composition with unnecessary overhead.

The general composition of chunks can be gathered from Figure 2.9. Every chunk has a fixed header structure that includes the chunk type, chunk flags\(^\text{11}\) and the chunk size. Two different chunks – DATA and INIT – are given as an example. These suffice for the understanding of the chunk composition possibilities. The chunk type is defined by a number (e.g. 0x00 DATA, 0x01 INIT, ...), and depending on the chunk type, specified chunk values were added. Furthermore, chunks have the possibility to add optional variable parameters. Only some chunks use this additional option, as for instance the INIT.

\(^{11}\) Flags are bit combinations in protocol headers used for state and condition signaling.
SCTP general chunk layout

<table>
<thead>
<tr>
<th>bits</th>
<th>0 - 7</th>
<th>8 - 15</th>
<th>16 - 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Chunk type</td>
<td>Chunk flags</td>
<td>Chunk length</td>
</tr>
<tr>
<td>32</td>
<td>Chunk value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCTP data chunk

<table>
<thead>
<tr>
<th>bits</th>
<th>0 - 7</th>
<th>8 - 15</th>
<th>16 - 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type = 0x00</td>
<td>Flags = 0x03</td>
<td>Chunk length</td>
</tr>
<tr>
<td>32</td>
<td>Transmission sequence number (TSN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Stream identifier (SSID)</td>
<td>Stream sequence number (SSN)</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>Payload protocol identifier (PPID)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>User data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCTP init chunk

<table>
<thead>
<tr>
<th>bits</th>
<th>0 - 7</th>
<th>8 - 15</th>
<th>16 - 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type = 0x01</td>
<td>Flags = 0x00</td>
<td>Chunk length</td>
</tr>
<tr>
<td>32</td>
<td>Initiate tag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Advertised receiver window credit (a_rwnd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>Number of outbound streams</td>
<td>Number of inbound streams</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>Initial TSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>Optional/Variable-length parameters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCTP optional/variable-length parameter

<table>
<thead>
<tr>
<th>bits</th>
<th>0 - 15</th>
<th>16 - 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Parameter type</td>
<td>Parameter length</td>
</tr>
<tr>
<td>32</td>
<td>Parameter Value</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.9:** SCTP chunk composition: DATA chunk (chunk header with fixed chunk values) and INIT chunk (chunk header with fixed chunk values) with an optional variable parameter.

The INIT is used to also inform the opposite endpoint about the supplementary addresses, which are used for multi-homing. The address format (IPv4 and IPv6) and number of addresses is variable, and this yields a different number of optional parameters with a
variable size. The new packet composition of SCTP, with variable optional parameters, is in this case required. Other protocol control information is transferred with the corresponding chunks listed in Table 2.2.

<table>
<thead>
<tr>
<th>Chunk Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>Carries user data payload</td>
</tr>
<tr>
<td>INIT</td>
<td>Initiates an SCTP association</td>
</tr>
<tr>
<td>INIT-ACK</td>
<td>Confirms the connection request</td>
</tr>
<tr>
<td>SACK</td>
<td>Acknowledges the reception of user data</td>
</tr>
<tr>
<td>HEARTBEAT</td>
<td>Keep-alive message send to idle destination addresses</td>
</tr>
<tr>
<td>HEARTBEAT-ACK</td>
<td>Response to a HEARTBEAT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Initiates an ungraceful termination of an association</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>Initiates a graceful termination of an association</td>
</tr>
<tr>
<td>SHUTDOWN-ACK</td>
<td>Confirms the termination request</td>
</tr>
<tr>
<td>ERROR</td>
<td>Reports operation errors</td>
</tr>
<tr>
<td>COOKIE-ECHO</td>
<td>Passes the state cookie of the initiation procedure</td>
</tr>
<tr>
<td>COOKIE-ACK</td>
<td>Confirms the state cookie of the initiation procedure</td>
</tr>
<tr>
<td>ECNE</td>
<td>Explicit Congestion Notification (ECN)</td>
</tr>
<tr>
<td>CWR</td>
<td>ECN Congestion window reduction</td>
</tr>
<tr>
<td>SHUTDOWN-COMPLETE</td>
<td>Final message of a graceful termination procedure</td>
</tr>
</tbody>
</table>

Table 2.2: Basic SCTP chunks

The chunk-based packet format of SCTP is especially designed to provide flexible extensibility; it allows a definition of 256 different chunks types. RFC 4960 defines 15 chunk types, which are listed with a short description in Table 2.2 (chunk type numbering starts from 0 to 14). Some additional chunks were already specified by the new extensions [16, 17], presented in Section 2.3.3.

The data transmission in SCTP is steered by chunks, chunk values and flags, whereas TCP uses only flags or the option field for it. This makes extensions for TCP difficult, because of the restricted possibilities by the small flag value. Another problem is the reaction to new extensions. While TCP implementations can behave differently (although they should normally ignore unknown extensions, in accordance with the "be liberal in what you accept“ rule), SCTP has a clear rule – simply drop unknown chunks and send an error report back. The protocol signaling elements (flags and chunks) are used for implementing the communication procedures, which are explained in the next sections.
2.3.2 The Initiation Procedure

A TCP connection or SCTP association between two endpoints gets established by a pre-defined message exchange. Instead of using a three-way-handshake for the initiation procedure like TCP, SCTP uses a four-way-handshake with a cookie-mechanism\(^{12}\). Both are shown and compared in Figure 2.10.

The change from a three-way-handshake to a four-way-handshake was made because the TCP setup procedure is very susceptible to attacks. A possible attack is called ”TCP SYN flooding”\(^{13}\), which is used for ”denial-of-service“ attacks\(^{14}\)(DoS). The problem is that a clear identification of the asking terminal node is not possible and the memory reservation is performed immediately at the connection requests. An abuse cannot be detected easily and leads to an overload of the receiver host. Possible scenarios are: all resources are used up or further connections are no more permitted because the server protects itself from overtax. Both scenarios are unthinkable for service providers and must be avoided. Therefore, a new initiation procedure (for SCTP) was developed and a cookie mechanism guarantees the authentication of the asking terminal nodes.

\(^{12}\) Transmission information, together with a valid lifetime and a signature for authentication is exchanged between the hosts; this is called a ”cookie”. Afterwards, the contained information is used to allocate the necessary memory for the transmission.

\(^{13}\) A multitude of connection setup requests is called SYN flooding. The name comes from the SYN-flag set in the TCP packet to signal the initiation request.

\(^{14}\) The servicing host gets paralyzed due to a systematic overload.
Figure 2.10 shows what happens if both connection setup procedures are simultaneously started. For the signaling TCP uses flags in the header; SCTP uses the special control chunks. The arrival of each packet is represented by a numbered event in the figure and explained in the following description:

1. It is assumed that both connection attempts start at the same time. HOST A sends out a TCP SYN (with a sequence number) and, with an occurring delay, the INIT chunk. The gap is caused by the checksum calculation and the greater effort of forming the INIT. SCTP uses a more effortful method (CRC32) in comparison to TCP (CRC16). If this calculation is carried out by the hardware (checksum offload), the time difference is insignificantly low. CRC16 checksum offload is rather supported by most hardware while this is not always the case for CRC32. The INIT includes various connection information for HOST B such as: verification tag, IP addresses, number of streams (outgoing and incoming) and with the applied extensions also a possible adaptation indication (RFC 5061 [17]). The adaptation indication is used to signal a special requested behavior and informs the distant terminal about it, such that an appropriate reaction can happen (e.g. using proper layers between SCTP and the application). The benefit is that additional data messages, needed for some kind of configuration signaling, are avoided in such a case.

2. Host B receives a TCP-SYN, increments the received sequence number, generates its own sequence number and sends a SYN-ACK back. The transmission control block\(^\text{15}\) (TCB) is generated immediately and this behavior is exploited by TCP SYN flooding attacks. Too many requests at once are problematic, as they let the host run out of memory because of the large number of created TCBs.

3. HOST B receives the SCTP INIT and sends back an INIT-ACK. The used chunk for it looks much like the INIT chunk, except that a variable length parameter, the “State cookie”, is included. No resource allocation happens this moment on HOST B for the SCTP association request. This prevents the resource attacks described before.

4. HOST A receives a SYN-ACK, increments the sequence number generated by HOST B and sends it back together with an ACK. The ACK packet may be bundled with any data. The connection on HOST A is established.

5. HOST A receives the INIT-ACK from HOST B and sends back the received “State Cookie” with a COOKIE ECHO. Additional data can be bundled with the COOKIE-ECHO. Furthermore, it is possible that HOST A receives an adaption indication from HOST B. This notification is contained in the INIT-ACK, too.

6. HOST B receives the ACK and the connection is completely established on both hosts.

\(^{15}\) The TCB is a data structure created for each connection and contains the status and operational information for the endpoint to maintain and manage the transmission.
7. HOST B receives the COOKIE-ECHO, validates the cookie and creates a TCB by the included information. From there on the association on this host is in an established state. A COOKIE-ACK is sent back and may be bundled with any DATA chunks (and/or SACK chunks).

8. HOST A receives the COOKIE-ACK, and changes to the established state. With this the SCTP association setup procedure is complete.

Note that these points describe only a standard procedure. A behavior in fault cases is here not taken into account. From the figure one can see that the TCP connection is set up faster than the SCTP association. The parallel initiation of both protocols will come to practical use in Chapter 3.

For the usage of the new SCTP features, like multi-homing and multi-streaming, packets have to carry more information at startup. The information is transferred with parameters in the INIT and INIT-ACK chunks. The main parameters of the new functions are: the multiple IP- addresses and the number of in- and outbound streams.

TCP does operate in a similar manner, and transfers additional values at startup – for instance, to inform the other host about the scale factor, if the window scaling option is enabled. After connection setup the window scale is fixed in each direction [28].

2.3.3 The Shutdown Procedure

An open connection (or association) of reliable transport protocols always needs a teardown procedure at the end. Two different methods exist for it: the “graceful shutdown” or the “abortive shutdown”. The benefit of a graceful shutdown is that any locally queued user data will be first delivered to the application and the connection will only be closed after all sent packets are acknowledged. Figure 2.11 shows a comparison of a graceful shutdown of TCP and SCTP.

Should an SCTP endpoint decide to abruptly terminate an existing SCTP association, an ABORT chunk is send to the endpoint. The association discharges any locally queued user data and the association gets closed immediately after receiving the ABORT. TCP aborts the connection in the same manner, with a packet that has the reset flag set and a corresponding sequence number. An abortive shutdown should be avoided, because of the possible data loss.

Both TCP and SCTP follow a predefined schema, which is shown in Figure 2.11. SCTP uses a three-message handshake for gracefully closing the association, unlike TCP which can use four messages for it. The occurring teardown events are numbered in the figure and described in the following:
Figure 2.11: Comparison of the termination procedure of TCP and SCTP

1. Both close attempts start at the same time. SCTP packets take longer to leave the host because of the more effortful checksum calculation (CRC32).

2. HOST B receives the FIN from HOST A and replies with an ACK packet, the acknowledgement for the closing request. At this point HOST B decides if the other direction needs to be closed, too. In such a case HOST B sends out its own closing request to HOST A. Should this not happen, the connection is said to be “half-open”. The same can also happen if one of the hosts has aborted the connection. Half-open connections are expected to be unusual, but applications that use the FTP protocol deal with this case.

3. HOST B receives the SHUTDOWN chunk and replies with a SHUTDOWN-ACK chunk. With this the host confirms that all buffered data has been sent and is acknowledged, so that the association can be closed.

4. HOST A receives the ACK from HOST B and closes the connection on this side. Usually, the host will also get the FIN from HOST B and close the connection completely; but without this the connection is in a “half-closed” state.

---

The file transfer protocol (FTP) defined in RFC 959 is a standard network protocol used to exchange and manipulate files over a TCP/IP based network.
5. HOST A receives the SHUTDOWN-ACK chunk, removes the TCB and sends out the SHUTDOWN-COMPLETE.

6. HOST A receives the FIN from HOST B and sends out the acknowledgement of the closing request of HOST B.

7. HOST A receives the SHUTDOWN-COMPLETE chunk and closes the association completely.

8. HOST B receives the ACK of the FIN and changes to the closed state. The connection is now closed from both sides.

From the figure one can see that SCTP does a faster shutdown as TCP. The reason for this is a deliberately introduced gap between the ACK and FIN from HOST B to highlight the half closed state. Usually, the TCP connection is closed from both endpoints and the ACK and FIN (steps 4 and 6) are sent together in a single packet. In this case the TCP teardown performs as fast as SCTP, even a bit faster.

SCTP has a special sort of packets that sometimes force an abort, called “out-of-the-blue” (OOTB) packet. An OOTB packet is identified if the receiver is unable to associate the peer-address contained in the packet. The arrival of OOTB packets can be an indication for some errors. The receiver therefore needs to examine the packet and then to decide what reaction should follow. In general an ABORT or ERROR chunk is sent back to the packet’s source.

2.3.4 Data Transfer - SACK, Flow and Congestion Control

The data transmission of connection-oriented protocols is usually carried out reliably and ordered. A predefined schema is used for the realization, based on the window and acknowledgement strategy of TCP described in RFC 813 [30]. A simple example flow with the corresponding acknowledgements is shown in Figure 2.12 - diagram 1. For the receipt of every data packet the host must send a confirmation back and the data packets need to arrive in the same order as they are sent out. Otherwise only the last packet, which is in sequence, gets acknowledged. SCTP offers broader possibilities for reliable data transmissions, like unordered-reliable or partial-reliable, but for the comparison in this thesis a look at the reliable ordered option, and the behavior in such cases, is sufficient. An example for the usage of the unordered reliable feature is explained in [31].

How a confirmation of the received packets occurs has changed over time and is defined in different RFCs related to TCP. The SCTP procedures based upon that and relevant for the comparison are: the delayed acknowledgment (RFC 1122 [4] and RFC 2581 [32]) and selective acknowledgment (RFC 2018 [12] and RFC 2883 [33]). The differences between these methods are that the received data is no more confirmed separately and immediately, but collectively and delayed. Especially multiple packet losses can have a catastrophic effect on
the throughput rate. The new algorithms were designed to provide a better behavior in such a case. The delayed acknowledgment is shown in Figure 2.12 - diagram 2, where one can see that the acknowledgement is only sent upon exceeding a time limit or after a predefined number of arrived packets. Selective acknowledgments (SACK) inform the sender about all segments that have arrived successfully, with an indication of missing gaps. The benefit is that the sender only needs to retransmit the packets which have actually been lost, to fill up the gaps.

Higher transmission rates can be achieved by exchanging the standard acknowledgment algorithm. The reasons for it are the diminution of the required acknowledgment packets such as the enhanced retransmission behavior if packets get lost. SCTP uses a delayed SACK per default and TCP can use SACK and delayed ACK optionally. Figure 2.13 shows the delayed SCTP SACK.

Figure 2.12: Acknowledgment procedures: (1) simple ACK and (2) delayed ACK

Figure 2.13: Delayed SACK in SCTP

It is possible that a delayed ACK is sent out before the time limit is reached. For example (done by SCTP), when two packets (not only two chunks) are received, the ac-
Acknowledgement is sent. The timer of SCTP SACK is set per default to 200 msec and the value must not be greater than 500 ms if a time limit modification is made. The emerging problem with higher delays stands in relation with the computation of the retransmission timeout (RTO) and also has an influence on the flow/congestion control. Generated delays have in general a big impact on the complete data transmission, which is shown with the practical measurements in Chapter 4.

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.

---

\[\text{Figure 2.14: Five different SACK cases of SCTP}\]

The SACK procedure of SCTP has different cases [13, p.141-142], which are graphically depicted in Figure 2.14. The description is the following:

1. HOST B receives a packet and, if it also has to transmit data to HOST A, then the SACK is sent immediately with the data. A precondition is a possible chunk bundling (SACK chunk and DATA chunk).

2. Only one packet is sent to HOST B and no others arrive at the endpoint within the timeout limit. After the predefined timeout (e.g. 200 ms) a SACK must be sent.

3. Two packets arrive to the endpoint. After the second one, the SACK is sent before the 200 ms timeout occurs.

4. Two packets arrive at the endpoint and one of them is a duplicate (recognized via its similar sequence number). HOST B applies the same rule as for two different packets, and the SACK is sent without delay.
5. A packet gets lost and the second packet (recognized via its different sequence number) reaches HOST B. The loss will be detected and the SACK is sent back without a delay. Subsequent SCTP packets containing new data will cause additional SACKs without delay until the missing data packet is retransmitted successfully.

These cases are important to know if one is interested in speeding up SCTP applications. First, it shows where some delays come from, and second the received SACKs influence the flow and congestion control. The SACK chunks exchanged between the endpoints contain the receiver windows from the hosts and are used for the congestion window computation. How flow and congestion control works, and why they must be used, is the content of the next descriptions.

**Flow control** Problems occur if the packets are sent faster than the receiver can process them; the consequence is that newly incoming packets are rejected. This results in sending repetitions through the applied retransmission by the sender, which slows down the whole data transmission and charges the sender additionally. For solving this problem, the receiver informs the transmitter through the *Sliding-window* mechanism, shown in Figure 2.15. This procedure determines how many packets are allowed to be sent. It is a technique whose primary purpose is to properly match the transmission rate of the sender to that of the receiver. During the data transfer phase, the window field is used to adjust the rate of the flow and define how many segments should be sent. This information, called *receiver window* (rwnd), is transmitted together with the acknowledgments and continuously updated. The maximal window size is in general fixed by the reserved buffer space on the receiver host. The same principle is used on both end hosts, because SCTP and TCP allow full duplex transfers.

![Flow control mechanism - sliding window](image-url)
Congestion control Congestive collapse is a condition that a network can reach when little or no useful communication is happening due to congestion. The network can get in a congested state where traffic demand is high but little bandwidth is available. Many packets get lost (dropped or rejected) by routers because their input/output queues are overloaded (no memory space for buffering left). The congestion problem is mentioned in RFC 896 [34] and contains possible problem scenarios. Van Jacobson added the first congestion control functionality to TCP [35]. With an end-to-end control mechanism the sending behavior is regulated, in order to avoid congestion, in addition to the flow control mechanism. How many packets a sender can inject into the network is calculated by the received acknowledgments. Congestion control follows the same principle as the flow control and a value, called congestion window (cwnd), limits the amount of segments which can be sent. The window size decreases if loss is detected. Otherwise two different incremental algorithms are applied to let the window grow. Every packet loss is interpreted as congestion on the network, even if there is none and a packet is lost due to other problems. The congestion calculation is also closely related to the retransmission timer as well as the amount of packets in flight.

![Figure 2.16: Standard congestion control algorithms](image)

Figure 2.16: Standard congestion control algorithms
RFC 5681 [23] specifies the standard algorithms for congestion control: slow start and congestion avoidance such as fast retransmit/fast recovery. The functionality of these algorithms is shown in Figure 2.16.

Slow Start is always used at the beginning of a transmission. The conditions are unknown and require probing the network to determine the available capacity. During slow start, the growth of the congestion window is calculated by the received acknowledgment packets (increased in general by one MSS or acknowledged bytes). Slow start stops when another state variable, the slow start threshold \( ssthresh \) is reached. From here on the congestions avoidance algorithm is applied. During congestion avoidance, the window is incremented by at most one MSS per round-trip time (RTT). Congestion avoidance continues until congestion, because of packet loss, is detected. The slow start algorithm shows an exponential growth, whereas congestion avoidance lets the window grow linear (additive increase).

Detected packet loss implies that the congestion window gets reduced. For loss detection it must be distinguished between two cases: the retransmission timer expires or the reception of duplicate ACKs (three dupACKs) is signaled. In both cases the sender immediately set ssthresh to one half (multiple decrease) of the current congestion window size, but not less than two MSS. Furthermore, the two cases are handled in different ways. If congestion was indicated by a timeout, the congestion window is reset to one segment, which automatically puts the sender back into Slow Start mode. The other case is that congestion was indicated by duplicate ACKs, and then the fast retransmit and fast recovery algorithms are invoked.

Fast retransmit starts to “inflate” the congestion window artificially (set cwnd to ssthresh plus \( 3 \times MSS \)) and retransmits the lost segment. For every following dupACK the cwnd is incremented by one MSS and then a segment is transmitted. On arrival of the acknowledgment of new data the congestion window is set to the ssthresh-value – congestion window “deflating” – and congestion avoidance is performed again instead of falling back into slow start (therefore called fast recovery). It is possible, that the ACK for new data does not arrive within the retransmission timeout. In such a case the sender changes from fast retransmit/fast recovery into slow start.

The congestion control behavior is related to the used acknowledgement procedures. Different effects of congestion control caused by the acknowledgement procedures are practically shown in Chapter 4. As one may expect, throughput rates vary with the congestion control behavior.

\[ \text{18 The initial value of ssthresh is by many TCP versions set to the size of the receiver window.} \]
2.3.5 Features of SCTP - Multi-homing and Multi-streaming

SCTP has advanced functionalities in comparison to TCP: multi-homing with an integrated heartbeat mechanism and multi-streaming.

The multi-homing feature of SCTP allows assigning more than one IP address to an SCTP endpoint for redundancy purposes. Through this, more end-to-end paths can be used and the availability of them is checked periodically with so-called heartbeat messages. The heartbeat-mechanism is implemented with appropriate chunks.

The other key feature is multi-streaming, which defines streams within the association. A stream is a unidirectional channel, where the transmission sequence is handled independently for each stream. Thus, a multiplexing of ordered and unordered delivery within the same association is possible. Although the send behavior is handled by every stream separately, some restrictions exist on the receiver side.

Detailed descriptions of both features are presented in the following. They explain the key details, which are important for further steps in the next chapters.

Multi-homing includes that hosts have multiple network interfaces with different IP addresses (IPv4 or IPv6 unicast addresses) on all active interfaces. The SCTP port (TCP has its own port numbers) number is shared with the addresses and defines together with them the association. One of these addresses is selected as the primary path and the others become alternate paths. Generally, the data is sent to the primary path. Only if this path is not available anymore, the data is sent over an alternate path. Path errors are recognized by both endpoints (combination of RTO and HEARTBEAT-chunk) and a switch-over to an alternative path happens automatically, if within a predefined time the retransmission is not carried out successfully. Should the primary path later become available again, a changeover back to this path is done.

The multi-homing property allows a robust and more reliable data transmission within predefined time-limits. The time-limit can be adjusted by several parameters, like the RTO timer values, the heartbeat timer interval and the retransmission threshold. However, the optimization of multi-homing is not subject of this work. Information on this topic is contained in [36] and another detailed analysis concerning path changes with multi-homing is provided in [37]. Note that the default settings of RTO values and maximum path retransmission, which are recommended by RFC 4960, sometimes need to be adjusted. The exponential backoff\(^{19}\) with the default settings implies that a path change after approximately one minute \((1 + 2 + 4 + 8 + 16 + 32\) seconds) occurs [2].

\(^{19}\)Exponential backoff is an algorithm in networks that multiplicatively decreases the sending rate to prevent network overload. Therefore, retransmission timer multiplicatively increase timeout values.
Figure 2.17: Multi-homing – hosts use multiple IP addresses on the Internet and build redundancy paths for SCTP associations

Figure 2.17 shows two multi-homed (host A and host B) and one single-homed host (host C). TCP was not designed to use more than one address per connection. Therefore, only a combination of pairs is possible for TCP connections, while for SCTP associations all possible address combinations are allowed. The condition for a valid SCTP association is that all addresses belonging to the association share the same SCTP port and are not allowed to be occupied twice.

Multi-homing requires that every end-point has a separate parameter list for all configured paths. The values for the MTU as well as the congestion and the receiver windows are held separately for every path of the association. Thus a fair sharing of the network resources to other transport protocols, like TCP, is guaranteed.

SCTP does not yet support “Concurrent Multipath Transfer” (CMT). CMT is the concurrent transfer of data over multi-homed source and destination hosts, in order to increase the throughput by using multiple end-to-end paths. Additional information about CMT is contained in [38].

**Multi-streaming** allows data to be sent parallel on independent streams, so that data loss in one stream does not influence the send behavior in other streams. Each data packet uses two different sequence numbers. The *transmission sequence number* (TSN) is used for the detection of message losses, in order to guarantee a reliable data transmission. The *stream sequence number* (SSN) together with the *stream identifier* (SID) determines the sequence of delivery on the receiver host, where the SSN is not set for unordered transmissions. A more complex data packet composition than for TCP is necessary for the implementation of the multiple stream concept. SCTP’s data chunk format (shown in Figure 2.9) has explicitly been designed for it. The TSN, SID and SSN are permanent parameter values of data chunks.
Multi-streaming was developed with specific background knowledge about existing problem cases. One for example is that several files are sent in sequence over a single TCP connection, where it may happen that subsequent file transmissions get blocked, because a part of a previously sent file was lost and a retransmission is required. The arising problem is called "head-of-line blocking" (HOL). This unfavorable behavior is caused by the ordering mechanism of a reliable transport protocol and adds unnecessary delay. The same applies when an association of SCTP will be used without multi-streaming. In the following, this problem case and its solution are explained in detail.

![Diagram: Single TCP and parallel TCP connection vs. SCTP multi-streaming](Image)

**Figure 2.18:** Single TCP and parallel TCP connection vs. SCTP multi-streaming

It will be assumed that various files or messages need to be sent and they should be used immediately after receiving. The ordered delivery (FIFO principle) does not allow sending a further packet before all previous packets are transmitted successfully. Therefore, different file parts cannot be sent outside the sequence if only a single connection is used. If during the transmission a packet is lost, all other packets in the queue are blocked until the lost part is finally transmitted. The described case is illustrated in Figure 2.18, where three files are transferred with different implementations, namely with a single TCP connection, with parallel TCP connections and with a SCTP multi-streaming association. It is assumed that all transmissions treat the same error case and the first part of the second file (shown in red in the figure) gets lost during the transmission. Within the single TCP connection all subsequent parts of the files are blocked in the sending queue, while the transmissions carried out with multi-streaming or parallel TCP connections show another behavior. In
In this case, only the transfer of subsequent parts of the second file will be blocked. The other file-parts can still be transmitted. The ordered delivery restriction only applies to the belonging connection or association stream. A probable consequence is that the first and third file reach the endpoint before the transfer of the second file finishes and the application can accomplish further operations with the already arrived files. Note that the transmission progress in such a case depends on the implementation and operating system scheduling, too.

The HOL-blocking is avoided by parallel transmission, but new side effects arise with this approach. The congestion control of multiple connections is adverse (more aggressive than a single connection), because for every connection the congestion control is applied and this leads to a bandwidth sharing proportional to the number of used connections. The consequence is an increased throughput rate that can be achieved, but with the disadvantage that network behavior is no longer fair. Parallel data transmission takes place in many everyday applications. One example is a web browser, where the individual data, like pictures and text, can thus be transmitted faster. For distributed systems and cloud applications a parallel data transfer is mandatory. The network behavior changes fundamentally through parallel transmissions and could also be disturbed through such applications. Multi-streaming offers an alternative solution and could be helpful to achieve a beneficial behavior. However, some important things must be considered.

![Diagram of data transmission with multi-streaming and shared transmission control values](image)

Figure 2.19: Data transmission with multi-streaming and shared transmission control values

Multi-streaming enables an independent parallel transfer with a single association. Therefore, a few values are shared by all streams. Some important ones are the congestion window and the receiver window (shown in Figure 2.19). These values are used and modified by all stream transfers. As already mentioned, the multi-homing feature needs separate congestion windows for each path, by which the transfer rates of all streams are influenced, too.
If multi-streaming is used, special attention must be given to the behavior of the receive buffer. Figure 2.20 depicts in detail how a data transfer with multi-streaming is handled by the Linux network stack implementation. For the following description is assumed that, again, three files of different size are transferred with multi-streaming (one file per stream) and each file is interpreted as a single message. The consequence is that the messages are fragmented at transport layer. The network stack generates the corresponding data chunks, together with a transmission sequence number (TSN) and stream sequence number (SSN) for each stream. The sequence in which different files are passed from the application to the network stack and the chunks are put to the send buffer depends on the application and protocol implementation. Furthermore, this process is influenced by the operating system scheduling. After the chunks have been passed to the send buffer, they are successively sent and processed by the receiver host.

![Diagram of an SCTP association with multi-streaming]

Figure 2.20: Detailed view of an SCTP association with multi-streaming

Generally, only complete messages (i.e. after reassembly) are put in the receiver buffer, which then are ready to be read from the application. However, in some cases a partial delivery of messages is required.

Note that a message fragmentation can occur at the application layer and/or transport layer. The fragmentation at the transport layer has a big influence on the receiver behavior, especially in connection with multi-streaming. Most side-effects caused by the transport layer fragmentation are avoidable through an appropriate application level fragmentation e.g. by splitting up big files into MSS-sized messages.

Parts of an incomplete message may be read by the application from the receive buffer.
A partial delivery must be performed if the received message is bigger than the available buffer. Otherwise all transmissions (from different streams, too) are stopped and the whole application is stuck. With partial delivery, buffer space is freed and this problem is solved. For partial message delivery a limit value can be specified, which has to be exceeded, before this operation is carried out. Setting this to low values will cause partial deliveries to happen often. Applications can use this possibility deliberately for receiving only parts of messages, which in some cases is beneficial. The reason for an earlier delivery of a message, until its completeness, is application dependent and steered by the “partial delivery point”.

In general SCTP messages are passed to the application within a single read operation, so that the original message size is preserved. Delivery of partially received messages breaks this rule and a complex task must be carried out, for which two different implementations are possible.

One is to prevent the interleaving of any messages; if a partial delivery begins no other messages will be delivered to the application. Other messages, which could be delivered because they would be complete, are blocked. In this case, the HOL-problem is solved by multi-streaming and the data transfer, from sender to receiver, happens faster. However, another problem exists, because the available data of other streams is not delivered faster to the application – the restriction is now in the receive buffer. In order to solve this issue another method must be used.

The alternative is to allow complete interleaving of messages. This requires the receiving application carefully to observe the stream number at each read operation performed. By enabling this option a partial message may be delivered for stream ”X” and the next read operation at the application layer may return a message from stream ”Y”. With this, the above mentioned blocking case is avoided and a faster message processing is possible. Both described implementations are part of the currently proposed socket API draft [18].

A difference exists between TCP’s and SCTP’s end of file (EOF) signaling method. TCP signals the end with a specified EOF character, because of the byte-oriented approach. SCTP conserves message boundaries, and appropriate flags in the data chunk are set to identify the different message parts if they were fragmented. Alternatively, this can also be controlled by the application. However, this possibility should be used with caution, because of the many side-effects which must be considered.

The data transfer with multi-streaming in comparison with TCP is the central point of this work, used for the implemented approach in Chapter 4.
2.4 SCTP Extensions

This section has two intentions: first to exemplify the possible extensibility of SCTP with new chunks. Second, to present relevant extension related to the thesis, of which some are still in draft.

Particular attention was paid to the standardization of SCTP, namely to define a protocol which is easily expandable. The result was the “building-block“ approach, where packets are composed of different chunks. The SCTP standard describes 15 specified chunk types, which are identifiable through a unique number (8 bit field). The remaining values could be used for extensions, except for some that are already reserved by the IETF. Two extensions were already specified in an RFC and make use of new chunk types: the Partial Reliability Extension and the Dynamic Address Reconfiguration Extension.

2.4.1 Partial Reliability Extension

RFC 3758 [16] specifies an extension of SCTP, namely the partial reliability (PR-SCTP) option. The extension defines a new parameter for INIT and INIT-ACK, and a new FORWARD-TSN chunk type. Receiving a FORWARD-TSN permits the SCTP receiver to neglect one or more data chunks by simply advancing the TSN number to the communicated new TSN value.

PR-SCTP makes it possible to provide for the transmitted data a lifetime and after expiry a transmission of the corresponding data chunk may not be carried out in any case. It is up to the transmitter to determine the send behavior (ordered or unordered, reliable or unreliable) and the lifetime of a message. Application fields are still investigated. One, for example, is the streaming of multimedia content [39].

2.4.2 Dynamic Address Reconfiguration Extension

Dynamic Address Reconfiguration, specified in RFC 5061 [17], enables SCTP to change association values (addresses) at runtime. For the implementation two new chunks (ASCONF, ASCONF-ACK) with seven new parameters were introduced.

Hosts may have multiple points of attachment to a network and many operating systems allow the dynamic addition and deletion of network interfaces. As SCTP was designed for multi-homing, the used interfaces (addresses) had to be defined at the connection initiation. The extension fixes this lack and enables associations to add and delete IP address at runtime. Furthermore, the primary path can be changed, too. These modifications are useful in many cases, because it better reflects the host and network changes that may occur and which need a reconfiguration.

Moreover, an additional functionality was added by this RFC. A 32-bit integer value called adaptation indication. The value can be exchanged at startup (mentioned in Section 2.3.2) and is useful for signaling of a required adaptation.
2.4.3 Extensions in Draft

Some extensions which are just in development are briefly described, since they stand in close connection with this work. These drafts are:

- SCTP Socket API [18]
- Stream Reset Extension [21]
- SCTP encapsulation with UDP [20]
- SCTP and NAT behavior [19]

The socket API (Berkeley socket interface) has provided a standard mapping of the Internet protocol suite for many operating systems. In [18], an informational description explains the new interfaces of the socket API. The descriptions of the cited draft provide a basis for implementing applications with the new SCTP features. The API has been specifically designed for portability, so that applications can be migrated to SCTP with a few changes. However, the SCTP API is still in development, like the SCTP network stack implementations [18].

Some SCTP application programmers requested the ability to start the sequence numbering of different streams anew. This allows the applications to track the message flow on reused streams via the stream sequence number. Without this feature, transferring a message again on a previously used stream would result in sequence numbers larger than expected. For this new chunk types were introduced and they are used to signal the stream reset event [21].

Existing SCTP applications with multi-homing mechanisms pose unique challenges in NAT environments. The occurring issues are documented in [19] as well as in [40]. The end-to-end path between hosts may consist of one or more NATs and/or firewalls without rules for handling SCTP traffic. As long as middleboxes do not support SCTP, it seems that UDP encapsulation is the only reasonable solution; it is described in [20]. Another approach is drafted in [19] where specific SCTP chunks and procedures for NATs are described to implement similar features as the ones provided by TCP NAPT. An approach for SCTP NAT is being developed by the “Center for Advanced Internet Architectures” (CAIA). The “SCTP over NAT Adaption” (SONATA) project provides a released SCTP NAT implementation for the FreeBSD 7 platform and uses an existing NAT framework (ipfw) for it. However, until now, there is no NAT implementation available for Linux systems.

---

22 Network Address Translation (NAT) is the collective term for procedures that automatically replace address information in data packets to connect different networks.

23 Generated protocol packets are interpreted as single PDUs and transmitted by another protocol. In most cases UDP is used, because it is almost always supported by middleboxes and generates little overhead.

24 Network Address Port Translation (NAPT) is a method of NAT with a port number translation. When NAPT is used the NAT router assigns a private IP address on the LAN side and a public one on the WAN side.
All mentioned extensions and drafts show that SCTP is still under development and many open issues need to be clarified.

2.5 Summary and Conclusion

This chapter gave an overview of the history and the current SCTP development state. The information provide an insight into the necessary background for the next chapters. Therefore, the packet structure was presented and the new features of SCTP were discussed in detail. Furthermore, a closer look was taken at major differences between TCP and SCTP; they must be considered if one or the other protocol should be used. The conclusion is that SCTP shows a lot of benefits over TCP and promises to be an attractive protocol for many application scenarios.

SCTP seems to be a potential candidate for replacing TCP as the dominant transport protocol in today’s Internet. In this context the theoretical comparison of SCTP and TCP was helpful to identify shortcomings of SCTP that make such a protocol exchange difficult. Different approaches, in order to deploy SCTP alongside TCP, with the intention to benefit from the new SCTP features, are presented in the next chapter.
Chapter 3

Motivation, Design and Implementation

SCTP promises to be suitable for various network application scenarios, but there are nevertheless difficulties of succeeding with this protocol on the Internet. One of the biggest problems is the supremacy of the TCP/IP protocol suite and the related effort for enabling SCTP support through reprogramming. A general “protocol translation service” seems to be a better solution than rewriting existing applications, in order to promote the deployment of SCTP alongside other transport protocols.

In Section 3.1, already available “translator software” is presented, which enables TCP applications to support SCTP. The reason for using SCTP over TCP is mainly due to benefits achievable with the advanced SCTP features. The motivations for using multi-homing as well as multi-streaming are presented in Section 3.2.

Analyzing the available “translator software” has shown that the focus was laid on benefiting from SCTP’s multi-homing. Another valuable SCTP feature, namely multi-streaming, was not taken into account. In order to fill this gap, either a modification of the existing software or the design of a new approach is required.

In a first step the applicability of multi-streaming was studied in detail; the background for this is presented in Section 3.3. The analysis shows that multi-streaming demands a special implementation style for parallel transmissions. Based on this, the adaptation of existing tools as well as some new drafts for protocol translation with integrated multi-streaming are discussed in Section 3.4. However, a suitable solution to extend existing software could not be found, and a new approach – the mapping of TCP connections onto a SCTP association with multi-streaming – was drafted. The detailed step-by-step description in Section 3.5, beginning with the basic application of SCTP’s multi-streaming up to the implementation details for the gateway approach, builds the cornerstone for the final solution, which will be used to show that it is possible to attain an enhanced data transmission.
3.1 Preliminary Investigation

The most remarkable advantages of SCTP compared to other transport protocols are the advanced features like multi-homing and multi-streaming, which seem to be predestined for enhanced Internet services. Nevertheless, SCTP is prevalently used for SS7 signaling\(^1\) in modern IP based telephone networks (for which it was originally designed) and has not yet been widely deployed for standard applications that use the Internet. The analysis in [41] as well as the conducted Internet research confirm what was said before.

An investigation about available SCTP applications has been carried out, consisting of two parts: the first part, explained in Section 3.1.1, analyses the availability of applications that are SCTP-enabled. The second part discusses approaches for increasing the deployment of SCTP – so-called “protocol translators”– which add SCTP support to existing applications, without having to reprogram the application itself. Two such translator tools have been reviewed in detail to obtain an overview of their capabilities and features. The differences that exist between them are described in Section 3.1.2.

3.1.1 Available SCTP Applications

Possible SCTP application scenarios are described in [13], and some of the presented ideas there, like a web browser and an FTP client, are already implemented (with the corresponding server for both, of course). Another web application discovered on the Internet is an online poker game which has SCTP support built-in. Furthermore, there are various network applications listed on [42] which use SCTP for special network purposes, different from standard applications. However, only the above-mentioned applications could be found with an Internet search.

This is the short list of web applications with SCTP support:

- FTP client/server with SCTP (native SCTP) [43]
- Apache - web server (adapted for SCTP) [44]
- Mozilla Firefox - web browser (adapted for SCTP) [44]
- Thpoker - gaming software (TCP and native SCTP) [45]

The list shows that only a handful of standard applications have been adapted; they might not suffice to promote the Internet-wide deployment of SCTP. This raises the question: what are the reasons why SCTP is not used much, given that it has improvements over other transport protocols like TCP?

\(^1\) SS7 (Signaling System 7) is a set of telephony signaling protocols, which are standard for control signaling in the PSTN (Public Switched Telephone Network).
While SCTP provides basic TCP-like services in addition to its advanced features, SCTP is not directly interoperable with TCP and all implementations need to be rewritten in an appropriate way to support both protocols. The mere protocol change is not always beneficial, because application requirements can differ. Hence reprogramming and/or dual-stacking\(^2\) all applications seems an unreasonable effort, even if the SCTP socket API [18] has been designed to simplify the transition from TCP to SCTP.

An alternative to reprogramming is to carry out the protocol translation in a more general way. The required functionality is a transparent translation service which adds SCTP support to the original application and allows an optional fall-back, should the SCTP service not be available or should it be more efficient to use the original protocol. Two tools, which implement such a translation method, were found as part of the investigation.

### 3.1.2 TCP to SCTP Protocol Translator Software

The need for protocol translators has been recognized early, and two tools are available that enable TCP applications to support SCTP:

- **withsctp [1]** – user space implementation in Linux
- **TCP-to-SCTP translation shim layer [2]** – kernel space implementation in FreeBSD

This software enables the protocol support of SCTP and three special properties of it become thereby introduced: security at startup, multi-homing and a message based communication instead of byte streaming. Analyzing these tools shows that both use a simple one-to-one mapping strategy for the translation: every TCP connection corresponds to an SCTP association. As a result of this approach only one of SCTP’s more valuable features – multi-homing – gets enabled, while its multi-streaming property is neglected. A closer look at the functionality of this software was taken for two reasons: first, to identify possible attachment points on the operating system levels for protocol translations, and second, to discover decisive differences that exist between the two tools.

One difference is the application spectrum, because the tools have been developed for different operating systems. The *withsctp* tool is a user space terminal application that works under Linux (included in the LKSCTP-tool package) and the *shim layer* is a kernel module for the FreeBSD [46] operating system. Hence, they attach on different operating system levels as shown in Figure 3.1.

In terms of the technical realization, the *withsctp* implementation is less complex than the *shim layer*. This is related to the operating system address space they are operating in (kernel space vs. user space\(^3\)). Therefore, *withsctp* can be applied by a user without kernel programming knowledge, making it easier to port to different Linux systems as well. As the

---

\(^2\) “Dual-stacking” is the support of both transport protocols (SCTP and the original protocol), such that a backward compatibility to the original protocol is preserved.

\(^3\) The differences between these operational modes are explained in more detail in Section 3.3.1.
**Figure 3.1:** General overview of different operating system levels and where the mapping tools are attached (1 = withsctp, 2 = shim layer)

*shim layer* comes as a kernel module, which is not integrated in the FreeBSD distribution, a user needs to compile and insert the module. For these steps, quite some knowledge of the operating system is required. Portability is an issue as well as it is the kernel which makes UNIX based operating systems different. The pros and cons of the individual implementation possibilities are discussed in [47] and in [2].

Furthermore, the activation of the protocol translation is accomplished in a different way. The *withsctp* software enables the SCTP support for each application separately, by starting the translator as terminal-command together with the application, for which the corresponding program call is passed as parameter to the translator software. The *shim layer*, however, requires a configuration by so-called “shim-rules”. The rules consist of an IP address, a subnet mask and a port. Applying them specifies and activates the corresponding TCP service for supporting SCTP additionally.

Another characteristic that distinguishes the tools is the implementation of dual-stacking (using TCP and SCTP) if an application makes use of the protocol translation. A finer difference is that, unlike *withsctp*, the *shim layer* allows an optional handling of TCP and SCTP within the same program instance. The application of *withsctp* only allows the exchange of the TCP protocol with SCTP. In this case a decision must be taken by the user about which one of the two protocols is preferred or, alternatively the applications must be executed twice (two instances of the same application; one for TCP and one for SCTP).

SCTP’s multi-homing feature is enabled by both, but not in a really convenient way. Either all available IP addresses of the endpoints are assigned to the multi-homed association or only a single one can be used. The possibility to specify a few interfaces explicitly or to change addresses during a transmission (the use of the dynamic address reconfiguration extension presented in Chapter 2 - Section 2.4.2) is not included.
The state of the art shows that existing SCTP translator software introduces only some features of SCTP and improvements need to be integrated. The demand for a new approach with multi-streaming exists and is also mentioned in [2]. However, before a new approach will be drafted, the motivations for using multi-streaming have to be specified.

3.2 Motivation

The main problem of any new network protocol is the issue of deployment. The above presented software was developed with the goal of porting TCP application in a simple way to SCTP and to benefit from the advanced features. Since TCP is the most frequently used transport protocol in the Internet, its replacement will be of most interest. SCTP has at least two distinct features which lead to an improved functionality compared to TCP: multi-homing and multi-streaming.

The multi-homing support is only one reason for replacing TCP connections with an SCTP association. The benefits thereby achievable are the ability to provide fault tolerance to applications and a faster transmission via the capability of concurrent multipath transfers (CMT). Since CMT is not yet supported the benefits of multi-homing are little. One problem of this feature is the small numbers of applications that can make a proper use from the higher reliability. This is due to the fact that a standard Internet user in general does not have several Internet connections (and thus more IP addresses) than for example, some companies have. Furthermore, multi-homing is still affected by issues, like SCTP’s Network Address Translation problem [40], and is either therefore sometimes avoided or simply impossible to apply.

Another quality in networks is important and requested from all users: fast data transmissions. SCTP’s multi-streaming promises to provide faster data transfers as TCP (explained in Chapter 2). The idea (pursued in this thesis) is that a mapping of all TCP connections using the same end-to-end path onto a single SCTP association with multi-streaming will be beneficial. According to this, the possibilities of multi-streaming need to be studied in detail and then applied in an appropriate manner. The motivation for a modified data transmission is linked to the congestion control behavior in computer networks and aims at two goals:

The first goal is to achieve a new congestion control behavior for parallel transmissions. The congestion control should be applied only once on each end-to-end path, instead of having one congestion control instance for each TCP connection. A more complex but similar congestion control management is specified in RFC 3124 [50]. The motivations mentioned therein are valid for this thesis, too.

The second goal is to attain a faster data transfer, which is theoretically possible, should a data transmission already happen or have previously happened on the same path.

---

4 This is assumed by interpreting statistical reports about Internet traffic in [48, 49].
The reason for the speedup is a larger congestion window value, which implies that more packets can be injected into the network at the transmission startup phase. Consequently, faster data transmissions are possible, akin to HTTP’s persistent connections\(^5\).

To summarize, it is of general interest to develop a “protocol translation service” since the following improvements are possible due to SCTP-specific properties:

- higher path failure resistance by multi-homing. Further improvements are possible in future by CMT.
- advanced network behavior by multi-streaming for several simultaneous data transfers on the same end-to-end path. Congestion control information can be shared and throughput improvements are possible.

Important for a protocol translation service is a possibility to fall back to the original protocol, should either an end host not provide the protocol translation service or should the SCTP association setup fail due to other circumstances. Regarding all of that, clear rules for the further work and the main intentions of the thesis are specified in the next section.

### 3.2.1 Intentions and Related Questions

The goals of this thesis are to find a suitable approach for a transparent connection mapping with multi-streaming (TCP to SCTP with integrated backward compatibility) and to implement an appropriate software element that proves its applicability in practice. The intention is to achieve a new congestion control behavior for parallel data transmissions on the same end-to-end path and consequently a faster data transfer under certain conditions. This is depicted in Figure 3.2.

The background to this is the following: usually, every new TCP connection starts in slow start mode, even if there are parallel transmissions on the same path. Thus every TCP transmission shows a poor performance in the startup phase. To solve this issue many improvements were already developed. One of them is described in RFC 2140 [51], where transmission parameters of an earlier connection are used to initialize some parameters of a new connection. Another one is HTTP’s keep-alive mechanism. The general idea of all of them is to use more appropriate congestion window values at startup. The reuse of connections or sharing of connection information for further data transmissions makes a faster data transfer possible, because subsequent transmissions start with a higher congestion window value and hence a higher transmission rate at startup is attainable. The same principles can be applied on SCTP associations due to multi-streaming.

TCP applications need several connections for parallel transmissions, in order to avoid the HOL-blocking\(^6\). The side-effect of parallel connections is a working congestion control for each connection on the same end-to-end path and this is not always beneficial. The data

\(^5\) HTTP applications reuse TCP connections for subsequent data transfers.

\(^6\) The “Head-of-line” blocking for exactly this case was exemplified in Chapter 2.
flows compete with each other for network resources and, as a result, they can be overly aggressive against a single flow on the network. A solution for this problem is the congestion manager, which is specified in RFC 3124 [50]. This document describes a module that enables multiple concurrent flows on the same end-to-end path to share the same congestion information. SCTP’s multi-streaming seems to be well designed to implement a similar behavior, but in an easier way. The congestion control sharing for parallel transmissions is simply applicable through an appropriate implementation with multi-streaming. However, some important questions regarding the implementation have appeared, and they are related to SCTP’s performance.

1. Is SCTP in general (without multi-streaming and CMT) able to do a data transfer as fast as TCP?

2. What are the important tuning parameters for SCTP, and where (on the operating system levels) are they positioned?

If the use of SCTP would result in slower data transfers compared to the use of TCP, users will refrain from using the envisioned protocol translator. The presented questions can be only answered by practical measurements. Already conducted measurements [52, 53, 54] confirm that in some situations, SCTP shows a faster data transfer, but not always. Moreover, the performance results presented in the cited documents are not up-to-date anymore, as they depend on the performance of the SCTP stack implementation, which is under active development and improves with every new kernel version. The measurement results furthermore reflect only special application cases and are not universally valid.
As data transmissions are carried out in several ways, this must be taken into account for the intended migration from parallel TCP transfers to a parallel transmission with SCTP’s multi-streaming. The different data transfer modes have a big impact on the design of the intended translator software. An analysis of different possibilities for data transmissions is necessary in order to figure out an appropriate implementation style.

3.3 The First Steps

Different data transfer modes are described in Section 3.3.1 from an application level view. In Section 3.3.2, it is explained how connection setup works for dual protocol support, which is essential for protocol translation as already stated. Finally, in Section 3.3.3, the implementation of parallel data transfer with SCTP multi-streaming is explained. In this context a special required implementation style is presented, for avoiding possible blocking situations, which are otherwise introduced by multi-streaming.

3.3.1 A Simple Data Transfer

As the data forwarding procedure of the network stack (including packet composition and transmission details) has already been described in Chapter 2, the following descriptions are now focusing on the way the application layer interacts with the network stack in terms of data transmission. Depending on the amount of files and the amount of processes/threads used, a data transfer from host A to host B can be implemented in different ways as shown in Figure 3.3.

![Figure 3.3: Thought process for considering various data transmission modes](image)

A single file is usually transferred in serial mode, sending every piece of the file over a single TCP connection or SCTP association. On host A, the application uses a socket as the interface between the application and the operating system for the data forwarding. The operating system then forwards the data received through the socket to host B which,
also using a socket, passes the data from the network stack up to the application, taking into account the chosen delivery method\(^7\).

Things get complicated with modern multi-processor machines and multi-threading capable operating systems. The single pieces of a file could be passed from the application with several processes/threads through multiple sockets to the network layer, which creates multiple connections to transfer the data in parallel. Such an approach is standard for Grid or Cloud Computing and in parallel systems, but it is used for simple data transfers as well, in order to achieve higher transfer rates.

The operating system provides to the applications a so-called “user level abstraction”, by which access to the hardware with different levels of access to resources is meant. The different access levels are regulated in Linux systems by two distinct modes of operation\(^8\). In *kernel mode*, also referred to as system mode, it is possible to execute any instruction and access all memory addresses. For everything other than operating system specific functions, the *user mode* is used – a non-privileged mode for user programs (applications) with restricted access rights. Thus, all user applications must request to use kernel functions for accessing the hardware, by means of a system call\(^9\) in order to perform privileged instructions. The same principle is applied by the socket API operations because of input/output operations on the network interface.

Sockets are the interfaces which are used by an application to access the network device. The user level abstraction implies that memory segments of sockets, which for the above-mentioned reasons cannot be accessed directly by the user application, are referenced by numbers. Merely the operating system in kernel mode identifies through the socket number a corresponding connection and has access to the real memory segments because of access rights policies and memory management. Due to the fact that not every socket maps to a different connection, it is impossible to derive a one-to-one relation on user space between a socket number and a connection. Processes and threads may inherit or share sockets, so that a single connection is referred to by multiple sockets (connection sharing). The socket sharing (or socket exchange), required for connection sharing between unrelated processes, is only possible with a special implementation, which is exemplified in [55] (UNIX domain sockets). These details are important to know for the different approaches (presented in Section 3.4), because they have a big impact on the design.

The transmission of several files is in general carried out in two ways. One is the serial transfer of many files in sequence on a single connection/association (mostly avoided by TCP because of HOL-blocking). The second one is a parallel transfer with different connections/associations for each file or file parts. Note that transmission modes influence the congestion control behavior differently, and several ways for implementing parallel transfers exist.

---

\(^7\) The delivery options as well as related restrictions were presented in Chapter 2.

\(^8\) Like the mode also the memory access has to be distinguished; called kernel space and user space.

\(^9\) System calls are the functions that are called to perform the services of the operating system.
3.3.2 Dual Setup of TCP and SCTP

One of the various types of transfer is in the following modified for a dual support of TCP and SCTP. For simplifying matters a serial data transfer (one file and one process/thread) is being used and the necessary adaptations to the application are shown in Figure 3.4.

As mentioned in the previous section, an application has access to the network interface through the socket API and uses a corresponding socket for each connection/association. A modification of standard TCP applications for additionally supporting SCTP on either side (receiver and sender) requires the creation of an alternative socket descriptor. The intention is to use SCTP if available while falling back to TCP otherwise, but not to switch between different protocols during a transmission. A current Internet-Draft [56] describes the introduction of new technologies to HTTP and contains information on how to create a parallel initiation procedure. This proposal appears to be generally applicable for connection setup of TCP and SCTP and is also applicable in this case. The descriptions in Chapter 2 are helpful to figure out how a possible parallel startup of TCP and SCTP would proceed.

With the above-mentioned adjustments, an application is ready for a data transfer and may choose between the transport protocol used (TCP or SCTP). The benefit of the dual method is the possibility to fall back to the original protocol, in this case TCP, if one of the endpoints should not support SCTP. This guarantees a full interoperability of SCTP and TCP services within the same program instance.

The presented example forms the basis for a protocol translator in user space and has shown what needs to be done for additional protocol support. The shim-layer, presented in Section 3.1, operates in almost the same way, with the difference that all adaptions mentioned here happen in kernel-space. As the shim-layer’s dual setup (like the explained modification, too) only takes advantage of the SCTP multi-homing feature, a further adaptation is required for using multi-streaming.
3.3.3 Parallel Data Transmission with Multi-Streaming

In the following a parallel data transmission with multiple SCTP associations is assumed, which will later be adapted for a parallel data transfer with multi-streaming. The principle differences between parallel associations and a multi-streaming association are shown in Figure 3.5.

![Figure 3.5: Differences between parallel transfer modes in SCTP: (a) parallel associations, (b) single association with multi-streaming](image)

A data transfer with parallel associations can be created with several processes and/or threads. Depending on the implementation style (threads or processes) the socket descriptors are located in the same or in a different memory space. Different access rights from threads and processes to a socket descriptor are the consequence of memory management rules from the operating system. Every process always has its own memory space, while threads may share memory segments. These memory segments also contain the socket descriptors and can thus be shared by threads. Unlike threads, processes cannot share sockets but only exchange or inherit socket descriptors. With this method, as well as with a socket descriptor sharing, it is possible to get access from different processes/threads to the same association. In order to implement parallel data transfers with a single association and multi-streaming, association sharing between processes and/or threads must occur.

When it comes to association sharing further details have to be considered. Parallel transmissions on the same SCTP association share the values for congestion and flow control. Hence, every transmission on the multi-streamed association influences the congestion and receiver window in contrast to independent parallel transmissions with multiple associations.
The obvious way of porting parallel associations to a single association with parallel multi-streaming is to exchange/share sockets between processes/threads. In addition, a different stream is assigned for each transmission (stream mapping). An example is shown in Figure 3.6, where two threads share one socket descriptor and use two different streams for the transmission. The same is applied on receiver side, were the socket is shared and each thread reads only from the assigned stream. However, this strategy shows some deficiencies and such an implementation creates new error cases. The occurring problems and corresponding solutions are described in the following.

![Figure 3.6: A wrong approach of SCTP multi-streaming for parallel data transfers](image)

The transmission starts and the thread activities are carried out according to the scheduler rules in the operating system. Transmissions are dependent on the scheduling algorithm; sender and receiver threads are not synchronized. Two disadvantageous situations are now possible: receiver threads try to read if no data is available, or they do not read when data is available. Hence, concurrent transmissions could possibly be blocked, because the receive buffer is filled up completely with data from a single transmission. The reason for it is the single receive buffer, which is shared by all stream transfers, as a consequence of socket sharing. Furthermore, the flow control is applied only once for all transmissions on the association and an individual flow control per stream is missing. Mutual blocking can also occur, because the receive buffer contains messages for one receiver, which have to be processed first, before the subsequent message for another receiver is readable. Note that streams cannot be accessed separately through the socket interface with the current network stack implementation in Linux. SCTP sockets only refer to a single receive buffer which contains all messages, also from different streams.

In principle, the implementation with receive buffers for each stream is reasonable, as the figure in [13, p. 130] shows (as well as many other documents available on the Internet, too). In this illustration, “reordering queues” (receive buffers) for each stream are presented, which are used to pass the messages to the application. However, this is not valid for Linux applications, since in its network stack implementation, access to a separate buffer for each
stream is not feasible. The real access to the receive buffer in Linux was depicted in Chapter 2 - Figure 2.20. This figure shows the single receive buffer (used from all streams), which is accessible through the socket and contains all SCTP messages belonging to the association. As specified in [18], read operations may deliver the message to the application together with the stream number from which the message was received. With this option, separate buffers per streams could be implemented at the application layer. However, a further issue must be taken into account, which is related to SCTP’s message oriented data transmission. SCTP messages preserve their size when they are dispatched to the application. This can lead to problems if not the entire message fits into the receiver buffer, so that only parts of the message can be delivered. The partial delivery option and the fragmentation interleave control therefore need to be configured in an appropriate way, as otherwise a partial read of one message may block the receipt of another message, which could be delivered in the meantime.

The parallel transfer with multi-streaming demands a special implementation style, for which different solutions are possible. One possible approach is illustrated in Figure 3.7.

![Figure 3.7: Solution for parallel transmissions with SCTP’s multi-streaming](image)

The basis for this approach builds a separate message queue for each stream. The socket API does not provide such a possibility and therefore it must be implemented from scratch. One possibility is an appropriate user space implementation\(^\text{10}\) that creates a message queue for each stream and a single reader thread which reads all messages from the SCTP socket\(^\text{11}\). The processed messages are put from the reader-thread into the assigned queue from where the corresponding receiver threads can independently read the messages. The new stream buffers, used by the receiver threads for message receipt, must grow dynamically. Otherwise a buffer overflow may occur and an unprocessed single message in the socket receive buffer

\(^{10}\) For simplifying matters a user space implementation is presented in this work. However, a similar implementation can also be done in kernel space, which would be more complex, but with higher performance than a user space implementation.

\(^{11}\) The general rule for solving the multi-streaming blocking problem is that messages need always to be read if available (information from Michael Tuxen obtained by mail discussion).
can block all other transmissions. The mentioned approach solves the blocking problem for small messages, but for large messages a blocking can still occur. In order to solve this issue partial delivery as well as the message fragmentation interleave must be allowed. Alternatively, it has to be avoided to send messages bigger than the predefined maximal receive buffer size, which also solves the stated problem. Taking into account all these conditions a proper parallel multi-streamed data transfer is finally feasible.

Note that application messages larger than the PMTU are modified by the fragmentation and reassembly strategy of the protocol stack (mentioned in Chapter 2). However, this should be avoided as far as possible for transmissions with multi-streaming, since dependencies can be created thereby which may influence the transmission, as it is described in [57]. One solution is splitting up the data file into multiple messages at application layer, such that each message at the SCTP layer results in a PMTU-sized SCTP PDU and thus further fragmentation on the transport layer is not required.

The usage of multi-streaming for parallel transfers does not seem to be used in practice so far, as missing available implementations and references in this context show. As this is probably one of the first approaches using SCTP multi-streaming in such a way and not all issues are well studied yet, the presented approach should be used with caution.

For the sake of completeness, it should be noted that SCTP sockets have an increased feature set, compared to TCP and may operate in a UDP-like style. This allows the usage of a single socket for parallel transmissions to different end hosts and is described in [18] as the “one-to-many socket style”. The different associations are identified by an association-ID. Figure 3.8 shows what the socket style looks like, although this method is not used in this thesis. In connection with this mode it should be noted that the socket API functions and options are sometimes distinct or have a different functionality for each style.

Figure 3.8: One-to-many style of SCTP sockets

For the sake of completeness, it should be noted that SCTP sockets have an increased feature set, compared to TCP and may operate in a UDP-like style. This allows the usage of a single socket for parallel transmissions to different end hosts and is described in [18] as the “one-to-many socket style”. The different associations are identified by an association-ID. Figure 3.8 shows what the socket style looks like, although this method is not used in this thesis. In connection with this mode it should be noted that the socket API functions and options are sometimes distinct or have a different functionality for each style.
3.4 Practical Ideas for Implementations

In this section, the design of an enhanced protocol translation service, based on the previously discussed parallel multi-streaming principle, is described. The goal was to figure out a reasonable general approach for replacing all TCP connections that share the same end-to-end path by a single SCTP association with multi-streaming. Three different approaches will be discussed in the next sections, of which the last one shows to be most suitable. The first proposal in Section 3.4.1 is an SCTP encapsulation method, where TCP packets are transmitted as payload of SCTP packets. The second approach (Section 3.4.2) describes the adaptation of existing protocol translators, where some problems were encountered, such that a modification of the existing solutions is only possible to a limited extent. Still some ideas are used from the first and second model onwards, for the design and implementation of the third approach – the connection manager gateway – which is described in Section 3.4.3.

3.4.1 First Proposal: SCTP encapsulation

The initial idea was to encapsulate the TCP traffic within SCTP packets, but the idea does not seem to be completely new, as a patent application by Cisco Technology, Inc.[58] and a corresponding available product [59] demonstrate. The principle is to bundle multiple TCP packets from the same flow and to send the generated SCTP message over a multi-streamed association. Upon receipt on receiver side, the TCP packets are extracted from the SCTP message and forwarded. Multiple TCP sessions may be bundled into one SCTP association on multiple streams.

![SCTP encapsulation diagram](image)

**Figure 3.9:** SCTP encapsulation – a TCP packet is encapsulated within a data chunk

---

[12] This was only discovered after the tools described in this thesis were implemented and tested.
This approach was already rejected before the Cisco implementation was discovered, because of many open issues and complex tasks which would need to be implemented. Figure 3.9 shows what a possible encapsulation could look like; a drawback of this method is immediately ascertainable: the additional overhead due to the TCP header incorporation into data chunks.

A theoretical review revealed some further issues, and they are:

- An MSS-adaptation for the TCP PDU is necessary, such that one TCP PDU fits in one SCTP data chunk and a TCP PDU fragmentation gets avoided.
- A dynamical MSS-adaptation calculation is required, to consider variations over time due to the PMTU change on the path.
- Problems with the mapping (one connection onto one SCTP stream) because of missing connection identification information at startup (connection request port and effective transmission port can be different).
- The flow/congestion control mechanisms are performed twice (for TCP and SCTP).

The proposal did not seem appropriate to us, because of the introduced overhead that theoretically decreases the effective payload and the consequence being a reduced goodput rate. On the other hand, a field-tested implementation from Cisco already exists. However, implementation details of this solution are not available and thus unknown to us.

3.4.2 Second Proposal: Dual Setup with SCTP Multi-streaming

The second idea was to extend the presented withsctp tool with SCTP’s multi-streaming. The concept was developed with respect to the defined requirements for the new protocol translator approach, specified in Section 3.2. According to this, first of all, a dual setup (presented in Section 3.3.2) must be integrated, since this is not yet a part of this software. For this purpose socket-sharing is used, in order to exchange and to share SCTP sockets among processes and threads.

In Figure 3.10 an application scenario is illustrated with different access possibilities for socket descriptors of threads and processes. After applying the intended protocol mapping, the access to the socket descriptors should look like those in Figure 3.11, where by all applications a single SCTP socket is used for the same end-to-end path. The socket numbers in the figure represent different end-paths, and each TCP connection is mapped onto an SCTP stream. Furthermore, a TCP connection will not use SCTP if it is not supported by the opposite end host. Additionally, a general receiver thread must be used, together with dynamic stream message queues (arrows shown in red in the Figure) for each SCTP association, so that messages from all streams are processed in parallel. This is necessary if the SCTP association should work fault-free with parallel multi-streaming, as already previously described in Section 3.3.3.
Figure 3.10: General overview of different applications with processes/threads and the corresponding socket access (assuming that each socket maps to a TCP connection)

Figure 3.11: Possible application scenario after an applied protocol modification with SCTP multi-streaming (e.g. X-SY = socket X and stream number Y)

The basis for the new concept builds on the implementation of withscp. The withscp software modifies the socket API function calls in user space with a method offered by the operating system that allows the replacement of dynamic loadable system libraries (shared object files) at runtime, called preloading. This is also applicable for all socket functions, which are part of the dynamically loaded system-library. With preloading, original functions are simply replaced by a new function implementation, whereas the function call does
not change. The preloaded function works in the newly implemented manner and causes no reprogramming effort for the application itself, because the implementation changes are carried out directly in the library functions. The new library may be loaded for each application separately, which makes then use of the new functionalities in a transparent way (each TCP socket call performs the corresponding SCTP call). A detailed step-by-step description of the preloading-technique is included in [47]. Based upon this method the new implementation model in Figure 3.12 was developed.

Figure 3.12: Implementation model for a protocol mapping in user space

The model consists of three major part: the so-called “socket sharing pool”, the receiver thread (RT) and the preloaded socket API library. The composition is shown in Figure 3.12. By using the protocol translator, the application does not have access to the default API socket calls anymore. They are replaced (all preloaded) by new implemented functionalities, which make use of the original API functions in combination with specific SCTP functions. The application still retains and works with the original TCP socket descriptor, which is passed as usual parameter to all socket functions. The “new” socket-calls always make a look-up to the “socket sharing pool“ (socket look-up table). In this way, a SCTP socket descriptor and stream is obtained that are associated with the TCP socket parameters. The SCTP socket and stream are used for the further function execution, by

\[^{13}\text{This is valid only for those applications that use the protocol translator. The functionality of other applications is not influenced and thus the original function calls are used.}\]
which the TCP onto SCTP mapping is implemented. Note that the look-up table must additionally operate with IP addresses, because different socket descriptors may belong to the same connection. The relation building between sockets, connections and associations is implemented within the \texttt{bind()}, \texttt{connect()} and \texttt{accept()} functions. These functions build the new relations between SCTP association, SCTP stream and TCP sockets and are stored in a shared data structure, the ”socket pool”. The data structure for the look-up must be chosen carefully (e.g. hash tables), otherwise the socket descriptor look-up is carried out too slowly and makes the protocol translator unusable. Should it not be possible to set up an SCTP association, the application uses the TCP socket and thereby retains the original functionality. From the view of the application it is not obvious if SCTP is used, because only the TCP sockets are visible to the application. SCTP sockets are only used by the preloaded socket API. Every preloaded socket function works transparently and executes the corresponding functions with the SCTP socket, except for read operations; they are modified such that a reading occurs from the corresponding receiver stream buffer instead of the socket.

This approach has been implemented almost completely and was already working for small applications. However, a decisive issue was detected during the development: many applications make use of the non-blocking mode of socket functions, for which the \texttt{poll()} and \texttt{select()} call are used. These functions are mostly utilized in combination with \texttt{macros}, which are applied to register state changes on socket descriptors. The problem is that macros cannot be preloaded and thus a transparent replacement of them, as for the socket API functions, is not possible. An alternative strategy to obtain a translation behavior for these macros is the emulation of all registered socket state changes (TCP on SCTP and vice versa). An emulation at this operating system level would extremely slow down the mapping process. Moreover, the delay is not predictable, because it depends on the number of sockets used by \texttt{poll()} and \texttt{select()}. In order to solve this problem another attempt was made and the SCTP socket descriptors were made available to the application instead of the TCP descriptors. However, in this way, one of the most important properties gets lost: the possibility to fall back to TCP. Due to the encountered problems, it was the knowledge gained that this model has some shortcomings and does not suffice for a general applicability. The conclusion of all the performed work is that a modification of the \texttt{withsctp} software is only possible to a limited extent.

An alternative is to perform the same socket modification on a different operating system level and bypass the macros. A kernel implementation, like the \textit{shim layer}, seems to be better suited for a transparent mapping of all socket functions. However, the more complex implementation on this operating system level for the receiver threads and the realization of the look-up table must be considered. Note that also on this layer the socket states must be set appropriately, because one association belongs to multiple connections. In order to avoid running into a dead end again, the implementation of this model was suspended and a completely new approach was drafted.
3.4.3 Third and Final Proposal: Connection Manager Gateway

The third approach, depicted in Figure 3.13, describes the implementation of a connection management with a gateway application. This model combines the concepts of both previously presented approaches. The idea is not to transparently modify the application code in order to use SCTP, but instead the connection itself, akin to the first proposal. Since the TCP protocol is the most commonly used protocol on the Internet and the objectives (mentioned in Section 3.2) are focused on that protocol, others, like UDP, are not taken into account. However, it should be noted that the presented solution would also be applicable for different protocols in a similar manner. The following descriptions explain the proposal, which is shown in the figure.

![Diagram of Connection Manager Gateway](image)

**Figure 3.13:** Proposal for a connection manager gateway (CMG)

The first step of the implementation is the replacement of the original TCP connection, for which the original end-to-end connection by TCP is interrupted and partially replaced by a SCTP association. A crucial difference between the first proposal (explained in Section 3.4.1) and this one exists. The initial idea was to encapsulate the whole TCP packets into SCTP packets, so that the end hosts retain the original TCP connection. The new idea is to process only the contained payload of TCP packets, in order to avoid the additional
overhead otherwise introduced by the included TCP header. Therefore, all TCP packets first have to be received by the gateway and the payload needs to be extracted. The data payload will be put into a SCTP message and sent over a SCTP association with multi-streaming, which is established between the gateway hosts. The opposite gateway receives the data and forwards it to the receiver hosts, again with a TCP connection. All TCP connections on the same end-to-end path share a single SCTP association through the gateway application, where multi-streaming is used for parallel transfers. The gateway performs transparent mapping of each TCP connection onto an assigned SCTP stream. A cornerstone for this approach is a method somewhere on the end-to-end path that enables the TCP application to use the SCTP gateway.

One possibility is a connection redirection to the gateway. The idea is to redirect the connection request itself, so that a connection gets established between the host and the gateway. However, this method implies that the connection between the two end hosts is not established anymore, while it is still established between the sender and the first gateway and the second gateway and the receiver – provided that a SCTP gateway path is available on the original TCP path. After the intermediate connections are set up, which build the new path, a data transfer can happen. The applications on both end hosts work in the usual manner and have no knowledge about the SCTP transmission that happens in between.

As obvious from the description, the complex part of this approach is the implementation of the gateway service. The gateway application has to carry out many operations and the different tasks are briefly summarized in Figure 3.13. Additionally, a protocol for the communication between the two SCTP gateways is required, in order to exchange the connection management information.

The positioning of the gateways along a connection is variable and may happen at any point on the network. Therefore, a positioning of the gateways directly on the end hosts is possible, too. Note that a change of the gateway positioning results in different transmission behaviors. This is related to the congestion and flow control on the partial-paths from the end hosts to the gateways. As already explained in previous sections, TCP transmissions start in slow start mode and this implies a lower send rate at startup. Considering this, it is recommended to bring the gateway as close as possible to the endpoints, such that the disadvantageous effect of the congestion control at startup will be reduced.

An alternative to avoid problems with the congestion control on the TCP sub-paths is an adapted gateway design. For that purpose the gateway is placed directly on the end hosts and another communication protocol between application and gateway is used. A reasonable possibility for a local communication between gateway and application is the use of UNIX-domain sockets. Thus the stated problem with the TCP congestion control

---

14 Note that the end-hosts interpret the gateways as actual end-hosts. Different ports and IP-addresses may be used; this must be considered when the gateway is applied.

15 A Unix domain socket is used for inter-process communication (IPC) to transmit data as byte streams on the local host. The functionality is similar to an Internet socket, but it does not use a network protocol for the communication.
could be resolved, but a different method for the application is required to work with the gateway. This idea was developed to keep something in reserve, should the TCP connections between end hosts and gateways cause overlooked side-effect in a practical test. However, the advantages of maintaining the original TCP protocol by the user application prevail (less modification of the original application), and additionally a simple fall-back to the original application functionality with TCP is feasible. Therefore, the decision was made to develop a prototype implementation for a SCTP gateway with TCP connection mapping. In Figure 3.13 the essential properties of the connection manager gateway are highlighted, which are taken into account for the implementation in Section 3.5.3.

3.5 Implementation

This section presents relevant parts of the programming work. At first an insight is given into the multi-streaming programming principles in Section 3.5.1. In this context, the general development of software for a simple data transmission is described. Building on that, some software elements were implemented and made serviceable for configurable data transmissions, which are briefly explained in Section 3.5.2. The implementation details of the proof-of-concept software – a simplified form of the connection manager gateway – are the content of Section 3.5.3.

The implementation was carried out with the aid of books [55, 60] and SCTP programming articles [61, 62, 63]. Indeed, the most important document during the development was the currently available Internet-Draft of the SCTP socket API [18]. Note that the SCTP socket API is still in development. Therefore, the SCTP implementation stage varies across different operating systems and the socket interface is not yet completely integrated in Linux. The members of the LKSCTP-project [1] are engaged with the SCTP network stack implementation in Linux. Assistance for related problems with it can be achieved by the mailing list provided by this group.

3.5.1 Network Basics and SCTP Multi-streaming

At the very beginning some simple data transfer applications for both protocols, TCP and SCTP, were implemented. The following descriptions explain the overall developmental process of these network applications as well as the specific details to implement SCTP’s multi-streaming.

First of all a socket descriptor must be created, which builds the main interface for TCP connections and SCTP associations. A socket can be obtained by the socket() call and is further used as parameter by all other socket API functions. In Listing 3.1 it is shown how TCP and SCTP sockets are created. For the sake of completeness also the possibility for SCTP sockets in a one-to-many style is presented.
Network applications with the client/server principle usually consider a common sequence for the implementation, which is illustrated in Figure 3.14. All original functions of the socket API can also be used by SCTP sockets, but not all advanced features of SCTP are implementable thereby. The SCTP socket API introduces new functions, which complement the standard API and must be used for specific SCTP features. Figure 3.14 shows (in red) some new functions for using multi-homing and multi-streaming. The `sctp_bindx()` and `sctp_connectx()` functions are necessary for the multi-homing configuration, while the `sctp_recvmsg()` and `sctp_sendmsg()` functions are used for a data transmission on different streams. The detailed interface description for all SCTP functions is contained in [18] and is also part of the Linux manual pages.

The new data I/O functions of SCTP are of special interest, because they are not only used for receiving and sending data, but they provide the interface for protocol event messages, which are called notifications. The SCTP stack, at times, has information which is useful to share with the application. For this purpose additional messages, the notifications, may be passed to the application. By default all notifications are disabled. Specific notifications get enabled via a socket option call after the socket creation. In the same way the multi-streaming needs to be activated, as shown in Listing 3.2. Note that the presented configuration is only suitable for working with multi-streaming without a partial delivery. The message interleaving as well as the partial delivery of data messages must be configured additionally (per default both are disabled).

```c
/* SCTP events (enable data io r/w on stream) */
struct sctp_event_subscribe events;
/* enable receipt of SCTP Snd/Rcv Data via sctp_recvmsg */
memset((void *)&events, 0, sizeof(events));
events.sctp_data_io_event = 1;
if (setsockopt(sctp_socket, SOL_SCTP, SCTP_EVENTS,
              (const void *)&events, sizeof(events)) == -1) {
    perror("setsockopt() - sctp_socket");
    exit(EXIT_FAILURE);
}
```

Listing 3.2: Initial SCTP socket configuration for multi-streaming
Figure 3.14: Standard function sequence for a client/server application. SCTP functions for multi-homing and multi-streaming are shown in red.

Moreover, multi-streaming requires an initial configuration for the number of streams intended to use. This must occur before an association is set up, because in- and outbound stream values are exchanged at start-up with the INIT and INIT-ACK chunks and cannot be changed afterwards. An example of a stream number configuration is shown in Listing 3.3. An extension, until now only as Internet-Draft [21], describes a method to add and delete streams at runtime. Thus the mentioned limitations of the stream configuration would be solved.
The proceeding, after these special SCTP configurations, is the same for a TCP and SCTP implementation (assumed multi-homing is not used). The corresponding functions are bind(), connect(), listen(), accept(), whose application order is shown in Figure 3.14. After the association is established the data transmission can occur. The specific socket I/O functions (e.g. recv(), send()) can be used to read and write data. Furthermore, all standard data I/O functions can also be used for the data transmission with sockets, such as for instance the read() and write() function call. Note that all TCP applications, which are modified for SCTP support by simply exchanging the initial parameters of the socket() call, work from then on with messages instead of using a byte stream. This can eventually influence the whole application behavior and must be considered, should TCP applications become reprogrammed in such a way.

The data dispatched through the write() function is interpreted as a single message and the corresponding read() call on the opposite host must provide sufficient buffer space to read the complete message again. The same applies for all alternative socket I/O functions. The read()/write() calls as well as the other socket I/O functions do not suffice for multi-streaming and must be exchanged with sctp_sendmsg() and sctp_recvmsg(). Listing 3.4 shows an example for how a message can be sent with an additional setting of the payload protocol identifier16 (PPID). The stream number value specifies on which stream the message is sent and the message size is determined by the send_bytes parameter.

Listing 3.4: Example of sending SCTP messages over a specific stream

16 This value indicates the type of payload protocol data. In this work the PPID value was also used in another context, namely to signal the start and the end of application layer fragmented files, so that additional messages for that could be avoided.
After the data is dispatched, the SCTP network stack proceeds as described in Chapter 2. On the opposite host the packets are received and put into the receive buffer. The network stack behavior as well as the optional delivery methods were described in the multi-streaming example in Chapter 2 and illustrated in Figure 2.20.

The receiver host reads the messages from the buffer with a `sctp_recvmsg()` call, as in Listing 3.5. Data messages as well as protocol notifications may be returned. The function call returns only messages for enabled events (e.g. data I/O events as configured in our case in Listing 3.2) together with the corresponding additional values and the data structure parameter. The flag parameter of the `sctp_recvmsg()` call must be used for distinguishing notifications from data messages.

```c
/* read from SCTP association - message, stream number and ppid */
if ((read_bytes = sctp_recvmsg (sock, (void *)buffer, sizeof(buffer),
    (struct sockaddr *)NULL, 0, &sndrcvinfo, &flags )) < 0) {
    perror("sctp_recvmsg()");
}

/* get stream from which was read and ppid */
stream = sndrcvinfo.sinfo_stream;
ppid = sndrcvinfo.sinfo_ppid;
```

Listing 3.5: Example of reading from an SCTP association with multi-streaming

At the end the association is closed on both hosts. This happens with the `close()` call for a graceful shutdown of the SCTP association. An abrupt termination is possible, by sending the corresponding abort-message via a send call or simply through a program interruption. Based on the described programming steps the first software elements were developed.

### 3.5.2 Different Software Elements

Self-developed software elements were extended in a second step with parameterization options and with a transfer time measurement. The configuration settings enable the application to use different methods of file transfers. Additionally a polling method was implemented that allows logging of the congestion window (cwnd). The polling is realized by an independent thread and started at the beginning of the data transmission. The polling interval can be modified through a parameter. The cwnd value is retrieved by the `getsockopt()` function. An example for various connection and association status values obtainable with this function call is shown in Listing 3.6 (for SCTP) and in Listing 3.7 (for TCP).

Of course, a polling method is not really suitable for logging cwnd values, because values in between can be omitted due to the fact that a periodical value retrieval is not always fast enough. However, this possibility was sufficient for preliminary investigations of the interrelation between congestion control and acknowledgment procedures, which are described in Chapter 4.
In order to guarantee a fair comparison between TCP and SCTP, the same workflow is applied to both protocols, and the implementations follow the same principles. The same client and server were implemented for TCP and SCTP. Additionally, the possibility of transferring files using multi-streaming was implemented. The developed software provides the following data transfer modes:

**Simple transfer**  A serial and unidirectional data transfer of one or more files on a single connection/association. After all files are sent, the sender signals the transmission end by closing the connection.
Parallel transfer A data transfer of multiple files occurs in parallel with a connection/association for each file.

Parallel transfer with multi-streaming A data transfer of several files executed in parallel and with multi-streaming. Every file is transferred on a different stream with a separate process. The server reads only with a single receiver process from each stream.

It is worth mentioning that the data files transmitted by the SCTP applications are not interpreted as single messages, but split into smaller parts, for reasons mentioned in Section 3.3.3. The size of the application fragmentation is configurable via a parameter. To signal the end of a file, the protocol payload identifier (PPID) in the SCTP data chunk header was used and a corresponding value set, which is interpreted as EOF by the server.

The parallel data transfer with multi-streaming follows the same principles, but needs additional information on the receiver host about the number of files and corresponding streams used for the transmission. According to this, the received inbound stream value is used by the application to determine the amount of files (one file per stream).

3.5.3 The Gateway Prototype Implementation

The major design aspects of the developed prototype for the gateway concept in Section 3.3.3 are explained in the following. The corresponding source code is commented and available at the following web address: http://flori.bz.it/sctp.

The gateway implementation was carried out for two reasons: 1) to verify whether such an implementation is possible; 2) to prove the applicability and possible benefits of the presented gateway solution in practice. In these regard, two parts had to be implemented, namely the redirection of TCP connections and the SCTP gateway with multi-streaming.

TCP redirection The TCP connection attempt of the application must be modified in such a way that a TCP request is redirected to the gateway instead of going to the original end host. Additionally, the information of the original TCP connection must be sent to the gateway for the detection of a possible gateway path (the availability of an SCTP association on this end-to-end path). In case a connection setup with the gateway should not be possible, the original TCP function (fall-back solution) must be executed.

Gateway The gateway demands a service handling, which conforms to the concurrent server principle, such that parallel TCP connection attempts are processed with as little delay as possible. The original address information must be obtained for the gateway detection on the desired path. Should an association on this path already exist, it will be used; otherwise a new association has to be established. New association-requests from other gateways in the network must again occur in conformance with
the concurrent server principle. Information of the original TCP request is exchanged between the gateways. Based on the information received a new TCP request from the gateway to the original host is carried out. Upon success the redirected route between the end hosts is established and may be used for the data transfer. Data forwarding (read TCP - write SCTP and read SCTP - write TCP) must be carried out with various threads or processes and socket sharing. An appropriate parallel multi-streaming implementation (e.g. the presented approach in Section 3.3.3) is required for the reading threads, in order to avoid blocking-scenarios. At the end of the data transmission all used TCP connections are gracefully closed. The SCTP association is not closed and stays ready for possible subsequent transmissions. The stream assignment, data forwarding and error report require a special gateway protocol for the information exchange between the hosts.

From the abstract descriptions it already becomes obvious that the realization will get complex and needs a simplification for a first tentative implementation. Consequently, some assumptions were made for the development of the “proof-of-concept” software.

One self-imposed constraint is that only a single end-to-end path should exist, on which to deploy the gateway. Due to this restriction only the management of a single SCTP association must be considered and the implementation effort gets reduced for an otherwise necessary SCTP association management. The support of different IP-address formats is also closely related to the association and connection setup problem. Therefore the implementation only supports IPv4, while IPv6 is omitted. Furthermore, it is assumed that the number of parallel TCP connections is limited by a number of specified SCTP streams at gateway startup or by the maximal possible stream numbers (65,535) for one association. Finally, it is assumed that the gateway is well-positioned at local endpoints. All this constraints are valid for the implementation of both software elements and do not entail a modification of the discussed intentions.

The first part of the implementation is a method that redirects the TCP connection attempt to the gateway server on the local host. For that, the preloading technique, which is used by the withsctp software, was applied. As all TCP applications must use the connect() function to set up the connection, this function of the socket API was modified, such that a redirection occurs. The socket address parameters forwarded by this function call were exchanged with the local host address as well as the used gateway TCP port. The connection request is thus redirected to the gateway on the local host. Additionally, the original address parameters are sent to the gateway after the successful connection setup. Should the gateway server be unreachable or report an error (connection or association setup failure), the original TCP connection attempt is executed and the application operates in standard TCP mode. The redirection software is started together with the software that wants to make use of the gateway. In this way the application makes use of the “new” connect() call and if the SCTP gateway is reachable, the redirection process is executed. This solution is advantageous as a
reprogramming of the application is avoided. It should be mentioned that other possibilities for redirection are also possible, but not presented in this thesis.

The next step was the design of a simple protocol for the gateway communication. The gateway protocol is used for signaling the connection setup as well as for exchanging the information of the stream mapping between the gateway hosts. One association stream (stream zero) is reserved from the gateway for this purpose. Gateway messages carry a data structure which contains three values: the command, the TCP port and the reserved stream number. The protocol is kept as simple as possible and interprets only four cases which are signaled by setting the command as a number in the message. The commands and numbers are defined as:

- **Connection request (1)** - This message contains a reserved stream number and the TCP port of the original connection attempt. The reception of this message starts a TCP connection attempt from the gateway to the requested end host. The stream, which will be used for the connection mapping, is assigned if the attempt was successful.

- **Connection available (2)** - After successful completion of the TCP connection attempt (executed from the gateway to the end host) this message is sent to the opposite gateway.

- **Connection not available (3)** - After unsuccessful completion of the TCP connection attempt (executed from the gateway to the end host) this message is sent to the opposite gateway.

- **Close connection (4)** - Every unavailable TCP connection (EOF or ERROR) is signaled with this message, which contains the corresponding stream number for the mapped TCP connection. The reception of this message implies a TCP connection shutdown on the gateway where the message is received. Moreover, the stream number is released and may be used for other connection redirections.

According to these descriptions the message passing for the gateway protocol was implemented. The PPID value of SCTP messages was used to distinguish the received gateway protocol messages from the forwarded data messages, which are all processed by the single receiver thread. Hence the PPID value is appropriately set for each message type. Data messages are identified by a PPID value of zero, while gateway protocol messages have the value set to one. The simple gateway protocol is therewith complete and has been integrated in the described manner in the server application.

The last step was the implementation of a proper gateway server application. Unlike a sequential server, a concurrent server is able to serve more than one client at a time, exactly as demanded for this implementation. To meet these requirements, the practical examples presented in [60] were used as basis and a multi-threaded server architecture was developed. The workflow and activities of the different threads for a single TCP connection mapping are depicted in Figure 3.15.
The figure shows two running server applications; one for each gateway endpoint. First of all, the main process starts two service threads: the “connect service” thread (CS) and the “accept service” thread (AS). The CS-thread waits for incoming TCP requests from redirected connections, while the AS-thread takes over the SCTP association requests from other gateways. As soon as the CS-thread registers a TCP request the “request TCP” thread (Re) is started and further operations of the accepted connection are carried out by this thread. Acting in such a way allows to process parallel TCP requests. The Re-thread reads the address structure (IP and Port) of the original destination, which is sent after the connection setup through the preloaded `connect()` call. Subsequently a lookup\(^\text{17}\) for an already existing SCTP path is performed where the result is one of the following:

1. Association not available - A new SCTP association must be established, carried out by a new thread, the “connect stream reader” thread (CR). This is the only thread which reads messages from all streams of this association. Should it be impossible to establish an SCTP association to the opposite gateway server, this thread will be closed and the system returns to the original TCP functionality.

\(^{17}\)At this point the SCTP association management would need to be integrated, which is still missing in this implementation (only proof-of-concept).
2. Association establishment in progress - Should it happen that many requests come in at the same time, only a single SCTP initiation attempt must be performed, while all other Re-threads have to wait until the definitive association state is available. The system returns to the original TCP functionality if the association is unreachable.

3. Association available - The SCTP association to the opposite gateway as well as the CR-thread already exists.

After the successful SCTP association setup or the detection of an existing one, the mapping process takes place. One TCP connection is mapped onto one SCTP stream. This is carried out by the stream assignment procedure, which is shown in Listing 3.8. Should it happen that no free stream is available anymore, a return to the original TCP connection occurs.

```c
/* stream assignment loop */
do {
  if(streamlist[next_stream_avail] == -1)
    stream = next_stream_avail;
  next_stream_avail++;
  count++;
  next_stream_avail = next_stream_avail % MAXSTREAMS;
  /* don't use stream 0 for connection mapping */
  if (next_stream_avail == 0)
    next_stream_avail = 1;
} while ((stream == 0) && (count <= MAXSTREAMS));
```

Listing 3.8: Stream assignment loop

The assigned stream number, together with the original TCP port, form a gateway message, which is sent to the opposite gateway. The following events take place on the second gateway.

As on the other gateway server, the main process has started the two service threads (AS and CS). The AS-thread processes all SCTP association requests from other gateways. An SCTP request implies the start of a new thread, namely the “accept stream reader” thread. Acting in such a manner allows to process parallel SCTP requests. The running AS-thread is the only reader thread (AR) on this association and reads all messages from the different streams as well as the request message sent from the opposite gateway. Upon the reception of the “request-message” again a new thread, the “connect TCP” thread (Co), is started, which tries to setup a TCP connection between the gateway and the end host application. Should it be impossible to establish a TCP connection, an error is reported to the gateway from which the request was send (gateway signaling). Consequently, a return to the original TCP behavior occurs. On success, the stream number is assigned, the rw-thread for the data forwarding is started, and an appropriate success message is sent back. The gateway on this side is now ready for data forwarding. The subsequent events happen again on the first gateway.
Just after the request message was sent from the first to the second gateway, a message should be returned to it (after the previous mentioned steps happened on the opposite gateway). The gateway protocol message contains the information whether the connection to the endpoint could be established or if an error occurred. Based on this information a new thread, the “read-write” thread (rw-thread), is started or a return to standard TCP occurs. After the rw-thread was started also on this gateway the setup procedure for the TCP connection mapping is complete. The end hosts use the new gateway route for data transmission from there on, until one of the TCP connections is closed. The closing of one of the TCP connections starts the gateway messaging again and the corresponding gateway messages signal that the stream may be discharged. On both sides the corresponding rw-threads are closed and the final cleanup procedures are executed. The SCTP association is kept until one of the gateways is closed.

The execution of the gateway server application on both end hosts and other network applications in combination with the redirection element represents the implementation of the discussed approach in a simplified form, which was used for the measurements in Chapter 4. The solution has some shortcomings, but the implementation is sufficient to perform the first practical experiments.

### 3.6 Summary and Conclusion

Existing and new implementation approaches were discussed in this chapter. The main intention behind them is to increase the deployment of SCTP and thus achieve benefits. The best solution for this purpose seems to be protocol translators and gateway applications. During the analysis of existing software, it was noticed that multi-streaming was not taken into account, although this feature promises to be suitable for doing faster data transmissions. The motivations of using multi-streaming in a beneficial way for parallel transfers as well as related issues were presented. An attempt was made to modify the existing software and to extend it with the multi-streaming feature. However, a reasonable way for integrating this feature into existing approaches it could not be found. The reason is that the currently provided SCTP socket API demands a special implementation style, which is incompatible to the one used in these existing approaches. An alternative was the design and development of a new gateway solution, as presented.

The result of the different implementation steps are several software elements, which are appropriate for a comparison of SCTP and TCP on equal grounds, as well as a proof-of-concept implementation for the connection manager gateway. This software is used in Chapter 4 to gain more experience about SCTP transmissions and the multi-streaming feature. The goal is to verify the improved functionality by replacing TCP connections with an SCTP association and multi-streaming.
Chapter 4

Experimental Evaluation, Measurements and Troubleshooting

This chapter presents the performed measurements, which were made to obtain a performance estimation of the current SCTP network stack in Linux and to evaluate the proposed gateway solution. The experiments with the gateway prototype serve as a proof-of-concept that this idea is not only theoretically but also technically feasible.

Section 4.1 contains the experimental network configuration and parameter settings of transport protocols. Adaptations of optional protocol settings were necessary for a fair comparison of TCP and SCTP transmissions. Preliminary measurements are presented in Section 4.2. The first experiments had the objective to investigate the general SCTP performance and to identify use cases for the application of the SCTP gateway. Further experiments demonstrate the application of the gateway prototype in combination with standard network applications. The outcome is an enhanced data transmission in specific use cases, which is presented in Section 4.3. The various application scenarios are explained and the advantages as well as disadvantages are discussed. Discovered shortcomings of the prototype implementation are described in Section 4.4. Additionally, some SCTP related problems are mentioned at the end that should be considered by all SCTP applications.

4.1 Experimental setup

The network testbed used for the practical experiments is presented in Section 4.1.1. Depending on prevailing network conditions, appropriate adaptations are necessary for a good performance. Moreover, protocol-specific settings are possible and need to be considered for comparative measurements and their interpretation.
Similar protocol algorithms are the basic prerequisite for the experiments with the SCTP gateway, as the benefits of the gateway application can then be better investigated and other influences for an improved performance can be excluded. Parameters such as, for instance, the congestion control algorithm and the acknowledgement procedure need to be adapted. The specific protocol configurations for that are described in Section 4.1.2.

As measurement results are not only influenced by transmission algorithms but also by the network hardware, it is necessary to adjust network dependent parameters for both transport protocols in the same way. Used technologies on the data link layer determine bandwidth and path delays, which in turn influence the congestion and flow control behavior. Contrary to TCP, SCTP has no auto-tuning mechanisms integrated and requires appropriate manual adjustments for good performance. Steps to be taken for improving SCTP performance are described in Section 4.1.3.

The experiments were carried out with several software tools. Measurement instruments and standard applications that were used in this context are presented in Section 4.1.4.

### 4.1.1 Network Testbed

The initial idea was to perform experiments directly in the Internet, but this was not done for two reasons. First, SCTP has problems with middleboxes that do not support SCTP. Second, the Internet is not really appropriate for initial measurements because of many incalculable side-effects. Consequently, the practical measurements were carried out in a typical SOHO (small office/home office) network with specific protocol configurations of TCP and SCTP.

The experiments were carried out with the testbed depicted in Figure 4.1. The figure shows the device composition of a 100 Mbps Ethernet network. The details of hardware and software used for the basic configuration are described in Appendix A. Note that the use of other components can easily lead to changes in measurement results and this must be taken into account so that the results are reproducible.

![Network Testbed](image)

**Figure 4.1:** Network testbed used for the practical measurements
A network emulator (netem)\(^1\) was used to simulate properties of wide area networks. The emulator software was applied on the intermediate router, in order to generate constant path delays in both directions. A prerequisite to properly function is a sufficiently large queue at the router\(^2\). The reason for the network emulation is that a practical reference measurement would not be meaningful without delays on the network paths. The effects of congestion control algorithms get significantly influenced by packet delays, and they are small in a standard Ethernet network (0.5 ms) compared to the delays in the Internet (up to 200 ms and more [64]).

The protocol analyzer software *wireshark* was used to record the data flows of any transmission, so that a deep packet inspection could be made. Additionally, the results obtained with the measurement tools were controlled with the help of this software. The *wireshark* tool is very similar to the *tcpdump* software. The difference is that *wireshark* has a graphical front-end, and many more information sorting and filtering options.

Experiments require the use of the latest stable version of the protocol stack implementations. This is necessary for the results of performance measurements to be up-to-date. In particular, the SCTP network stack in Linux systems showed performance improvements with almost every updated kernel version. The details of software versions and operating system versions used for the experiments are listed in Appendix A. Note that a special SCTP kernel patch was applied. This patch adds a slight change to the congestion window calculation of SCTP if the association is in idle-state. RFC 4960 [10] specifies that the congestion window in such a case should be adjusted to \(\max(\text{cwnd}/2, 4 \times \text{MTU})\) per RTO. Although the draft recommends that this check is done every RTO interval, the SCTP Linux implementation – through the application of this patch – does this with every heartbeat interval (default 30s). The altered operation is useful for the gateway solution, as subsequent transfers benefit from the already open congestion window.

Table 4.1 summarizes important details of the network that must be taken into account for the parameter optimization, as is explained in the next sections.

<table>
<thead>
<tr>
<th>Network details</th>
<th>Specific values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link layer protocol</td>
<td>Standard Ethernet with CSMA/CD</td>
</tr>
<tr>
<td>Frame PDU (PMTU)</td>
<td>1500 bytes</td>
</tr>
<tr>
<td>IP version</td>
<td>IPv4</td>
</tr>
<tr>
<td>Network bandwidth</td>
<td>100 Mbit/s (12.5 Mbyte/s)</td>
</tr>
<tr>
<td>Path delays</td>
<td>0 - 200 ms (0.5 ms without generated delay)</td>
</tr>
</tbody>
</table>

**Table 4.1:** Summary of important testbed network details

\(^1\) The network emulator software is integral part of the Linux operating system (Ubuntu 9.04 – Kernel 2.6.31) and allows emulating variable delay, loss, duplication and re-ordering.

\(^2\) An emulation of higher path delay requires a router queue adaptation, otherwise packets can get lost and the consequence is a completely different network behavior.
4.1.2 Protocol Parameter Adaptation

A couple of protocol settings are configurable and the default settings vary with the used operating system. SCTP and TCP use different congestion control algorithms as well as different acknowledgment procedures that make a general comparison difficult. These differences lead to different transmission rates as consequence of the different congestion window calculation. Therefore, it is wrong to speak of a comparison of transport protocols, because actually only a comparison between specific protocol versions is possible. The selected protocol configurations, which are described in the following, represent such protocol versions for TCP and SCTP.

The experiments with the gateway approach require an adaptation of TCP and SCTP such that a fair comparison between both protocols is possible. A similar congestion control for TCP and SCTP is decisive for this\(^3\). The default TCP algorithm “CUBIC” was exchanged with the “RENO” congestion control. CUBIC TCP is an optimized congestion control algorithm for high speed networks with high latency, and is used by default in Linux kernels 2.6.19 and above.

Even if TCP and SCTP use the same congestion control, a different transmission behavior is possible due to the different acknowledgement procedures. The congestion window (cwnd) calculation of standard algorithms (like RENO) is related to the receipt of ACKs. Figure 4.2 shows how the usual ACK of TCP and the delayed selective acknowledgment (SACK) of SCTP in slow start proceed. The SCTP ACK-procedure reacts in several ways and the different cases were described in Chapter 2. The example in Figure 2.13 illustrates the default case for SCTP (every second data packet within the time limit is ACKed). Even from the theoretical observation (Figure 4.2), it is already evident that TCP increases the cwnd faster as SCTP, because of the larger number of received ACKs. Each received TCP ACK increments the cwnd in slow start by one sender MSS. Unlike TCP, SCTP does not increases the cwnd value consistently with the sender MSS. SCTP increments the cwnd by one MTU for each received SACK packet if the value of acknowledged bytes is greater than the MTU value. Otherwise (and only in slow start) the number of acknowledged bytes is added to the current cwnd value (similar to TCP Appropriate Byte Counting – RFC 3465 [65]). The different growth rates of cwnd in slow start were investigated in practice (in Section 4.2) and the results confirm the aforesaid. In summary, two SCTP settings must be considered when interpreting performance estimations: delayed SACKs and the size of PDUs.

A further difference exists between TCP and SCTP. The selective acknowledgment procedure is implemented for both, but SCTP has an enhanced SACK implementation. TCP SACK blocks (the number of gap blocks for each ACK) are limited by the option field in the TCP header, while the number of SCTP SACK blocks (per SACK packet) are limited by the PDU size. Each TCP ACK contains a maximum of three SACK blocks if

---

\(^3\) Benefits that can be achieved through an improved congestion control should not be the only reason for deciding in favor of a specific protocol, because the algorithm could be implemented for the other protocol just as well.
the timestamp option is enabled, or four blocks without using this option. This limitation is one reason why SCTP outperforms TCP if both have SACK enabled and comparative measurements are carried out in environments with high packet loss rates, as for instance in [2, 31, 43] and [52]. The results of these scientific papers have already proved that SCTP performs better than TCP under such circumstances. Hence, practical measurements with network conditions that cause many packet losses were omitted in this work and the testbed was configured to work fault free so that TCP SACK and SCTP SACK perform in a similar way.

In consideration of all these above-mentioned points, adapted TCP and SCTP versions were used for all experiments. The protocol adaptation was carried out with the help of a shell script by setting the various parameters via system control variables. The setting of system variables is explained in Appendix A.4.

4 The computation for these values is explained in detail in RFC 2018 [12].
5 With this it is meant that the testbed was tuned to produce as little packet loss as possible, as no completely fault-free network exists in practice.
4.1.3 Transmission Optimization

A couple of possibilities that provide enhanced performance to network applications must be considered, as they also influence the results of performance measurements. Important factors are:

- the minimization of packet transmission latency
- the minimization of system call overhead
- the adaptation of socket buffers (window sizes) for the bandwidth-delay product

The communication through the socket API implies that dispatched data is possibly split up into blocks, and PDUs at the transport layer are formed, which are limited by the MSS. The goal is to fill each transport layer PDU with as much available data as possible. TCP and SCTP use the Nagle algorithm to combine a number of small data blocks and to transmit them all at once within a single PDU. SCTP uses a chunk bundling for that and TCP appends additional bytes until an MSS-sized PDU is formed.

The Nagle algorithm introduces some latency, but with the benefit of minimizing the number of packets that need to be sent. However, the additional delay is not always desirable. In case where transmission latency has to be avoided as far as possible, the Nagle algorithm can be disabled (through an optional parameter of the socket API). As all measurements in this work are carried out with enabled Nagle algorithm, an additional performance improvement might be possible in some special cases, but this was disregarded in this work.

Conducted measurements in [41] show that SCTP has a reduced performance if many small packets are sent. For that reason, at least MSS-sized data blocks or larger messages are recommended in order to achieve higher transfer rates. Moreover, the data block size at the application layer may influences the throughput rates, because SCTP works message oriented. As detailed research outcomes about side-effects related to the fragmentation and reassembly strategy of larger SCTP messages are still missing, these effects were avoided by sending MSS-sized messages. Note that measurement results may change if different message sizes are used at application layer.

The reduction of system call overhead is another optimization possibility. All network operations must be executed in a special operating system mode (kernel mode). Whenever a reading or sending of data occurs, system calls are executed and a mode as well as context switch occurs\(^6\). These calls are time expensive and a reduction of context switches results in overall transmission speed-up. Especially the number of read/write calls has an impact on the application performance, and this is related to the buffer size used at the application layer to pass the data to the transport layer. The efficiency of I/O operations is investigated

\(^6\) A context switch involves storing the old state and retrieving the new state of CPU registers. As kernel processes are running with different privileges, a context switch happens when a kernel function has been called in user mode.
and exemplified in [55, p.177-179]. According to these results buffer sizes in the range of 2048 - 16384 bytes are recommended to reduce the overhead of read/write system calls.

SCTP is here in a dilemma. An application layer fragmentation of large files into MSS-message-blocks is recommended to bypass the fragmentation/reassembly mechanism of SCTP. Sending messages with this size (assumed Ethernet is used) implies more context switches compared to bigger messages. The SCTP message size determines the number of socket I/O calls and thus the number of context switches. Studies on this problem are missing and so the decision was made to use MSS-sized messages for the gateway approach. However, it should be kept in mind that changing this size might have an influence on the performance.

Not only the buffer/message sizes at the application level are important, but the buffer used by the network stack as well. This buffer is even more important because flow control is related to it. The optimal size depends on network conditions, such as for instance the available bandwidth and path delays. A special peculiarity of the Linux TCP implementation is the integrated auto-tuning mechanism, while SCTP always needs to be manually adjusted for prevailing network conditions. Therefore, SCTP has problems with fluctuating network conditions. On the contrary, in TCP, metrics are saved during the transmission and appropriate adjustments for optimal receive buffer sizes are carried out at runtime. As the measurements in this work are conducted with same bandwidth and constant path delays, an appropriate manual adaptation of SCTP buffers is practicable. For good performance, the sizes of send and receive buffers are important as otherwise the available network capacity cannot be fully utilized. The consequence would be a dramatic performance loss.

<table>
<thead>
<tr>
<th>Value settings</th>
<th>TCP</th>
<th>SCTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send buffer (swnd)</td>
<td>16 Kbyte</td>
<td>112 Kbyte</td>
</tr>
<tr>
<td>Receive buffer (rwnd)</td>
<td>85.3 Kbyte</td>
<td>112 Kbyte</td>
</tr>
<tr>
<td>Maximum segment size (MSS)</td>
<td>1448/1460 bytes</td>
<td>1452 bytes</td>
</tr>
</tbody>
</table>

Table 4.2: Default settings of buffer sizes in Linux and the MSS in the test network (TCP MSS with timestamp option enabled/disabled)

SCTP and TCP have different default values (shown in Table 4.2) for the buffer size. The size of the receiver buffer, which is used by the flow control algorithm, must be appropriately adapted for prevailing network conditions. An optimal buffer size can be calculated via the Bandwidth-Delay Product (BDP). The BDP determines the amount of data that can be in transit in the network and varies with the available bandwidth and latency. The BDP value is computed by $BDP = bandwidth \times RTT$ and the result corresponds to an optimal buffer size that may be used for saturating a link. Table 4.3 shows the computed BDP values which were used for the measurements in the next sections.
### Table 4.3: Bandwidth-delay product in a 100 Mbps network for different path delays

<table>
<thead>
<tr>
<th>Path Delay - RTT</th>
<th>Bandwidth-Delay Product (BDP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ms</td>
<td>125000 bytes (122 Kbyte)</td>
</tr>
<tr>
<td>30 ms</td>
<td>375000 bytes (366 Kbyte)</td>
</tr>
<tr>
<td>60 ms</td>
<td>750000 bytes (732 Kbyte)</td>
</tr>
<tr>
<td>100 ms</td>
<td>1250000 bytes (1.19 Mbyte)</td>
</tr>
<tr>
<td>200 ms</td>
<td>2500000 bytes (2.38 Mbyte)</td>
</tr>
</tbody>
</table>

The socket buffer in Linux has a special peculiarity; the reserved memory is shared between application and kernel. TCP maintains a part of the buffer for the TCP window, which is the advertised receiver window value. The remaining buffer space is used as application buffer, in order to isolate the network from scheduling and application latencies. The default setting (value of 2) implies that the space used for the application buffer is a quarter of the total space. The remaining buffer space is the receiver window (in this case three quarters). SCTP socket buffers work in a similar way, but do not have the possibility to change the share ratio, as with TCP socket buffers. Thus, an adaptation is necessary that both protocols share the buffer in a similar way – otherwise SCTP and TCP work with the same buffer settings but with different receiver window values.

Both protocols, TCP and SCTP, have been adjusted for the measurements by taking into account the mentioned optimization possibilities. The various options were configured by setting global operating system control variables (exemplified in Appendix A.4). The script was executed before the experiments have taken place. Note that, after restarting the system, such settings are lost and the default operating system values are loaded.

### 4.1.4 Software Used for Experiments

The practical experiments were carried out with various software, which is listed in Appendix A.3 together with the used version. The most important software in this context is the operating system (OS). The choice of the OS determines the performance of protocols, because they are generally a part of the kernel. The experiments in this work were carried out with the Linux operating system Ubuntu [66] – kernel version 2.6.31. Note that default settings of specific TCP and SCTP parameters may vary with the operating system. The practical measurements required an appropriate adaptation of these default values. The modified system variables with the new values are listed in Appendix A.4.

The initial performance estimation of TCP and SCTP was made with `iperf` [67]. This software can be used to measure the TCP and UDP performance, but SCTP support is not implemented. Internet research has revealed that an older version of this software exists, which was modified for SCTP support. However, the decision was made to use the newest available version and to carry out SCTP measurements by using `iperf` in combination with
the withsctp software, which exchanges only the TCP socket with SCTP sockets while the main functionality remains the same. However, it should be noted that SCTP and TCP performance also depends on the application implementation.

Comparative measurements of SCTP and TCP are particularly difficult, because not much software is available that makes it possible to carry out experiments with the same workflow for both protocols. Therefore, some software elements were self-developed, in order to compare identical SCTP and TCP data transfer implementations. Moreover, the self-developed software was used to investigate the congestion window evolution of TCP and SCTP transmissions at runtime.

The final gateway experiments were carried out with an “Apache” web server [68] and the “Firefox” web browser [69]. The web browser was additionally equipped with the “Firebug” Add-on [70]. Firebug is a web development tool for firefox that can accurately analyze network usage and performance. The integrated time measurement was used to measure the time it takes to load different web pages from the web server. On the server various web pages are stored, including several images of the same size. By downloading these web pages, different ways of transmission were examined and the transfer time was measured. The same experiments were carried out with the “GNU Wget” [71] software, which also can be used for downloading files from web server. This software is a non-interactive command line tool and was integrated in a Python-script together with a simple self-programmed time measurement method.

All of the following experiments were executed several times to guarantee consistency by reducing the variance of measurement results. Each individual experiment was executed 30 times and the results presented in the diagrams are the computed mean of all runs with same configuration.

4.2 Preliminary Measurements

Preliminary measurements were carried out to investigate the performance of the SCTP network stack. A further purpose of these measurements was to figure out how performance may change if SCTP instead of TCP is used for the data transmission. The various experiments are described in the next sections and the results are depicted graphically.

At first, throughput measurements were carried out for an initial estimation of SCTP’s performance in comparison to TCP. The experiments show why protocol parameter adaptations are necessary. In a second step, various experiments were carried out with the self-developed software elements. In this context, the principle of persistent connections was simulated to highlight the possible speedup of data transmissions with an already open congestion window. Moreover, the evolution of SCTP’s and TCP’s congestion window in slow start has been studied in more detail. In the end, different methods are described that can be applied for the transmission of several files with TCP and SCTP.
4.2.1 Performance Measurements with Iperf

The throughput measurements with iperf show the performance of TCP and SCTP with default settings and with adapted protocol settings. Note that SCTP experiments were carried out with a software combination of iperf and withsctp. All iperf experiments were conducted with the preconfigured standard test case: a single unidirectional data transmission with a duration of 10 seconds. The default application buffer size of iperf, used for read/write operations, was 8192 bytes, by which also SCTP’s message size was determined.

**Experiment 1.1** The measurements with iperf were carried with a constant bandwidth of 100 Mbps and with different RTTs of 0.5 ms, 10 ms, 30 ms, 60 ms, 100 ms and 200 ms\(^7\). The path delays were generated with the network emulator on the intermediate router. For both end hosts the standard OS settings for SCTP and TCP parameters were used. As a result of this measurement, a large performance difference between TCP and SCTP was expected, because the default configurations of protocol-specific parameters are not always optimal.

![Throughput measurement with iperf in 100 Mbps network (10sec)](image)

**Figure 4.3:** Iperf measurements with default TCP and SCTP settings. The comparison is not quite fair because TCP uses the CUBIC congestion control, and the receiver window of SCTP is not automatically adjusted, like that of TCP.

Figure 4.3 depicts the results of the iperf throughput measurement with different RTTs. From the diagram it can be seen that SCTP’s performance drastically decreases with increasing latency. The monitoring of data transmission has revealed that SCTP did not

\(^7\) The RTT values were determined with the ping-command on the Internet (60ms, 100ms, 200ms). The smaller values (10 ms and 30 ms) were chosen freely to fill the gap in between. The value of 0.5 ms is the real RTT in the Ethernet testbed.
always use the full bandwidth. The transmission rate was limited by flow control due to the small receiver window, and this drawback increases with growing BDP. However, this shortcoming was not detected for TCP transmissions. The reason is that, in contrast to SCTP, TCP (in Linux) has a method implemented which automatically adjusts socket buffer sizes at runtime. Thus, flow control permits higher transmission rates for TCP than for SCTP. Not only flow control is influenced by path delays but also congestion control. The measurements show that with increasing latency throughput rates decrease. Due to the delays of ACK packets the calculation of cwnd gets influenced and thus it takes longer until link capacities are saturated.

During the experiment it was observed that TCP transmissions can be influenced by previous transfers\(^8\). This is due to network metrics that were recorded and then used for an optimization of TCP parameters at runtime. Moreover, different congestion control algorithms decisively influence measurement outcomes and make a fair performance comparison between TCP and SCTP difficult (TCP CUBIC vs. SCTP RENO). The aim of this work was to investigate specific SCTP properties for general transmission acceleration. Therefore, similar TCP and SCTP protocol versions must be used for the initial assessment of performance.

The conclusion of this experiment is that SCTP requires appropriate manual adaptations, because automatic buffer tuning, like for TCP is not available. Otherwise, SCTP cannot compete with TCP and shows a poor performance in networks with high bandwidth and/or high path delays. Moreover, similar TCP and SCTP congestion control algorithms should be used for further measurements, so that a comparison on equal grounds can occur.

**Experiment 1.2** The previous experiment was repeated with some modified protocol settings. The size of SCTP’s socket buffer was adjusted according to the BDP. New buffer sizes (Table 4.3) were set by passing the values as parameter to the iperf application, but only for SCTP. TCP used the automatic buffer tuning. The congestion control of TCP was changed from CUBIC to RENO. In addition, some protocol-specific settings were adapted (settings in Appendix A.4), so that TCP and SCTP work in a very similar manner. These configurations allow a better comparison of TCP with SCTP, and a similar outcome for TCP and SCTP was to expect.

The measurement results of the second iperf experiment are illustrated in Figure 4.4. In comparison with the results from the first experiment (Figure 4.3), a huge performance gain can be seen for SCTP. The reason for the improved SCTP performance is the socket buffer adaptation. Due to the optimized socket buffers, the full bandwidth could be used and thus higher throughput rates were achieved.

A further observation can be made by comparing results of the first and the second experiment. For both experiments, different TCP protocol configurations were used, but the

\(^8\) This is an optional setting of TCP and was disabled for all TCP measurements in this work, so that results are reproducible.
results show an almost similar performance, except for higher latency. The difference is due to the applied congestion control algorithm – TCP CUBIC for the first (Figure 4.3), and TCP RENO for the second experiment (Figure 4.4). As reminder, TCP CUBIC is optimized for high speed networks with high latency and outperforms RENO congestion control when such network conditions prevail, as for example in the test with a 200 msec RTT.

The diagram in Figure 4.4 shows that protocol versions of TCP and SCTP were found which make a fair comparison possible. SCTP and TCP show a similar performance for the second experiment, but still TCP performs slightly better. Theoretically, the contrary would be to expect – a marginally better performance for SCTP – since SCTP can transfer larger PDUs than TCP. A larger difference between TCP and SCTP is striking in the test run with 10 msec RTT, but the cause could not be determined. It was also tried to once more adapt buffer sizes (larger or smaller than BDP), but these modifications did not lead to improvements.

Based on the protocol analysis in Chapter 2, some points could be identified that may influence the performance of TCP and SCTP in a different way. Some of them are:

- the acknowledgement procedure – SACK vs. delayed SACK
- the PDU composition at transport layer – byte stream vs. messages
- the connection/association setup – 3-way handshake with data bundling at startup vs. 4-way handshake without data bundling
- the CRC checksum calculation – CRC16 vs. CRC32
It has not been investigated in detail to what extent the transmission rate is influenced by all the mentioned points, as this is beyond the scope of this work. However, an investigation of the evolution of cwnd values was carried out, which is explained in the next section. The cwnd computation is related to the acknowledgement procedure and due to the theoretical observations, described in Section 4.1.2, a decisive impact on the performance is to be expected.

The outcome of the iperf experiments led to the conclusion that transmissions are generally carried out faster with TCP than with SCTP even if a similar protocol version is used. A further decisive advantage of TCP is the possibility to use different congestion control algorithms. The measurements confirmed that this is beneficial because TCP CUBIC works in some cases better than the RENO algorithm of TCP and SCTP. Optional congestion control algorithms are still missing for SCTP, and the same applies to the automatic socket buffer adjustment. Such advanced functionalities definitely need to be implemented for SCTP too, as otherwise its use probably will be avoided in many cases.

### 4.2.2 Data Transfer Measurements - TCP vs. SCTP

Measurements were performed to investigate in more detail the reasons for the difference in performance of TCP and SCTP. For that purpose, self-developed software elements with an identical workflow implementation for TCP and SCTP were used to measure the time for the reception of differently sized files. Reception times were tracked at the receiver. The measurement started after the connection was established and ended after the entire file was received. In addition, the cwnd values were recorded, which allow a conclusion on the time evolution of transmission rates.

For all following measurements a bandwidth of 100 Mbps with an emulated constant RTT of 60 ms was used. This RTT was chosen because the iperf experiments have shown the smallest variance with this value. Besides, this RTT also reflects a real value in the Internet.

**Experiment 2.1** The same protocol configurations as for the second iperf experiment (experiment 1.2) were used. Single files of different size were transferred with the self-developed software. The SCTP message size was determined by the application buffer size (4096 bytes). The time was measured on the receiver host. All test runs were additionally controlled with wireshark. Due to the previous measurement results (Section 4.2.1), a better performance for TCP than for SCTP was expected. However, the difference between TCP and SCTP should be less when larger files are transferred, because SCTP uses a larger MSS than TCP. Consequently, SCTP has a better goodput than TCP. The number of required data packets for the different files are listed in Table 4.4. Moreover, SCTP increases the cwnd with the MTU value (in this case 1500 bytes), while TCP adds the sender MSS (in this case 1448 bytes). Thus, it might be possible that SCTP is even faster than TCP.
Table 4.4: Number of data packets for the file transfer if MSS-sized PDUs are sent.

<table>
<thead>
<tr>
<th>Filesize [bytes]</th>
<th>TCP packets</th>
<th>TCP packets</th>
<th>SCTP packets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSS 1448 bytes</td>
<td>MSS 1460 bytes</td>
<td>MSS 1452 bytes</td>
</tr>
<tr>
<td>1 KB</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>128 KB</td>
<td>91</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>256 KB</td>
<td>182</td>
<td>180</td>
<td>181</td>
</tr>
<tr>
<td>512 KB</td>
<td>363</td>
<td>360</td>
<td>362</td>
</tr>
<tr>
<td>1024 KB</td>
<td>725</td>
<td>719</td>
<td>723</td>
</tr>
<tr>
<td>2048 KB</td>
<td>1449</td>
<td>1437</td>
<td>1444</td>
</tr>
<tr>
<td>4096 KB</td>
<td>2897</td>
<td>2873</td>
<td>2888</td>
</tr>
<tr>
<td>8192 KB</td>
<td>5794</td>
<td>5746</td>
<td>5778</td>
</tr>
<tr>
<td>16384 KB</td>
<td>11586</td>
<td>11492</td>
<td>11555</td>
</tr>
</tbody>
</table>

Figure 4.5: Comparison of single file transfers with TCP and SCTP. The measurement results show only the time for the data reception on the receiver host. TCP and SCTP use a similar protocol configuration.

The results of the time measurement for the reception of a single file with TCP and SCTP are depicted in Figure 4.5. Note that the time for the setup/teardown procedure is not included in these results. The diagram shows only the time required for the data reception. However, the measurement results could easily be supplemented with the time
for the connection establishment. Usually, SCTP requires two RTTs (in this case 120 ms)
for the initiation, while TCP only needs one RTT (in this case 60 ms). These values could
be added to those in the diagram, since all time measurements start after the connection
has been established. Such an adjustment would change the results in favor of TCP. The
outcome would then show that all files are transferred faster with TCP and the difference
between SCTP and TCP would be greater, too.

The diagram in Figure 4.5 shows that small files can be received faster with TCP, but
larger files reach the endhost faster with SCTP.

One reason for this behavior is that TCP and SCTP use a different cwnd calculation,
which depends on two things: the acknowledgement procedure (determines the time steps)
and the values used for the increment. Theoretically, SCTP might almost always be faster
than TCP because larger values (MTU vs. MSS) are used for the cwnd adjustment. However,
a raise of cwnd happens more often for TCP than for SCTP in slow start mode. Only during
congestion avoidance, the same time steps (each RTT) are used for the cwnd adjustment.
This explains why the performance difference between TCP and SCTP changes with the
size of files that are sent. The cause for the poor SCTP performance thus lies in the initial
phase of the transfers (association setup and slow start).

The higher goodput rate of SCTP is a further reason that its performance improves
with increasing amounts of data. SCTP uses a larger MSS than TCP, so that fewer packets
are required for the file transfer. It should be noted that TCP might use a even larger MSS
than SCTP. However, this is not usual, because the timestamp option must be disabled in
such a case. Sending of less data packets is always advantageous, however, due to the little
difference between TCP’s and SCTP’s MSS, is also the effect of the higher SCTP goodput
little and only perceptible over a longer period.

Furthermore, it was observed that the outcome gets influenced by the application
implementation itself, although an identical one was used for TCP and SCTP. The transfer
of the 1 Kbyte file was analyzed in detail, because SCTP is for this test, in which only a
single data packet is sent, two RTTs slower. Two reasons were identified:

1. SCTP cannot send data together with connection establishment packets.
2. SCTP must wait until the SHUTDOWN chunk is received to stop the data reception.

TCP may bundle data with packets that are used for connection establishment. Such
a possibility is not yet implemented in Linux for SCTP. Thus, the SCTP application must
wait for one RTT after the association establishment until the first, and in this case the
only, data packet is received.

The other problem is the detection of the transmission end. The end of a file (EOF)
is not explicitly signaled by the used implementation and only recognizable by connection
termination. Since SCTP and TCP have used an almost identical implementation, lacking
specific SCTP function calls. TCP recognizes an EOF already within the byte-stream,
because an encoded EOF character is transmitted together with the data. SCTP works message based and requires a specific EOF signaling if multiple SCTP messages are used to send a single file. This explains why SCTP must wait one RTT longer than TCP to stop reading. With an appropriate SCTP implementation this problem can be solved. Note that all SCTP transfers of this experiment are affected from the mentioned additional RTTs. Therefore, some minor performance gains (one RTT) are still possible for SCTP.

The conclusion of this experiment is that using SCTP for transmissions with long duration might be advantageous. However, data transfers of short duration are generally carried out faster with TCP. The better performance of SCTP for large files is not sufficient to justify a complete migration from TCP to SCTP. Therefore, SCTP’s behavior in slow start phase must be improved. One possibility could be to disable the delay of SACKs.

**Experiment 2.2** The previous experiment was repeated with some changes. The delay of SCTP’s SACK was disabled and the message size was modified. The value of SCTP’s MSS (1452 bytes) was used for the new message size\(^9\). Furthermore, the automatic TCP buffer tuning was disabled, and the same manual adaptation of socket buffers was made for TCP and SCTP. The evolution of the congestion window was investigated for each test run with the self-implemented polling function at the sender host. In addition, a sequence analysis was carried out with wireshark.

![Graph](image)

**Figure 4.6:** Comparison of single file transfers with TCP and SCTP with optimizations.

\(^9\) This choice of message size is related to the gateway prototype, which uses exactly this message size for the data transmission.
Figure 4.7: Magnified view of Figure 4.6. Note that for files up to 750 kbytes only slow start is applied.

Figure 4.8: The evolution of cwnd in slow start for TCP and SCTP with default settings vs. adapted settings. The values were obtained by periodically polling the current cwnd value. The values shown are only a part of the entire slow start phase.
Window adjustments at runtime cause delays.

Figure 4.9: TCP time sequence graph: Transmission of a 128KB file. TCP data transfers are slower if the automatic buffer adjustment is applied because sending is sometimes delayed due to the receiver window adaptation at runtime.
Figure 4.10: Comparison of SCTP SACK procedure: SACK with standard delay vs. SACK without delay. The data transfer of a 128KB file is carried out faster if SACK delays are disabled; new data packet may be sent earlier.
Figures 4.6 and 4.7 show the time for the data reception of single files (experiment 2.1 and experiment 2.2). From the diagram it can be seen that TCP transmissions perform decisively better if socket buffers are manually adapted rather than by automatic buffer tuning. The reason for this improvement was investigated and it turned out that the automatic TCP window adjustment at runtime sometimes delays the sending of new data packets. The sequence analysis with wireshark revealed that this always occurs when window values change. An example is shown in Figure 4.9.

Furthermore, the measurement results show that transmissions with SCTP are faster if SACKs are not delayed. The reason is that more SACK packets are sent and thus the cwnd grows faster in slow start. A sequence analysis of SCTP’s SACK (with standard delay and without delay) is depicted in Figure 4.10. Further performance improvements might be possible for SCTP, as the used SCTP message size of 1452 bytes is not an optimal value to reduce system call overhead of data I/O operations. However, this was not further investigated, because the SCTP prototype uses only MSS-sized messages anyway.

The outcome in Figure 4.6 and Figure 4.7 show that data transmission times are different for the individual protocol configurations. They are mainly due to different transmission rates in slow start, which in turn are determined by the different cwnd values. The evolution of cwnd in slow start is shown in Figure 4.8; it explains the differences in performance of the individual protocol configurations.

The conclusion of this experiment is that disabling SACK delays may lead to faster data transmission, but with the disadvantage network traffic increases. Such an adjustment improves SCTP’s performance, but TCP still performs better if socket buffers were manually adjusted for both protocols. The experiments have revealed a decisive cause – the different cwnd calculation for TCP and SCTP – which is responsible for the fact that SCTP in many cases cannot compete with TCP. SCTP loses performance especially at the beginning of a data transmission. Larger amounts of data, however, might be transferred faster with SCTP. The measurements have shown that differences in performance exist, but they are not very large. Thus, the current SCTP performance could still be enough to observe improvements with the SCTP gateway under certain conditions. The next experiment shows the basic principle that is used for an acceleration of data transmissions with the gateway.

**Experiment 2.3** Two files were consecutively sent over a single connection/association. At first, a larger file was transmitted (16 Mbyte-file, to open the cwnd), followed by a differently sized file, such that the altered cwnd value (open cwnd) was used for the second data transfer. The time for the data reception of the second file was measured on the receiver host. It was expected that these files are transferred faster than in the previous experiment, where a transmission has always started in slow start. A similar principle is also used by some Internet applications for speeding up data transmissions, and the same applies for the gateway proposal, which tries to exploit this effect in a beneficial manner. These measurements served to estimate the achievable time gain if files of different size are transferred with an open cwnd.
The measurement results of SCTP data transfers that start in slow start and those that use an open cwnd are compared in Figure 4.11. Almost identical results were also achieved for TCP transmissions. The diagram shows that an enormous speedup is possible if the cwnd is already open – especially for a transmission of small files, where all data packets can be sent within a single RTT. The improvements were achieved because slow start is avoided. Transmission rates are limited by the cwnd and if the cwnd has a bigger value (already open), many more packets can be injected into the network, so that all files up to the maximum of cwnd may be sent in one RTT. All data transfers show a speedup because the available link capacity can be saturated immediately. However, the obtainable speedup decreases with increasing file size. The knowledge gained by the measurements is that the beneficial effect of this transfer mode is larger if many small files are continually transmitted, but only little for single and large data transfers. Note that the speedup is dependent on the BDP and might be even greater, but also smaller with other BDP values.

The conclusion is that the transmission of several files over a single connection or the reuse of connections can be used for speeding up data transfers. However, consecutive transmissions of several files over a single connection are in many cases avoided by TCP applications, because of the HOL-blocking that may occur. SCTP multi-streaming does not suffer from HOL-blocking and, thus it is possible to exploit the presented effect with an appropriate implementation.
4.2.3 Data Transfer Methods - TCP vs. SCTP Multi-streaming

Additional experiments were conducted with the self-developed software elements to investigate different methods for the transmission of multiple files. The obtained knowledge is briefly summarized in the following descriptions. The investigated cases were:

- the parallel transmission with individual connections/associations for each single file
- the transmission of multiple files over a single connection/association
- the transmission of multiple files in parallel with a single SCTP association and multi-streaming

The performed experiments lead to the conclusion that parallel connections are in the majority of cases the fastest way to send multiple files simultaneously. Due to the individual congestion control for each single connection the time to be spent in slow start is reduced. Thus, the available bandwidth is saturated in a shorter time than with a single TCP connection or a single SCTP association with multi-streaming. However, parallel connections cause some side-effects. One is particularly serious, namely that they can be overly aggressive against a single flow on the same end-to-end path. Further consequences are the additional network traffic caused by the connection setup/teardown and the increased demand of resources for several TCBs. Despite all these deficiencies, this data transfer method is the most commonly used.

The benefits of parallel transmission, however, are restricted, and the transfer of several files over a single connection can sometimes be faster. This is the case if several files are sent at various times or if very high latency prevails in networks. The general intention of a serial transfer is to save the time for the connection establishment and to exploit the altered cwnd value. The speedups that can be achieved with a connection reuse it was already shown in the previous experiments. However, a serial transfer is not always suitable because of HOL-blocking. Moreover, it is difficult to determine whether sending several files consecutively is more advantageous than using parallel transmissions, because many factors must be taken into account, such as the file size, path delays, link loss, frame size, just to name a few. A combination of the two mentioned transfer modes seems a good solution and is also used in practice, but the problem of HOL-blocking is still present. A better solution, therefore, might be the use of SCTP’s multi-streaming.

SCTP multi-streaming does not suffer from HOL-blocking and enables applications for parallel data transfers with a single association. However, practical experiments have shown that parallel transmissions with a single SCTP association and multi-streaming are not as fast as parallel transmissions with several connections/associations. If multiple files were sent simultaneously it takes longer with multi-streaming than with several parallel connections until the available link capacity is saturated. The difference is caused by the number of applied congestion control entities (one vs. n-times). Using a single congestion
control for parallel transmissions may also lead to a better network behavior in some cases. A big advantage is the faster reaction to emerging congestion because the transmission rate can be adjusted for all parallel transfers in one step. Thus, network load may be reduced faster. A further advantage is achievable for data transmissions that start at different times, but use the same end-to-end path. Due to multi-streaming subsequent transmissions may share the association and with it also the cwnd value. Consequently, a faster transmission is possible, because slow start gets avoided. Furthermore, using a single SCTP association is beneficial, since traffic for connection setup/teardown is reduced and resources for TCBs are saved.

The conclusion is that SCTP’s multi-streaming seems to be very well suited for implementing parallel transmission. However, this functionality will likely only be used if the unfairness of the bandwidth sharing (one SCTP association vs. N TCP connections) gets fixed. The same applies for the gateway solution. Two factors that could foster the usage of multi-streaming are:

- a new congestion control (e.g. a weighted congestion control).
- restricting the number of connections from the same source hosts.

The experiments in the next section were carried out in a network where no additional network traffic influences the SCTP association with multi-streaming. For a final deployment of the SCTP gateway the mentioned problems before must be taken into account.

### 4.3 SCTP Gateway Evaluation

This section presents the proof-of-concept for the proposed SCTP gateway solution in Chapter 3. Since the SCTP gateway is a kind of connection management – a mapping of TCP connections onto an SCTP association with multi-streaming – this approach was called *connection manager gateway (CMG)*. This term is used in the next descriptions and refers to the SCTP gateway implementation and/or its application.

Various experiments were carried out with the developed CMG prototype to show its practical application and to investigate possible performance improvements. The goal of these experiments was not only the demonstration of benefits, but a critical evaluation of pros and cons for the suitability of the CMG in practice. At first, a performance assessment of the CMG prototype was carried out by throughput measurements with iperf, which are explained in Section 4.3.1. In a second step (Section 4.3.2), the CMG was used to map TCP connections of standard web applications. The same SCTP and TCP versions as for the experiments in Section 4.2 were used for comparative measurements. These specific protocol versions allow a fair comparison and a better study of effects that emerge by application of the CMG. However, the measurement results in the previous sections have shown that a protocol switch from TCP to SCTP is fraught with disadvantages. Thus, it was to expect
that the application of the CMG could not only create benefits, and several aspects had to be taken into account for the performance evaluation. In this context two different states of the CMG were analyzed: the startup phase (worst case) and the operating status with an established CMG path (best case).

Due to the knowledge obtained in Section 4.2, it is to assume that a mapping of TCP connections onto an SCTP association with multi-streaming leads to slower transmission in the initial phase. A possible scenario is that no SCTP association is available and a new CMG path between the end hosts must be established first. The causes for the loss of efficiency in this phase are:

- the SCTP initiation procedure (4-way handshake without data bundling)
- the cwnd calculation in slow start (TCP is faster than SCTP)
- the additional TCP connection requests (from the request host to the CMG and from the second CMG to the end host)

In contrast to this, faster transmissions are to be expected if the SCTP association already exists and congestion avoidance is applied. The reasons for the speedup are specific SCTP properties:

- the congestion information sharing due to multi-streaming (open congestion window)
- the congestion window calculation (SCTP’s MTU vs. TCP’s sender MSS)
- the transmission of larger PDUs (if TCP has timestamps enabled)

The basic requirement for a performance gain with the CMG is a fast protocol change. This essentially depends on the CMG implementation itself and can also lead to a performance loss. In this context it must be noted that the used prototype implementation is far from optimal and only suitable for an initial investigation of the gateway concept. The purpose of this gateway prototype is to figure out key points for a proper implementation and to gain more knowledge about possible advantages and disadvantages.

Some basic configurations were carried out for the practical evaluation, and they are valid for all subsequent experiments. All measurements were carried out with the same network conditions: 100 Mbps and a constant RTT of 60 ms. The default system settings of TCP socket buffers and the CMG socket buffers were adjusted for the BDP (buffer sizes of 750000 bytes). The same protocol settings as for the measurements in the previous sections were used (Appendix A.4). One CMG application was started on each end host and the delay of SACKs was configured for the individual test case by passing an optional parameter to the CMG application. The original TCP client applications were redirected to the CMG with the help of the developed “connection redirection software”. The same TCP server application was used from original TCP and redirected TCP transmissions.
4.3.1 Throughput Measurements with Iperf

The first tests with the gateway prototype were carried out in combination with iperf. The throughput rates of redirected TCP transmissions over the CMG were compared with that of original TCP and SCTP. Moreover, the influence of SACK delays on the performance of the CMG was investigated. The experiments in Section 4.2.2 have shown that SCTP works faster if the delay of SCTP SACKs are disabled, and the same should apply for the CMG.

CMG experiment 1.1 The same iperf experiment as in Section 4.2.1 (experiment 1.2) was carried out: a single unidirectional data transmission with a duration of 10 seconds. CMG measurements were performed with enabled and with disabled SACK delay. Furthermore, two different states of the CMG were simulated: the startup phase (with slow start) and the established state (fully open cwnd). To investigate these situations, the CMG application was restarted for each test run and iperf applications were executed twice in succession. The first iperf measurement was the result for CMG at startup and the second for the CMG in established state. The first data transmission establishes the SCTP association and opens the congestion window (after approximately one second fully open). Due to the applied kernel patch, the cwnd value does not change in idle state until a Heartbeat-chunk is sent (default 30 sec). Thus, the second iperf transmission can reuse the already established SCTP association of the CMG and the altered cwnd should lead to a faster data transmission. In this specific case, the maximum performance gain which is possible with the CMG can be measured.

![Figure 4.12: Results of performance measurements with iperf for various configurations](image-url)
The throughput measurements of different protocol configuration are depicted in Figure 4.12. The diagram shows that the default TCP configuration (automatic socket buffer adjustment), the adapted TCP version (manual adapted socket buffers for the BDP) and SCTP have different throughput rates. This is mainly due to the different cwnd calculation in slow start which was already explained in Section 4.2.2. Furthermore, the measurements confirm that also CMG transmissions are faster if the delay of SACKs is disabled.

One measurement result is particularly striking; mapped TCP transmissions (CMG startup with SACK delay) are slightly faster than a direct transmission with the SCTP configuration of iperf and withsctp. Since the decisive difference between these two transmissions is the message size, this must be the reason for the different throughput rates. Unlike the direct SCTP transmission, which uses messages with a size of 8192 bytes, the CMG uses a message size of 1452 bytes (size of MSS). A closer examination led to the realization that MSS-sized messages cause less overhead than messages which are not multiples of SCTP's MSS. Moreover, the message size is responsible for a different cwnd calculation. This measurement result confirms a further presumption which was mentioned in previous sections: message sizes of SCTP have a significant impact on the performance.

Furthermore, the measurement outcome in the diagram shows that the adapted TCP version and the CMG configuration without delay of SACKs (in startup phase) have a very similar performance. Therefore, these configurations were used for the further experiments. However, it must be considered that throughput rates vary with the duration of data transmission. This measurement result shows that transmissions with TCP and with the CMG have an identical performance if they start from scratch. However, the outcome of the CMG at startup can be worse if the data transmission takes less time, because SCTP’s performance becomes better than that of TCP as the transmission proceeds. Performance losses at startup can therefore be compensated over time.

The most important detail of this experiment is that with the CMG larger throughput rates are achievable if the cwnd is open than with default TCP transmissions. Of course, TCP transfers with open cwnd would also show larger throughput rates, but a decisive difference between CMG transmissions and TCP transmissions exists. The gain in performance of the CMG can be exploited by parallel transmissions with multi-streaming.

Owing to the results of this experiment, the conclusion can be drawn that a protocol change as carried out by withsctp might not always be beneficial. It can happen that performance is lost by inappropriate SCTP message sizes. This can easily happen accidentally, if a protocol translation is carried out and the identical TCP implementation is used for SCTP without appropriate adaptations. Therefore, the protocol switch with the CMG seems to be a better alternative. Due to the measurement results obtained, it is to assume that the prototype implementation will suffice to demonstrate significant performance gains in the next experiments.
4.3.2 Data Transmission with Standard Applications

TCP connections of standard HTTP applications\(^{10}\) (web server and web client) were mapped with the CMG prototype to evaluate the practical application of this solution. In addition, time measurements were conducted for a performance assessment of the CMG. For this purpose, the time was measured which is required for the reception of a requested HTML-document\(^{11}\) with images. The use of web applications for these experiments was deliberately chosen, because they allow the analysis of various data transmission methods:

- the download of individual files one by one (each file uses a new connection)
- the serial download of multiple files with a single connection
- the download of multiple files with parallel connections

Due to the knowledge obtained by the previous experiment, it is to assume that beneficial effects of the CMG change with the transmission method, as the time spent in slow start is different for each method. The experiments in Sections 4.3.1 have shown that faster data transmissions with the SCTP gateway are only possible when the CMG path is already established. Moreover, it was shown that the throughput rates of SCTP are better than with TCP if larger files are transmitted. This leads to the conclusion that potential benefits of the CMG can vary and depends on two factors: the transmission time and the transmission method.

The practical evaluation of the CMG was carried out gradually. At first, the application of the CMG was studied in combination with the “firefox” web browser, where different web pages were loaded to verify the results obtained by the previous iperf measurements. In a second step, it was attempted to show that a faster data transmissions with the CMG is also feasible if files are sent in parallel. In the end, the previously mentioned transfer methods were applied for a download of several files with HTTP. Time measurements were carried out for transmissions with TCP and for redirected TCP transmissions over the CMG gateway. The CMG experiments were conducted for two different situations: the CMG in startup phase and the CMG with open cwnd. Due to the applied kernel patch it is possible to fully open the congestion window with a previous data transmission and then to reuse the open cwnd value for a subsequent transmission over the CMG. Note that this scenario represents the optimal case, namely that a transfer has taken place and the new transmission started when the other finished. The worst case of the CMG is if no SCTP association exists between the end hosts or if transmissions start from scratch. The measurement outcomes of the different transmission methods and the two extreme situations cover essential use cases, such that the general CMG behavior in specific situations can be estimated.

\(^{10}\) The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems – RFC 2616 \[72\]

\(^{11}\) The Hypertext Markup Language (HTML) is the predominant markup language for structured web content e.g. web pages.
CMG experiment 2.1 A web page with a single image file was downloaded from the web server. The “firefox” web browser was used as client application. The time required for the loading process (the overall download time for the HTML-file and the image file) was measured with a web browser extension (“firebug” add-on). The time measurement started with the request and stopped when the complete data was available. One measurement was made for the HTTP download with TCP and a second measurement was carried out with the CMG in established state. Therefore, an additional operation was performed before the measurement with the CMG was started. A data transmission with iperf was carried out (duration of 5 seconds in both directions) to establish the CMG path and to open the cwnd. After the data transfer stopped, the redirected HTTP download over the CMG was started. This experiment was repeated for different web pages (image files of different size). Specific settings of the web browser and web server, which could lead to an enhanced data transmission (e.g. caching, connection reuse, pipelining) were disabled. This will ensure that measurements are not influenced by other advantageous effects. The purpose of this experiment was to show that faster transmissions are always possible, regardless of the file size, if the CMG is in established state and altered cwnd values can be used.

The time measurements for loading a web page with a single image are depicted in Figure 4.13. The diagrams show the maximum time difference which is possible between HTTP downloads with TCP and mapped TCP transmissions (for this gateway prototype). The outcome is that all test cases are significantly faster if the CMG is used for the data transmission. This measurement verifies the effect observed by the previous throughput measurements with iperf; TCP transmissions are carried out faster if they are mapped to SCTP by the CMG. However, it is visible from the diagram that the performance difference for larger files becomes slightly smaller, although the opposite would expected due to the specific SCTP properties that were explained earlier. The reason for the performance loss is the introduced delay of the mapping process. Therefore, it is possible that for very large files, TCP is in turn faster than the transmission with the CMG. This problem can be avoided by an improved gateway implementation. For a proper implementation, a very effective mapping procedure is necessary, such that the extra time for connection mapping is kept as low as possible. Significant improvements could be achieved if the gateway would work in kernel space rather than in user space, since the many copy operations would then be less time expensive.

Note that this measurement result shows only possible improvements for a specific use case. The time differences of mapped TCP transmissions and default TCP transmissions can change with the BDP and with the number of connections. Even better results would be possible for data transmissions with the CMG if the BDP becomes larger. The contrary is the case for a smaller BDP. The influence of several connections was investigated with the next practical test.
Figure 4.13: Time for loading a web page with the firefox web browser: TCP transmission vs. mapped TCP. A single image file was loaded together with the HTML-page.
CMG experiment 2.2 A web page, which contains five images of same size, was downloaded from the web server. The image files were loaded in parallel. The time required for the complete data reception was measured with “firebug”. This experiment was repeated for different web pages (different file size of the images). The same procedure as for the previous experiment (CMG experiment 2.1) was conducted for the CMG measurement. The purpose of this experiment was to show that not only single data transmissions, but also parallel transfers are carried out faster with the CMG than with standard TCP. The time difference between TCP and mapped TCP transfers, however, should be less than for the previous experiment. Furthermore, it is expected that under certain conditions the CMG gets stuck due to an implementation error (retained aware). The error occurs only if parallel transmissions are carried out and at least one of the transmitted data files is larger than the TCP send buffer (750000 bytes) of the second CMG. The intention of this test is to provide a practical proof for the mentioned error case in Section 3.3.3.

The results of the time measurement for loading the content of a web page in parallel (HTML-document and five single images) are depicted in Figure 4.14. The outcome of this experiment was rather surprising. From the diagram it is visible that all transmissions which are redirected over the CMG are substantially slower than those with TCP. This is exactly the contrary of what was actually expected. The cause for the performance loss was analyzed, and it turned out that the redirection over the CMG delays the connection attempt. An example for the analysis of CMG transmissions and TCP transmission is shown in Figure 4.15. The problem is that the applied method for the connection redirection restricts the work flow of the web browser. Due to the modified call, connection requests are no more started in parallel, but they were executed serial. Only the data transmission is carried out in parallel. This behavior is visible in Figure 4.15 (1). Further details to this problem case are explained in Section 4.4.1.

The analysis also revealed that redirected data transmissions (which were started later) require much less time even if the same data amount has to be sent. This is shown in Figure 4.15 (2). The reason for this acceleration is the expected effect of the CMG. However, in the aggregate, this transmission acceleration is not enough to compensate the time lost at the connection request and explains why the CMG performance is so bad. The conclusion of this measurement is that also parallel data transmission can benefit from altered cwnd values, but the used redirection method must be revised, as otherwise it is not possible to benefit from the application of the CMG.

As expected, the measurements were only possible for certain file sizes. The CMG got stuck if files were sent which are larger than 2 Mbytes. The lack of a stream-based flow control caused a buffer overflow which in turn blocked the complete transmission. This proves that an additional flow control per stream is necessary if multi-streaming is used for parallel data transfers. Further details about this error case are described in Section 4.4.2.
Figure 4.14: Time for loading a web page with the firefox web browser: original TCP vs. mapped TCP transmissions. After the HTML-file is received, five equally large image files are downloaded in parallel.

TCP transmission

TCP transmission over SCTP gateway

Figure 4.15: Analysis of a parallel HTTP request with firebug: TCP transmission vs. redirected TCP transmission over the SCTP gateway. (1) time for the connection setup, (2) time for the data reception
The general applicability of the CMG was investigated by the previous experiments. They have shown that applications can be adversely affected by the implemented redirection method, so that the capabilities of the CMG can not be exploited in a satisfactory way. Therefore, another client application, the download tool “wget”, was used for the next experiments. This software is not affected by this issue and makes it possible to perform measurements with several HTTP transmission methods. Thus, the full potential of the CMG can be shown.

**CMG experiment 2.3** The “wget” tool was used to download a HTML document with five images of equal size from a web server. The experiment was repeated for different web pages (different image files) and with different CMG situations (startup and open cwnd). Three transmission methods were investigated: the serial download with a new connection for each single file (one by one), the download with a persistent connection (single connection which is reused for the subsequent transmissions) and a parallel download. The experiments were carried out with appropriate Python scripts for each single transmission method. The scripts implement a simple time measurement and contain the specific configuration of wget for the applied download method. As the wget software does not support parallel downloads, this transmission method was appropriately simulated. For this purpose, the Python script was extended to start a new process for each wget command (5 images and the HTML document). In this way all file downloads were started at the same time. This experiment should show advantages and disadvantages if the CMG is applied. It was to be expected that performance gains are achievable for all transmission methods. However, the benefits for the individual methods will vary as it depends on the time spent in slow start. The sequence is: serial (most benefit), with a persistent connection (medium benefit), parallel (least benefit). The contrary is the case for the CMG in startup phase.

The measurement results of the experiments are depicted in the following figures: the serial download in Figure 4.16, the serial download with a persistent connection in Figure 4.17 and the parallel download in Figure 4.18. All measurements showed the expected results, and thus all previous experiments are confirmed. The measurement outcome can be used to determine when the CMG should be used. Furthermore, the measurements highlight the performance problem that can arise through a protocol switch from TCP to SCTP (CMG in startup phase). The general problem of SCTP is certainly that it cannot compete with TCP in slow start. This shortcoming needs to be urgently addressed, as otherwise the use of SCTP instead of TCP will probably be avoided in many cases. Improvements of SCTP’s slow start would also have a big impact on the CMG and contribute to reducing the large performance loss at the startup phase of the CMG.
Figure 4.16: Time required for a serial download of five files with the same size.
Figure 4.17: Time required for a serial download of five files with the same size over the same connection (connection reuse).
Figure 4.18: Time required for a parallel download of five files with the same size.
Based on the results in Figure 4.16, it can be said that the use of CMG certainly yields great advantages if many transmissions are conducted consecutively. This is true even if the CMG needs to establish a new mapping path. However, some applications also reuse TCP connections for further transfers. In such a case, faster data transmissions are only possible if the CMG path is already established. If the CMG must start from scratch, performance losses are the consequence, as it is shown in Figure 4.17. The measurement outcome in Figure 4.18 has shown that also parallel transmissions can be accelerated with the CMG. However, if parallel transmissions are started simultaneously, and the CMG is in the startup phase, several TCP transmissions are always carried out faster than redirected transmissions over the CMG. Performance improvements for simultaneously started transfers are only achievable if the CMG path is established and the cwnd is open.

In summary, some basic rules must be considered for the use of the gateway. The application of the gateway is recommendable if the CMG path is used constantly and over a longer period, e.g., if long-term data transmissions take place or parallel data transmissions are conducted continually. In such a case all data transmission methods can benefit if the CMG is applied. In contrast to this, the use of the CMG should be avoided if only single and short (dependent on the BDP) and intermittent data transmissions are carried out. Performance losses would be the consequence, since SCTP is not yet able to compete with TCP transmissions in slow start phase. In this context it must also be noted that the CMG with the current SCTP version always introduces some performance losses at startup, which, however, can be compensated in the course of time.

The conclusion of these measurements is that the use of an SCTP gateway is practically feasible and faster transmissions are possible in many situations. However, disadvantageous cases exist that must be avoided and different problems must first be solved, so that a productive use in practice becomes recommendable. Important key points that must be taken into account were identified by the different experiments and were documented. Some problems that need to be solved in order to enable a general use of the CMG are described in the next section.

4.4 Discovered Problem Cases

Shortcomings were discovered by practical application of the CMG for which appropriate solutions must still be found. In Section 4.4.1 and Section 4.4.2, problems are described which are caused by a faulty implementation of the SCTP gateway prototype. The knowledge gained through the second implementation error (Section 4.4.2), however, is not only important for a proper gateway implementation. The detected problem case should always be taken into account if SCTP’s multi-streaming is used for parallel data transmissions. The last point, in Section 4.4.3, describes a general drawback that makes the deployment of SCTP applications difficult – the missing protocol support by middleboxes.
4.4.1 The Redirection Problem

As the experiments in the previous sections have shown, the developed prototype introduces in some cases additional delays at the connection initiation. This is caused by the implemented method which redirects TCP connections to the SCTP gateway. The gateway prototype requires additional information (original port and IP-address) which must be sent immediately after the TCP connection is established between the first gateway and the application. The connection information is used to create the new transmission path (TCP connections and SCTP association) for the protocol mapping. Therefore, the `connect()` function was modified and applications can make use of this new function call through “preloading”. The new implementation of the `connect()` call, however, blocks until the status of the complete CMG path (TCP-SCTP-SCTP-TCP path) is determined, such that in the case of non-availability a fall-back to original TCP can occur. This blocking is necessary, as otherwise applications can interpret the established TCP connection status in a wrong way. For example, it can happen that the socket status (TCP socket) of the redirected application is set to established, but this is only true for the connection from the TCP application to the first SCTP gateway. Usually, applications start to send immediately after an established connection is detected. If this would already happen before the complete CMG path is established, data could get lost and a fall-back to the original behavior would be impossible. Thus, the application could end up in an unsolvable fault condition. This problem case is avoided through the implemented blocking in `connect()` calls. However, the disadvantage is an additional introduced delay at the connection initiation.

Many applications implement non-blocking `connect()` function calls to optimize performance, since other operations can be carried out until the connection is established. This special case – a mapping of non-blocking connect calls – was not taken into account for the implementation of this prototype. An example for an application which suffers from this problem is the “firefox” web browser. The conclusion: the connection redirection must be adapted or a better solution must be found, such that all applications can use the gateway without additional delays.

4.4.2 The Flow Control Problem

The experiments have shown that the entire CMG application can get stuck when large amounts of data were transmitted in parallel. The reason for this is a faulty implementation style of SCTP’s multi-streaming. Already during the development it was evident that the implemented method for parallel transmissions with multi-streaming might theoretically lead to problems, which, however, was retained deliberately.

The intention of this improper implementation was to give a practical proof for the discussed problem scenario in the previous chapter. The practical tests have shown that the problem does not always occur, but as described in Section 3.3.3, errors occur only when TCP send buffers of the second gateway overflow and SCTP messages can no longer be
processed. The cause for this behavior is the lack of a stream-based flow control. The flow control of SCTP works only for the entire association but not for single streams. A solution for this problem was presented in Section 3.3.3 (dynamic intermediate buffer); however, it was not implemented, because it was suspected that the necessary additional copy operations in user space would make the mapping process too slow and performance gains by practical measurements could probably not be shown. Moreover, the solution with dynamic buffers is in this case not sufficient, because the amount of data to be sent can not be predicted. Therefore, buffers must be limited in size; otherwise, it might happen that the entire memory of the system will be used up. However, predefined buffers require additional mechanisms to regulate the data flow.

One possible solution would be to implement a simple start-stop method with message signaling for each stream. Of course, a better solution would be a sliding window mechanism for streams. Clearly, various solutions are possible, but how the final solution might finally look must still be specified. Our opinion is that not only a solution for this particular case (the SCTP gateway) should be developed, but a general one is required, such that also other applications could then use parallel data transmissions with multi-streaming and exploit the advantages of this method. A reasonable possibility would be to extend the SCTP protocol stack with additional stream-based flow control.

4.4.3 The Problem with Middleboxes

One common interpretation of the end-to-end principle is that protocol operations should occur at the endpoints of a communications system, or as close as possible to the controlled resources. The transport protocols work at the communication endpoints and intermediate routers should not be concerned with this layer’s functionality. However, such strict end-to-end transmissions are rare in today’s Internet e.g., network address and port translation \(^{12}\) (NAT and NAPT) is a commonly used method which breaks the rule of the end-to-end principle. The consequence is that changing transport protocols becomes difficult, because not every protocol is supported by middleboxes, and the same applies for SCTP. Consequently, the use of SCTP applications, and in this case also the application of the SCTP gateway, is limited or often simply not possible. This problem is well known and different approaches are already in development. In Section 2.4, two different solutions were presented: UDP encapsulation \([20]\) and an SCTP NAT implementation \([73]\).

UDP encapsulation is a simple method to circumvent the NAT problem, but with the disadvantage that overhead is introduced due to the additional header information. Therefore, this possibility should only be applied if no other alternatives exist. A better solution is an appropriate NAT and NAPT implementation for SCTP. The difficulty of SCTP NAT implementations is that the required functionality is fundamentally different

\(^{12}\) NAT and NAPT replace address information (IP addresses and ports) in data packets by others to share global IP addresses or simply hide address information.
The difficulties for the implementation of such a solution are explained in [40]; the main problem is the multi-homing feature of SCTP which must also be taken into account for a suitable solution. Note that the presented SCTP gateway could also easily be equipped with multi-homing. Appropriate solutions are then needed, such as for instance, the NAT implementation of [73]. The specification of SCTP NAT is still in draft (e.g. [19]) and not yet standardized. The conclusion is that the use of the SCTP gateway solution is only to a limited extent possible if middleboxes do not support SCTP NAT. Initial approaches to solve the NAT problem, however, are already available, so that a future application seems possible.

4.5 Summary and Conclusion

In this chapter, the performance of SCTP was examined closely, and compared with TCP. For this purpose, several practical experiments were conducted in a testbed network. Adjustments of protocol-specific parameters are required for a fair comparison and for a good performance. The appropriate parameter settings and configurations were explained in Section 4.1 and they must be considered for the interpretation of comparative performance measurements with TCP and SCTP.

The first experiments, in Section 4.2, had two objectives: one was to find suitable protocol versions of SCTP and TCP for a fair comparison, and the other one was to use these versions for a practical performance assessment of SCTP and TCP. The experiments have shown that in the initial phase of a data transmission is TCP faster than SCTP, even if similar protocol versions are used. The cause for this is mainly due to the different computation of the congestion window for TCP and SCTP. However, SCTP can compensate this disadvantage if data transmissions take place over a longer period. Furthermore, it was suspected that the delay of SACKs might reduce SCTP’s performance, and this was therefore disabled. The practical measurements have shown that substantial performance gains are possible if SACK packets are not delayed. Despite this modification, TCP was slightly faster than SCTP for the conducted test. The performance-optimized version of SCTP was used for further measurements with the gateway prototype. The conclusion of the preliminary experiments was that several improvements must still be carried out for SCTP, such that a completely migration from TCP to SCTP is reasonable. Both protocols have advantages and disadvantages so that it depends on the situation which protocol is more suitable. Therefore, to bind applications to a specific protocol does not seem very useful and a better solution is to switch the protocol as needed. The SCTP gateway concept represents one potential solution.

The idea with the SCTP gateway was not only to enable a protocol switch, but to speed up transmissions due to multi-streaming, which allows sharing congestion control information for parallel transmissions. The possible performance improvements were investigated by practical experiments with a rudimentary prototype implementation. The measurement
results were presented in Section 4.3. The intended proof-of-concept was carried out and
the theoretical benefits, mentioned in Chapter 3, were confirmed by practical measurements.
The experiments have shown that data transmissions can be accelerated if the gateway ap-
lication uses an established path. However, performance losses in the startup phase must
be considered for mapped TCP transmissions due to the limited performance of SCTP in
slow start. The conclusion is that the presented idea is promising as the protocol mapping
is feasible in practice, and performance gains are achievable in many situations. These were,
however, only the first steps and some problems must be solved for a productive use in
practice. These problems were explained in Section 4.4. They determine what further steps
should look like.
Chapter 5

Related Work and Open Issues

The thesis was inspired by many scientific papers which are briefly described in Section 5.1. The summary of these documents gives an insight concerning demanded improvements at the transport layer.

The future development of SCTP is discussed in Section 5.2, where some open issues are presented regarding the SCTP standard specification and its implementation in Linux. The first two points describe proposals for implementation adaptations of SCTP’s association setup procedure. Furthermore, two functionalities are pointed out, which are required for the proper implementation of the gateway solution.

5.1 Different Approaches

Many scientists are engaged in finding solutions that allow a simple change of protocols at the transport layer. As presented in Chapter 3, different possibilities exist to deploy new protocols alongside another one. One strategy is to carry out a protocol replacement with an appropriate software, like the “shim-layer” [2] or the “withsctp” software [1]. Alternatively, an application-specific reprogramming is possible.

The rewriting of HTTP services for supporting SCTP is described in [57], on the grounds that SCTP features better match HTTP needs than TCP. In [43], it is explained how FTP applications can be improved with SCTP’s multi-streaming. Both approaches have the same goal, namely to achieve an enhanced data transmission via the advanced SCTP features. However (in our opinion), the advantages mentioned by the authors are not only applicable to single services, like HTTP or FTP, but SCTP features show to be well-designed for an application in a general manner, such that almost every Internet service can be improved by an appropriate SCTP network service.

The exchange of transport protocols is a big challenge, because many new problems arise therewith. The preservation of former protocols (e.g. TCP) for compatibility reasons is one of the main problems and already starts with the connection setup. The authors
of [56] describe recommendations to allow a smooth transition for HTTP services from IPv4 to IPv6 and from TCP to SCTP. The cited draft describes a procedure for a simultaneous initiation of TCP and SCTP, which was used for the second proposal in Chapter 3. The algorithm works, but pros and cons of this setup procedure still need to be investigated in detail. Two other possibilities are proposed by the authors of [57], namely the usage of SRV records\(^1\) or a special URI scheme\(^2\) for SCTP services. A completely different approach is proposed by the authors of [74]. They describe a connection-establishment method with a bundled request for several transport protocols (not only SCTP) – a so-called “Meta-SYN”. Along with this proposal the reasons for the difficulties of protocol changes at the transport layer are explained, and as solution, a new transport service architecture is proposed. The authors of [74] describe in [75] in detail the intended transport architecture changes as well as improvements that can be achieved with it\(^3\). The same authors describe in [76] an enhanced protocol negotiation procedure at the connection initiation, which is based on the principles mentioned in [74]. They propose for the determination of applicable protocols an intersection of available protocols, which is formed on both end hosts by further taking into account the protocol support of middleware boxes. Furthermore, the authors compare the previous mentioned setup procedures in [57] and in [56] with each other. The outcome (described in [76]) is that these approaches (dual setup, SRV records, URI schema) are not really appropriate due to the low scalability; the proposed “cross-layer” negotiation would be a reasonable alternative.

In connection with the ideas described in previous-mentioned papers, the structured stream transport (SST)[77] protocol was drafted (by Bryan Ford, author of [76] and [74]). The SST protocol is designed to address the needs of modern applications that handle many communication activities in parallel. SST works, like SCTP, with independent streams to implement parallel data flows within a single connection. However, some decisive differences exist between SCTP and SST. A new feature of SST is the integrated stream-flow prioritization. Another special peculiarity of SST is that the UDP protocol is used for the actual implementation of SST. The available protocol implementation works at application layer in user mode and uses the underlying UDP service. Thus, the deployment of SST in the Internet is easier than for SCTP, because almost every middlebox supports UDP, but not always SCTP. However, SCTP has a big advantage over SST, namely of being already specified in RFCs, whereas SST is not. Protocol extensions or an appropriate implementation style may equip SCTP applications with similar functionalities as SST offers. Moreover, SCTP has at least one important feature, namely multi-homing, which is still missing in SST. All these points would be cumbersome for the standardization of SST by the IETF (in addition to the general difficulty of introducing new protocols).

---

\(^1\) A service resource records (SRV) is a specific information retrievable by DNS. For each service, further information will be provided, such as the server name and the protocol (e.g. \_ldap\_tcp.example.com).

\(^2\) A Uniform Resource Identifier (URI) is an identifier and consists of a string that serves to identify resources (such as files or web services), e.g. for SCTP “http-sctp://”.

\(^3\) Some of the goals mentioned in [74] match the goals of the thesis presented in Chapter 2.
Both, SCTP and SST, seem to be appropriate for implementing the new layer model in [75] or for the development of a general fairness-enhancing middleware on the Internet. The possibility to use single connections/associations for parallel transmissions with stream flow-priorities is a great help for implementing “congestion manager” properties (specified in RFC 3124 [50]). The goal of the congestion manager [78] is fair congestion control management and bandwidth sharing, independent of specific transport protocols and applications. The implementation of the congestion manager software is complex and in our opinion more likely to implement by using a suitable protocol, in order to manage all types of transmissions (e.g. with SCTP). The main idea in this thesis was to show that parallel data transfers can be accelerated if the congestion control values are shared; this is simply through an appropriate use of SCTP’s multi-streaming possible. A similar but more complex implementation method for TCP transmissions is described in RFC 2140 [51]. Ideas of the mentioned documents were considered and have fundamentally influenced the gateway prototype design. However, many open issues need to be clarified before the further development steps for the SCTP gateway (presented in this thesis) can be specified. This will also depend on the decisions taken by the IETF regarding presented drafts in this section.

5.2 Open Issues

During the implementation of the gateway solution, some cases were discovered that cannot (or hardly) be implemented with the existing SCTP network stack. They are explained in the following sections and lead to a suggestion of reprogramming the protocol stack or also possibly extending the SCTP standard.

5.2.1 Data Chunk Bundling at Association Startup

The SCTP specification in RFC4960 [10] allows to bundle data with the COOKIE-ECHO and COOKIE-ACK chunks, but this is an implementation decision and not yet implemented in LKSCTP. Such a possibility is beneficial, especially if small files are sent in networks with high path delays. Achievable advantages through a data bundling at startup vary and are dependent on the used link media (frame size), path delay and file size. Short data transfers are generally carried out faster with TCP than with SCTP, for two reasons: first, TCP uses a less time-consuming connection setup procedure than SCTP and second, TCP has already integrated data bundling at startup (data can be sent together with the SYN/ACK packet).

In order to promote the deployment of SCTP for short data transmissions, it is recommendable to implement the bundling of data chunks at SCTP association startup, but with an optional choice for using it. The reason for the option is that a chunk bundling at startup is not always beneficial; in fact the contrary can sometimes be a better choice. An example is mentioned in [13, p. 91], where it is stated that data bundling at the association setup should be avoided if the network is congested.
5.2.2 Earlier Delivery of an Adaptation Indication

RFC 5061 [17] specifies the adaptation indication. This value is exchanged at startup between end hosts and signals if one or more specific layers below the application are additionally used. The indication message is used to enable the additional required functionalities. The problem is that connect-calls block further application configurations until the association is established and adaptations can be carried out only after the association is established. Delays are the consequence, which are caused by the 4-way handshake (especially in networks with high path delays). The solution might be using a non-blocking connect-call, so that a faster adaptation can occur. As the indication values are exchanged with the INIT and INIT-ACK chunks, necessary configurations signaled by the adaptation indication could be carried out in the meantime, until the association is completely established.

However, such an implementation is not possible with the current LKSCTP implementation. Notification values are available to the application only after the association is completely established (after the 4-way handshake). The proposal is to make the adaptation indication values available immediately after the receipt of an INIT and INIT-ACK chunk. Together with the non-blocking connect-call and due to the earlier provided information, a faster adaptation (after one RTT instead of two RTTs) is possible.

5.2.3 Stream-Based Flow Control

A peculiarity of the current SCTP network stack implementation is that all messages are stored in one common receive buffer (draft specification in [18]). Therefore, different traffic streams are mixed at the endpoint and a blocking of message delivery from different streams can occur. Multi-streaming avoids head-of-line blocking, but the stream concept with a single receive buffer introduces new difficulties. A special way of implementation must be applied in order to avoid that data flows from different streams are blocked at the receiver side. This problem was already discussed in Chapter 3, where a solution was also presented for parallel transmissions with a single SCTP association and multi-streaming.

The problem of this solution is the performance loss which is caused by context switches and additional memory copy operations in user space. A much easier and better solution would be a stream-based flow control provided by the protocol stack. An approach to manage SCTP streams with a flow control mechanism for each stream is described in [79] along with reasonable use cases. The extension of SCTP by a stream-based flow control was discussed extensively in the IETF, but rejected. Our interpretation of the mailing list comments (TSVWG mailing list) is that the contributors to this discussion know about use cases which require such a property, but as it seems, they do not suffice to justify such an effortful standard modification. Therefore, IETF TSVWG participants recommended using proper implementations to match the requirements.
5.2.4 Stream Priorities

Data transmissions that are carried out with SCTP’s multi-streaming are subject to various adverse side-effects; one is mentioned by the authors of [57]. They recommend message size adaptations at the application layer to avoid dependencies, otherwise created by the fragmentation and reassembly algorithm of SCTP messages. It can happen that a bigger message (e.g. 1 Mbyte) blocks the transmission of a shorter message (e.g. 100 bytes) because the message is first fragmented and then all parts are sent in one step. There are no specific recommendations on how the SCTP stack should behave in such a case. Another problem is that each single message (passed from the application layer) must first be read from the shared socket buffer at the transport layer for further processing. Thus it is possible that parallel messages of different streams influence each other due to the socket buffer sharing. An orderly allocation of this resource would be desirable, but specifications in this context are missing.

Furthermore, multi-streaming in combination with parallel transmissions is dependent on the operating system scheduling. Single data flows are therefore not really predictable in their behavior and time estimation per flow is difficult, without using self-implemented scheduling or priority mechanisms. In this context it should also be noted that some applications could have a benefit from grouping streams into priority classes and prioritizing flows among these classes. Stream priorities, as described in [80], are a possible solution, but until now such an SCTP extension is not available. The reasons are probably the same as for the stream-based flow control.

5.3 Summary and Conclusion

Different approaches for a better handling of different transport protocols are in development and in discussion in the IETF. An insight was given in this chapter and some important activities were presented that will influence the SCTP deployment just as well. The related work section and the mentioned open issues show that SCTP is still in evolution. Various application fields depend strongly on the future development of the presented works. In addition, many open issues still need to be clarified. The outcome of discussions concerning the proposed specification changes will have a big impact on the proposed gateway solution and its final implementation.
Chapter 6

Conclusion and Outlook

The Internet is one of the most important communication media nowadays. The increasing areas of today’s Internet services are sometimes constrained by the little flexibility of the underlying protocols, which were created almost 30 years ago. There is a high demand for new and more appropriate protocols, like SCTP, which aims to replace TCP as the Internet’s dominant transport protocol. However, the change of transport protocols is a major challenge.

6.1 Conclusion

SCTP has several features, like multi-streaming and multi-homing, which provide enhanced capabilities for improving data transmission in networks. Despite the advanced features, SCTP is rarely used on the Internet. Investigations revealed that only a handful of applications are rewritten for the protocol support. Several reasons for the little use of SCTP were identified in this work. The main reason is SCTP’s limitation due to the lack of support by middleboxes, which makes the use of it in the Internet in many cases impossible. Another one is that advantages of SCTP features are not always sufficient to justify a complete migration from TCP to SCTP. Sometimes data transmissions are still carried out faster with TCP. As the effort for reprogramming is high and achievable benefits vary from application to application, it is highly unlikely that all existing applications will be rewritten for SCTP support. So, other incentives must be found for the deployment of SCTP.

An alternative to reprogramming is a general protocol translation, which allows to easily switch between transport protocols, such as for instance between TCP and SCTP. The protocol change must occur transparently and without limiting the functionality of existing applications. A couple of tools already exist that allow a smooth transition from TCP to SCTP without the necessity of rewriting applications. However, these approaches have some shortcomings; they enable only the advantageous use of SCTP’s multi-homing. The goal of this thesis was to demonstrate that another valuable feature of SCTP, namely
multi-streaming, can be applied in a very beneficial way. The thought was that an appropriate application of SCTP’s multi-streaming might sometimes lead to an improved data transmission irrespective of the application.

The idea was to use a single SCTP association with multi-streaming for parallel data transmissions that share the same end-to-end path, in order to achieve an improved transmission behavior. In general, parallel TCP connections are used for the transfer of several files, such that the HOL-blocking of TCP is avoided. However, an unfair network behavior is the consequence, due to the individual congestion control for each single connection. The SCTP multi-streaming feature was especially designed to avoid HOL-blocking, and a parallel transmission with a single association is possible. The task was to find a solution, so that individual TCP connections can transparently be replaced with a shared SCTP association. In this context existing protocol translator software was analyzed in order to figure out if these tools could be modified for such an implementation. However, no suitable solution could be found and a new approach was developed.

A reasonable implementation seemed to us an SCTP gateway, which transparently maps TCP connections onto an SCTP association with multi-streaming. Similar principles as by Performance Enhancing Proxies (PEP)\(^1\) for split connection implementations were applied in combination with multi-streaming. Through the gateway application, faster data transfers are possible, because subsequent data flows benefit from the adapted congestion window value. The proposed gateway solution was described in detail and a first prototype was implemented. With the help of a software element that modifies socket API functions, TCP connections were redirected to the SCTP gateway. The gateway implements the mapping process – each TCP connection gets mapped onto an SCTP stream. The actual application remains unchanged and does not have to be rewritten for using the SCTP gateway. Furthermore, a fall-back to the original functionality was preserved. Several experiments were carried out to obtain a current SCTP performance estimation in comparison to TCP. It turned out that SCTP is only partially able to keep up with TCP. However, the application of the SCTP gateway makes it possible that in certain cases a much faster transmission than with standard TCP occurs.

The proof-of-concept was given with a prototype implementation and its application under simulated network conditions. The experiments demonstrated that a mapping from TCP to SCTP with multi-streaming is not only theoretically beneficial, but also applicable in practice. However, many improvements must still be carried out to reach the ultimate goal – an appropriate gateway implementation for a productive use in communication systems. Further development steps regarding the gateway will also depend on the outcome of presented approaches from other researchers, which were explained in the related work section. Open issues need to be discussed and clarified, so that the next implementation steps can follow.

\(^{1}\) PEPs are used to improve the performance of the Internet protocols on network paths where native performance suffers due to characteristics of a link or subnetwork on the path – RFC 3135 [81].
The knowledge we have gained from this work was that SCTP is probably better suited for a general application than for specific applications. SCTP seems to be well-suited for the implementation of a congestion management, like it is described in RFC3124 [50]. The implementation of the congestion manager (RFC3124 [50]) and/or TCP Control Block Interdependence (RFC2140 [51]) is a difficult matter and the intended behavior may be better achieved with the help of SCTP. Through an appropriate combination of the principles of PEP [81] and SCTP, it might be possible to improve the network behavior in a similar way as intended by the mentioned RFCs (RFC3124 [50] and RFC2140 [51]). The consequence would be a fairer network bandwidth sharing for parallel transmissions as well as faster data transfers in certain cases. The first steps to explore prospects of using SCTP for this purpose were made with the presented gateway approach in this thesis. We believe that, as demonstrated in this work, one of the most interesting perspectives of SCTP is the use of this protocol for the implementation of a service that provides an enhanced network behavior in the Internet.

6.2 Outlook

The future deployment of SCTP and the use of the presented gateway solution will depend on various improvements that need to be made.

SCTP suffers from a very serious problem that prevents its deployment on the Internet – the general lack of support by middleboxes. This is the first problem that must be solved in order to promote the use of SCTP. Approaches are in development and currently under discussion in the IETF. The outcome will specify how the general support of SCTP by middleboxes has to be implemented. However, the next problem is that such a specification even gets deployed. In order to promote this, a sufficient number of SCTP applications must already be available. Otherwise, there is again a danger that middlebox-developers are not interested in implementing SCTP support. The presented gateway solution might certainly create an incentive to support SCTP in middleboxes, as benefits are achievable for a wide range of applications.

In addition to the mentioned problem with middleboxes, some other matters prevent the use of SCTP. The experiments in this work have shown that SCTP can not yet compete with the performance of TCP. Therefore, most existing applications will also continue to work with TCP and new applications will not always be equipped with an SCTP implementation or with support for both transport protocols. Possibilities to improve the performance of SCTP were identified, so that data transfers with SCTP can run almost as fast as with TCP. Some of them are:

- the modification of the cwnd calculation in slow start and data bundling at startup to improve short transmissions.
- the implementation of an automatic socket buffer adaptation.
• the optional choice of several congestion control algorithms for SCTP, similar to TCP.

The mentioned points could significantly contribute to improve the SCTP performance. Consequently, the use of SCTP may be reasonable for several types of applications. Moreover, these improvements are also essential for the presented SCTP gateway concept.

During the practical experiments with the SCTP gateway prototype various deficiencies were identified, which for a productive implementation still must be solved. Two general problems are:

• the connection redirection – to use the SCTP gateway.

• the flow control of parallel streams – to avoid blocking scenarios.

The practical application of the gateway prototype has shown that the proposed redirection method is only suitable to a limited extent, and a better solution is required. Furthermore, a stream control mechanism must be developed such that independent parallel transmissions over a single SCTP association are feasible. One possibility would be a stream-based flow control provided by the SCTP protocol. As it seems that this will probably not be implemented, other solutions must be developed. An alternative was presented in this work, namely the use of buffers which dynamically change their size. However, this solution is only suitable in specific cases and a more general solution is required. The solution finding for the mentioned problem cases are therefore the next steps for a further development of the presented SCTP gateway in this thesis.
Appendix A

Testbed Details

The results in this thesis were obtained by experimental analysis in a typical small office or home (SOHO) network. The hardware and software used for the testbed configuration are listed in the following. The network structure and applied modifications of default protocol parameter settings in Linux are briefly explained.

A.1 Network Structure

The network configuration used for the practical measurements is shown in Figure A.1. The connection to the Internet was disabled to assure that the measurements are not influenced by other data transmission. This way it is guaranteed that no background noise influences the measurements (assuming that the hosts have no additional special services running). The figure shows an optional extension which could be used for experiments with SCTP multi-homing. However, the multi-homing configuration was not used for the practical experiments. The structure of the testbed is formed by two PCs with the same configuration. In addition, one PC was configured as a Linux router.

Figure A.1: SOHO testbed network structure
A.2 Hardware

The following hardware components, network interfaces and network devices were used for the construction of the test network.

**Switch** Netgear Prosafe 5-port Fast Ethernet Switch FS105 (2x)

**Network interface A** Broadcom Corporation BCM4401-B0 100Base-TX (2x)

**Network interface B** Realtek RTL-8139/8139C/8139C+ 100Base-TX (2x)

**Network interface C** 3com Corporation 3c905C-TX/TX-M [Tornado] 100Base-TX (3x)

**Additional router** DG834GT 108 Mbps Wireless ADSL Router (Router, Switch, AP)

<table>
<thead>
<tr>
<th>Linux router</th>
<th>Hardware Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>AMD Sempron 2800+ Singlecore</td>
</tr>
<tr>
<td>Memory</td>
<td>256MB DDR-333 166MHz [2 x 128MB]</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>100 GB IDE 7200 rpm - 2MB Cache</td>
</tr>
<tr>
<td>Network cards</td>
<td>3 network interfaces type C</td>
</tr>
</tbody>
</table>

*Table A.1: Hardware details: Linux router*

<table>
<thead>
<tr>
<th>Linux host</th>
<th>Hardware Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Athlon64 X2 5000+ Dualcore - chipset NForce 430</td>
</tr>
<tr>
<td>Memory</td>
<td>2GB Dual Channel DDR2 677MHz [2 x 1GB]</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>320GB S-ATA 7200rpm - 32MB Cache</td>
</tr>
<tr>
<td>Network cards</td>
<td>1 network interfaces type A and 1 network interfaces type B</td>
</tr>
</tbody>
</table>

*Table A.2: Hardware details: Linux host*

Note that the used hardware parts have an influence on the outcome of measurements!
A.3 Software

The Linux operating system Ubuntu [66] was used for the basic configuration of all hosts. In addition, the following software was installed on all hosts which was used for the practical experiments. All installed programs used the default settings. This detail is particularly important for the router configuration.

Path delays were emulated for the practical measurements in the 100 Mbps network. Therefore, the size of the router queue must be large enough, such that packets can be stored for the emulation of delays. The default settings of the network emulator for the router queue (FIFO) are 1000 packets. This value was large enough for all simulated path delays (10 - 200 ms) and used for all practical experiments in this thesis.

Operating system Ubuntu 9.04 - Karmic Koala
Kernel version 2.6.31
Network emulator netem kernel component (already enabled in the kernel)
LKSCTP version lksctp-tools 1.0.9
Network analyzer wireshark 1.2.2
Network measurement tool iperf 2.0.4
Webbrowser Mozilla Firefox 3.5.6
Webbrowser Add-on Firebug 1.4.5
Webserver Apache 2.2.12
Network downloader wget 1.11.4

In addition to the mentioned software, some self-developed programs were used for the practical experiments. The corresponding software elements consist of three parts:

- various data transfer elements for SCTP and TCP with a time measurement for the data reception.
- the connection manager gateway implementation (CMG) with the redirection software.
- several scripts for the system configuration (shell script) and for some practical measurements (Python scripts)

The complete source code is “Open Source” and can be downloaded from the following web address: http://flori.bz.it/sctp. Moreover, it should be noted that a kernel patch was applied. The kernel patch and a patched Ubuntu kernel package (dep-package) can also be downloaded from this website.
A.4 Setting of Protocol Parameters

For comparative measurements with TCP and SCTP, protocol default settings of the operating system were adjusted so that a better comparison between these protocols was possible. For this purpose, the default settings of the used Linux operating system were modified appropriately. The protocol parameters were set to the values shown in Listing A.1. With this configuration, a very similar operation of TCP and SCTP is guaranteed.

```
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.core.rmem_default = 114688
net.core.wmem_default = 114688
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65538 16777216
net.ipv4.tcp_sack = 1
net.ipv4.tcp_dsack = 1
net.ipv4.tcp_no_metrics_save = 1
net.ipv4.tcp_adv_win_scale = 1
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_congestion_control = reno
```

**Listing A.1:** Configuration of system values for practical measurements

Note that system values can only be modified with corresponding user access rights. The example in Listing A.2 shows how system values can be set with the `sysctl` command. A shell script was used to set and to check the various system values for all experiments. The corresponding shell script can also be downloaded from [http://flori.bz.it/sctp](http://flori.bz.it/sctp).

```
# Network metrics are not recorded for the optimization of TCP
sudo sysctl -w net.ipv4.tcp_no_metrics_save=1
# Setting the socket buffer default values
sudo sysctl -w net.ipv4.tcp_rmem="4096 750000 16777216"
sudo sysctl -w net.ipv4.tcp_wmem="4096 750000 16777216"
```

**Listing A.2:** Example of system values setting with `sysctl`
Bibliography


[13] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP) - A Reference Guide*. Addison-Wesley, 2002. 2.2.1, 2.2.1, 2.3.4, 3.1.1, 3.3.3, 5.2.1


[31] Christoph Sereinig. Speeding up Java Web Applications and Web Service calls with SCTP. Master’s thesis, University of Innsbruck, 2008. 2.3.4, 4.1.2


133


[37] A. Caro, P. Amer, and R. Stewart. Retransmission Policies for Multihomed Transport Protocols. 2.3.5

[38] J. Iyengar and P. Amer and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. IEEE/ACM Trans on Networking, December 2006. 2.3.5


[45] PokerTH website. URL http://www.pokerth.net/. 3.1.1


[55] Helmut Herold. Linux/Unix Systemprogrammierung 3., aktualisierte. Addison-Wesley, 2004. 3.3.1, 3.5, 4.1.3


[58] Patent application title: Optimizing TCP traffic via an SCTP association. URL http://www.faqs.org/patents/app/20080205445. 3.4.1


[67] IPERF. URL http://iperf.sourceforge.net/. 4.1.4


[70] Firebug, . URL http://getfirebug.com/. 4.1.4

[71] GNU Wget. URL http://www.gnu.org/software/wget. 4.1.4


[79] Gerhard Rufa. Developments in Telecommunications - With a Focus on SS7 Network Reliability. Springer-Verlag, 2008. 5.2.3


Acknowledgements

I would like to thank my supervisor Dr. Michael Welzl for the guidance, assistance and patience through the thesis writing process.

Special thanks go to Vlad Yasevich for his support with the many questions concerning the LKSCTP implementation. In this context I would like to thank him for the provided Linux kernel patch that was applied for the practical measurements. I would also like to thank Randall Stewart and Michael Tuxen for the numerous e-mail discussions by which open issues regarding the SCTP socket API and various other related SCTP issues (e.g. SCTP NAT, Stream Reset) were clarified. Furthermore, I would like to thank Ryan Bickhart, Dan Wing, Preethi Natarajan and all members of the lksctp and tsvwg mailing lists, who have contributed via their given comments. The proofreader of this thesis, Mattia Rossi and Baur Helene, deserve a special thanks; they helped me with the English correction.

Last but certainly not least, I would like to thank my fellow students at the University in Innsbruck, Armin, Martin K., Martin R., Mattia and Peter, for the great study time spent together.

Finally, I would especially like to thank my parents, for taking interest in my study and supporting me along the way through graduate school.
Certificate of
authorship/originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Florian Niederbacher, Innsbruck on the 25. January 2010