Speeding up Java Web Applications and Web Service calls with SCTP

Master Thesis
Winter Term 2007/2008

Supervisor: Dr. Michael Welzl
Research group: Distributed and Parallel Systems

Author: Ing. Christoph Sereinig Bakk.techn
christoph.sereinig@student.uibk.ac.at

written in \LaTeX
April 15, 2008
Abstract

The Stream Control Transmission Protocol (SCTP) is still a relatively new general purpose end-to-end message transportation service for IP based networks. SCTP offers reliability, flow control, ordered delivery and full-duplex data transfer like TCP. However, SCTP also provides some other features unavailable in the current transport protocols of the Internet, like multi-homing and multi-streaming. It also removes some restrictions, such as strict byte ordered delivery.

Nevertheless SCTP has not yet been widely deployed due the relative small range of products and applications on the software sector.

This thesis focuses on speeding up Java web applications and web service calls by completely avoiding head of line blocking. SCTP seems to be a perfect candidate for this requirement through the features multi-streaming and unordered delivery. We have designed an SCTP Java library which implements a translation from TCP to SCTP, to have a simple but powerful tool to convert existing Java based TCP applications into Java based SCTP applications. For data transmission we are combining SCTPs advanced features multi-streaming and unordered delivery to completely remove head of line blocking. Due to the usage of the abstract factory method design pattern many legacy TCP-based applications will now have the opportunity to take advantage of SCTPs advanced features by changing only a few lines in their source code.

The performance of SCTP is naturally a key factor to SCTPs future as a general transport protocol. One of the cornerstones of this thesis is to solve the potential performance problems for applications which consist of a large number of independent transactions by using the features multi-streaming and unordered data delivery of SCTP. Within this thesis we will also present results of performance measurements of TCP an SCTP. We will demonstrate that using this library in an application gives it a performance gain in most cases, and keeps an equivalent performance in the other cases. A nice sideeffect of this library is that the developer and user now have a
tool to try out and take advantage of the features of SCTP without rewriting the whole source code.
Acknowledgments

Yes, while I’m writing this page i finally realize that the hard work during the last couple of years seems to have an end. This page is intended for thanking all those people that helped me to become a graduate engineer. I would like to start by thanking my supervisor Dr. Michael Welzl for the opportunity to write my master thesis at his group and his endless support, help and guidance during the development of this thesis. While I was able to manage my time schedule and project individually, he was always available and willing to help in any way. I would also like to thank Kassian Plankensteiner for having many insightful conversations, assists and proof reading. Without your help, I probably would have never finished my studies. And last but not least, I would like to thank my girlfriend Katrin for having so much patience with me during my studies.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Internet Protocol (IP) as the glue of the Largest Computer Network</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Stream Control Transmission Protocol (SCTP)</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Exchanging TCP with SCTP</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Organization of this Thesis</td>
<td>3</td>
</tr>
<tr>
<td>2 Theoretical Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Stream Control Transmission Protocol</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Historical overview</td>
<td>6</td>
</tr>
<tr>
<td>2.3 SCTP structure</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Key features of SCTP</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1 Paradigm of Message Delivery</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1.1 Ordered or Unordered Delivery</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1.2 Selective Acknowledgment</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2 Setting up an Association</td>
<td>10</td>
</tr>
<tr>
<td>2.4.2.1 Three-way-handshake</td>
<td>10</td>
</tr>
<tr>
<td>2.4.2.2 Four-way-handshake: A Cookie for the Attacker</td>
<td>11</td>
</tr>
<tr>
<td>2.4.3 One is not enough: Multi-Homing</td>
<td>12</td>
</tr>
<tr>
<td>2.4.4 Multiple Streams within an Association: Multi-Streaming</td>
<td>14</td>
</tr>
</tbody>
</table>
## 2.4.5 The End: Shutdown and Termination of an Association

## 2.4.6 Lack of half-closed state

## 2.5 Chunks on the way: Basic Data Transmission

### 2.5.1 The DATA chunk

### 2.5.2 Sending user Data

### 2.5.3 Summary

## 3 Motivation, Design and Implementation

### 3.1 Motivation

#### 3.1.1 Head-of-line blocking in Transport Protocols

### 3.2 Speeding up Java Web Applications and Web Service calls

#### 3.2.1 Unordered Delivery

#### 3.2.2 Multi-Streaming

#### 3.2.3 Together we are Strong

### 3.3 Library Design and Implementation

#### 3.3.1 Preface

#### 3.3.2 Substructure

#### 3.3.3 Sockets API

#### 3.3.4 Library Architecture and Implementation

##### 3.3.4.1 SCTPSocketImpl class

##### 3.3.4.2 Building user specific Data Chunks

##### 3.3.4.3 SCTPSocketImplFactory class

### 3.4 Design Summary

## 4 Experimental Evaluation

### 4.1 Experimental Setup

#### 4.1.1 Hard- and Software configuration

#### 4.1.2 Evaluating SCTP and File Transfers

##### 4.1.2.1 Experimental preparation

##### 4.1.2.2 Results and Evaluation of the measurements

##### 4.1.2.3 Size matters

##### 4.1.2.4 Impact of packet loss
Chapter 1

Introduction

1.1 The Internet Protocol (IP) as the glue of the Largest Computer Network

Over the past several years computer technology has been revolutionizing the way applications operate and communicate with each other. The largest computer network nowadays is the Internet. The Internet and the services it offers have gone through an exponential growth, from a small network containing only few nodes to a big global heterogeneous computer network with millions of hosts and users. The quickly growing popularity led to a state where having access to the Internet is as normal as having a TV or a mobile phone at home. This phenomenon builds on top of a quite old protocol, the Internet Protocol (IP) [1]. IP technology has become more and more established over the last 35 years, and IP has proved to be a very robust and reliable network protocol.

The main role for the Internet Protocol is to close the gap between the mixture of different networks. It is the language that each computer must speak and understand to carry and receive information.

The Architecture of the Internet protocol suite (see Figure 1.1) shows that the IP protocol is assigned to the so called Network Layer. On the top of the this layer we have the Transport Layer. Here two of the most popu-
lar and widely deployed protocols are located, the Transmission Control Protocol (TCP) [2] and User Datagram Protocol (UDP) [3]

<table>
<thead>
<tr>
<th>Application Layer</th>
<th>HTTP, SMTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Layer</td>
<td>SCTP, TCP, UDP</td>
</tr>
<tr>
<td>Network Layer</td>
<td>IPv4, IPv6</td>
</tr>
<tr>
<td>Link Layer</td>
<td>Ethernet</td>
</tr>
</tbody>
</table>

Table 1.1: Layered architecture of the Internet Protocol suite

On the one hand we have the TCP/IP protocol suite which was initially developed for academic and research communities to handle communication over internetworks. TCP has become a "de facto" standard protocol for transport layer functionality which provides a connection-oriented, congestion controlled, reliable, in-order delivery of one stream of bytes. On the other hand there is a very simple protocol called UDP. It offers a fast, connectionless and unreliable service for applications and services regardless to congestion.

1.2 The Stream Control Transmission Protocol (SCTP)

As the range of commercial applications, such as real-time multimedia or Internet telephony, grows wider, people have started to feel that transfer services offered by TCP and UDP are not adequate anymore. Nowadays, many applications still need reliable data transfer, but for some of these applications the characteristic of strict sequence preservation is not truly necessary.

In consideration of these facts, the Stream Control Transmission Protocol (SCTP) has been developed and standardized. Similar to TCP, SCTP provides a reliable transport service, ensuring that data is transported
CHAPTER 1. INTRODUCTION

across the network without errors. SCTP and TCP have many things in common, but SCTP removes some restrictions, such as strict byte ordered delivery. The decisive difference to TCP is the support of the advanced features *multi-homing* and *multi-streaming*. Another additional feature of SCTP is the possibility to deliver all data completely *unordered* but still *reliable*.

#### 1.3 Exchanging TCP with SCTP

In this thesis we will take advantage of the advanced features of SCTP. We will especially concentrate on the two abilities *multi-streaming* and *unordered delivery*. While each of these features are useful on their own, we want to show the power of combining them while exchanging data. Due the avoidance of the head-of-line blocking delay we should be able to reduce users perceived latency and improve throughput. Especially applications with a large number of independent transactions should gain a performance advantage by using these features. We developed a Java library, which supports an easy migration path for existing Java TCP applications to use the advantages of SCTP by changing the transport protocol. Many TCP-based applications will now have the opportunity to take advantage of SCTPs advanced features with only slight changes within the source code.

#### 1.4 Organization of this Thesis

The remainder of this thesis is structured into two main parts. The first part, starting in chapter 2, gives a brief introduction of the basic concepts of the Stream Control Transmission Protocol. Beside the theoretical background of SCTPs structure, services and features in section 2.1 and 2.3, also some interesting historical information is given in Section 2.2.

The second main part takes a detailed look at the related work and starts by focusing on the idea behind the design and implementation of the library in Chapter 3. Chapter 4 shows several experiments comparing the library against the performance of standard TCP. Related work and projects that try
to bypass the drawbacks of TCP by using SCTP as the transport protocol is discussed in Chapter 5. Chapter 6 concludes the thesis by giving an overview of possible future work and challenges.
Chapter 2

Theoretical Background

This chapter gives a brief introduction to the basic concepts of the Stream Control Transmission Protocol (SCTP). While SCTP has a lot in common with TCP (e.g. reliable and ordered data transmission), it provides some advanced features that will be discussed in this chapter. Section 2.1 starts with a definition of the SCTP protocol. Before examining the structure, services and features of the SCTP protocol in more detail (Section 2.3), Section 2.2 introduces the historical background of SCTP.

2.1 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) provides a general purpose reliable end-to-end message transportation service for IP based networks. The SCTP protocol was published as RFC2960 [4] by the IETF in October 2000 as a proposed standard. SCTP offers reliability, flow control, ordered delivery and full-duplex data transfer like TCP. However, SCTP also provides some other features unavailable in the current transport protocols of the Internet, like multi-homing and multi-streaming. It also uses different startup procedure, which we will discuss in the following sections. Similar to TCP [2] and UDP [3], the Stream Control Transmission Protocol belongs to the transport layer within the IP architecture shown in Figure 1.1.
CHAPTER 2. THEORETICAL BACKGROUND

2.2 Historical overview

The motivation for developing SCTP was initially introduced in 1998 by the IETF working group SIGTRAN, which was formed to design a mechanism to transport telephony signaling messages in commercial systems. The question whether TCP was good enough for telephony signaling applications was quickly answered. The working group pointed out some key issues that prevent the usage of TCP. Beside the **head-of-line blocking** and the **three-way handshake** which could become a major problem due to signaling delays which can escalate to an unacceptable level, also the wish for **path redundancy** support forced the working group to design a new transport protocol. After SIGTRAN began to work on the protocol, the IETF recognized the value of solving these problems for a wider audience than only telephony signaling. Many other applications could also gain advantages by using the SCTP protocol. So a new transport protocol called SCTP was born. [5]

2.3 SCTP structure

In the following section we will look at the basic format and structure of SCTP. As defined in [6], the protocol data units (PDU) of SCTP are called SCTP packets. If SCTP is used on top of IP, as described in RFC2960 [4], an SCTP packet forms the payload of an IP packet. An SCTP packet is composed of a common header and chunks. Multiple chunks may be multiplexed into one packet up to the Path-MTU size. A chunk may contain either control information or user data. Figure 2.1 illustrates the format of an SCTP packet containing multiple chunks.

The first 12 bytes of the **SCTP packet** contain the **common header**. For identifying an association, SCTP uses the same port concept as TCP and UDP along with a 32 bit Verification Tag. In contrast to TCP and UDP, which both use a 16 bit checksum, SCTP uses a more robust 32 bit checksum (Adler-32 algorithm [7]) to detect transmission errors. If an error
CHAPTER 2. THEORETICAL BACKGROUND

Figure 2.1: SCTP Data Unit with several chunks

occurs, packets with an invalid checksum are silently discarded.

Chunks providing SCTP with the basic structure need to carry information between different SCTP endpoints. There are two kinds of chunk classification. On the one hand we have control chunks which are, by definition, chunks that carry control informations and maintain the SCTP association. On the other hand there are data chunks, which are used to carry user messages between the endpoints. To distinguish between data and control chunk types, each chunk contains a Type field. The Flag field defines any special flags that the chunk may wish to use. The value of the Length field contains the length of the actual payload. One of the main reasons that a chunk-based format was chosen for SCTP was its extensibility. The designed format allows the definition of up to 256 different chunk types. Currently 13 chunk types are defined, leaving 240 additional chunk types that may be defined in the future by the IETF. For more detailed information on how chunk types are defined take a look at the RFC2960 [4].
2.4 Key features of SCTP

At first glance the two protocols SCTP and TCP look very similar. In this section we will take a closer look at the theory and will find distinct differences between these two protocols. The table 2.1 shows a comparison of services and features of SCTP, TCP and UDP.

<table>
<thead>
<tr>
<th>Services and Features</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>Full-duplex</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Reliable data transfer</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Partial reliable data transfer</td>
<td>yes</td>
<td>no</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>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>Selective ACKs</td>
<td>yes</td>
<td>optional</td>
<td>no</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>Multi-streaming</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Multi-homing</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Protection against SYN flooding attacks</td>
<td>yes</td>
<td>no</td>
<td>n/a</td>
</tr>
<tr>
<td>Allows half-closed connections</td>
<td>no</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>Reachability check</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Pseudo-header for checksum</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2.1: Services and features overview of SCTP, TCP and UDP

2.4.1 Paradigm of Message Delivery

As SCTP was initially developed to carry telephone signaling packets, its designers concentrated on implementing a message-oriented protocol. The advantage of this concept is to avoid overhead by sending several small messages within single SCTP datagrams. This is an important distinction be-
tween TCP and SCTP. While TCP is sending data between two endpoints in byte-oriented fashion, SCTP preserves message boundaries. The only exception occurs if the message is larger than the peer’s receiver window. In this case we have a partial delivery of one message without mixing different messages.

2.4.1.1 Ordered or Unordered Delivery

In contrast to TCP, where all messages are reliably delivered to a receiver in exactly the order used by the sending application, SCTP provides the ability of reliable delivery of messages with no order constraints. The possibility for unordered delivery is a big advantage of SCTP over TCPs ordered delivery when an application has to deal with a large number of independent transactions. The feasibility to send data unordered gives a developer a means to defeat the head-of-line blocking delay which we will discuss in more detail in section 3.1.1. As we mentioned in 2.4.1, TCP sends its data in a stream of bytes. This means that TCP has no possibility to recognize where an individual message begins or ends. So there is no other way for data delivery than to keep the original sending order.

2.4.1.2 Selective Acknowledgment

Selective Acknowledgment (SACK) is used by the SCTP endpoint to carry all transmission sequence numbers (TSN) that have been received, as a so-called Cumulative TSN. It is an indicator for all successfully received data which was already delivered to the upper layer (or is ready for delivery). Within the SACK chunk there are so called Gap Blocks, which are indicators for missing data within a received data block. If the sender of the DATA chunk does not receive the SACK chunk within a certain period of time, or four SACK chunks have reported gaps with the same missing data chunk, a delivery failure will be assumed and a retransmission (fast retransmission) of the data will be initiated [6].
2.4.2 Setting up an Association

To establish a relationship between two endpoints, both protocols have to walk through a predefined message exchange. Instead of using a three-way-handshake like TCP, SCTP is setting up a connection through a four-way-handshake with a signed cookie, shown in Figure 2.2.

A SCTP association is defined in [4] as:

*a protocol relationship between SCTP endpoints, composed of the two SCTP endpoints and protocol state information including Verification Tags and the currently active set of Transmission Sequence Numbers (TSNs), etc. An association can be uniquely identified by the transport addresses used by the endpoints in the association. Two SCTP endpoints MUST NOT have more than one SCTP association between them at any given time.*

![Figure 2.2: Setting up a connection with TCP and SCTP](image)

2.4.2.1 Three-way-handshake

In TCP, the connection phase consists of a three-way-handshake. The following steps are needed to establish a TCP connection.
• The active open is performed by the client sending a SYN packet to the server.

• The server replies with a SYN-ACK packet.

• Finally, the client confirms the receipt by sending back an ACK packet.

TCP's three-way-handshake is vulnerable to denial of service attacks, also called SYN attacks. This kind of attack is a very simple and effective one which can affect any system connected to the Internet using TCP based protocols like HTTP or FTP. A good explanation of this attack can be found in [8].

After receiving a SYN packet, the server allocates resources for the connection (like IP-address or port number) while waiting for the ACK message. A server in this state has a half-open connection. Attackers are exploiting this behavior by using the SYN flood attack, where a client is repeatedly sending SYN packets to the server which has to consume large amounts of resources to keep track of the incoming connections. The memory space available to keep the information of all these incoming and pending connections is limited and not infinite. Sooner or later there will be a point where the attacked system is unable to accept more incoming connections because all of its resources are consumed. This results in the denial of service for the other users. TCP has the possibility of using a timer to remove these half-open connections from memory. This may help the system to recover, but if the attacker continues to send SYN messages nothing will change.

2.4.2.2 Four-way-handshake: A Cookie for the Attacker

The establishment phase of SCTP uses a four-way-handshake to protect against SYN flood attacks by adding the so called cookie mechanism. The designers of SCTP pointed out that to avoid SYN flooding two things are necessary.

• a way to prove if the associations initiator (client) is using its real IP-address.
• the server should not start to allocate memory until an association is completely established.

To fulfill the first requirement the **cookie mechanism** was introduced. It is in principle a simple but powerful mechanism for avoiding flooding attacks. After receiving an INIT chunk, the server creates a secret key (a cookie) including transmission control block information, together with a valid lifetime and a signature for authentication, and sends this back to the association initiator. The client has to send this cookie back to the server afterwards. If the cookie is unmodified we can be confident that the initiator is no attacker.

The following steps are showing what happens when an SCTP association starts up.

• To initiate a connection, the client sends an INIT chunk to the server.

• The server replies with an INIT-ACK chunk. Inside the INIT-ACK there is a state cookie for identification.

• The client echoes back the cookie within the COOKIE-ECHO chunk.

• Finally, after receiving the COOKIE-ECHO, the server allocates the resources for the connection and returns an acknowledgement, the COOKIE-ACK chunk, back to the client.

The four-way-handshake could make it seem that SCTP needs more number of messages that must be exchanged before data can be transferred. However, SCTP permits to include data within the COOKIE-ECHO and COOKIE-ACK packets to avoid the problem of delayed data movement. This is an improvement over TCP, which has to wait until the three-way handshake has completed.

### 2.4.3 One is not enough: Multi-Homing

One of the key feature of SCTP is the native support of **multi-homing** which is a big advantage over TCP or UDP. Multi-homing is the possibility
to assign more than one IP address to an \textbf{SCTP endpoint}. A definition of a \textit{SCTP endpoint} can be found in RFC2960 [4]:

\textit{The logical sender/receiver of SCTP packets. On a multi-homed host, an SCTP endpoint is represented to its peers as a combination of a set of eligible destination transport addresses to which SCTP packets can be sent and a set of eligible source transport addresses from which SCTP packets can be received. All transport addresses used by an SCTP endpoint must use the same port number, but can use multiple IP addresses. A transport address used by an SCTP endpoint must not be used by another SCTP endpoint. In other words, a transport address is unique to an SCTP endpoint.}

This normally includes that the host machine has multiple network interfaces (NI). In the past, hosts usually had only one network interface due to the relatively high costs of network interfaces. So there were no serious need to follow a concept of multi-homing for traditional Internet protocols like TCP or UDP. Nowadays, the cost of additional network interfaces is very low compared to the advantage we can gain from using multiple network connections. SCTP breaks this lack of design concepts from traditional protocols and integrates multi-homing support at the transport layer. Figure 2.3 shows two endpoints with two network interfaces per host. This scenario supports four different paths through the network. One of the IP addresses will be selected as the \textit{primary address} and all other addresses of the peer become \textit{alternate addresses}. All data is sent to the primary address, as long as no endpoint failure occurs. If the primary connection breaks SCTP automatically continues transferring traffic over an alternate path. Another service taking advantage of multi-homing is the possibility to completely hide a connection failure from an application.

So far SCTP uses multi-homing only for redundancy purposes, not supporting simultaneous data transfer between two endpoints through the use of more end-to-end paths. Probably this feature will become a standard in
SCTP. As [9, 10] describes, the technical requirements are already here to also use multi-homing to provide simultaneous transfer of data between two SCTP endpoints through more than one end-to-end paths.

![Multi-homed SCTP connection](image)

**Figure 2.3:** Multi-homed SCTP connection

### 2.4.4 Multiple Streams within an Association: Multi-Streaming

Another key feature of SCTP is **multi-streaming**. A *Stream* itself is defined in RFC2960 [4] as:

*a uni-directional logical channel established from one to another associated SCTP endpoint, within which all user messages are delivered in sequence except for those submitted to the unordered delivery service.*

One SCTP association can include several logical streams. All the streams are independent but related to the association (as shown in Figure 2.4). Within a single SCTP association, up to 65,536 unidirectional streams can be used by either end for simultaneous data transfer.

Multi-streaming is an important approach to avoid the **head-of-line blocking** problem, which will be discussed in more detail later in section 3.1.1. All data within each single stream follows the concept of ordered
delivery. Simultaneous transmission over parallel streams provides a so called **partial ordered** delivery. If one of the streams gets blocked, for example awaiting a retransmission after a packet loss, this does not affect the other streams within the association.

This concept is also a big advantage over TCP. If a TCP connection wants to send multiple independent messages to a receiver at the same time, it has to open the same amount of independent TCP connections. It seems clear that this creates much more overhead and traffic than creating multiple streams within a single SCTP association.

![Figure 2.4: Multi-streaming: several streams within a SCTP association](image)

### 2.4.5 The End: Shutdown and Termination of an Association

Reliable transport protocols need a shutdown algorithm to close a connection. Both TCP and SCTP have to follow a predefined teardown procedure, which can be seen in Figure 2.5. SCTP uses a **three-way handshake** shutdown mechanism. If either endpoint starts the **graceful shutdown** procedure by sending a *SHUTDOWN* chunk, both sides of the SCTP association will empty their buffer and stop sending new data. The receiver of the *SHUTDOWN* chunk replies with a *SHUTDOWN-ACK* chunk which indicates that the *SHUTDOWN* chunk was
received and all buffered data has been sent, waiting for the \textit{SHUTDOWN COMPLETE} chunk to completely finish their association by closing their socket. Now no further data transmission can be permitted in either direction.

\section*{2.4.6 Lack of half-closed state}

The obviously biggest drawback of using SCTP instead of TCP is the lack of the \textit{half-closed} state. While TCP allows closing one side of the connection, which is the so called \textit{half-closed} state, the designer of SCTP removed this feature because of the extra complexity and the lack of use. While the \textit{four-way handshake} during the establishing process has been accepted as a great improvement over TCP, the lack of the half-open connection feature has been discussed already. As Figure 2.5 illustrates, tearing down a TCP connection involves a four-way handshake mechanism. One endpoint sends a \textit{FIN} packet responded with an \textit{ACK} from the receiver endpoint. At this point we have a \textit{half-closed} state until the remaining endpoint sends a \textit{FIN} packet of its own for complete shutdown. By default SCTP doesn’t allow half closed state. While in TCP both endpoints have to agree finishing the
connection in SCTP only one side has to start the teardown mechanism. The SCTP designers correctly assumed that the demand of a half-closed state is relatively small but there are some existing and long-established applications requiring this state of the connection for correct operation. One of the most important examples is the file transfer protocol (FTP). Under some circumstances it requires a half-closed connection for transferring the files correctly [11].

2.5 Chunks on the way: Basic Data Transmission

2.5.1 The DATA chunk

To transfer data between two SCTP endpoints two chunks, the DATA chunk and SACK chunk, are required. While we have already discussed the SACK chunk in 2.4.1.2, we will take a closer look at the DATA chunk now (see also Figure 2.6).

<table>
<thead>
<tr>
<th>Type</th>
<th>0</th>
<th>Reserved</th>
<th>UB</th>
<th>E</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream Identifier</td>
<td>Stream Sequence Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload Protocol Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6: The Data chunk

The different fields of the DATA chunk are described as follows:

**Chunk type** This field defines the type of the chunk. In our case, to use a payload Data chunk, this field is set to 0.

**Chunk flags** The three flag-bits $U$, $B$ and $E$ are used within the DATA chunk to define some special behavior.
CHAPTER 2. THEORETICAL BACKGROUND

Setting the \( U \) bit indicates that user data are delivered in an unordered way.
When the bit \( B \) is set to 1, the beginning part of a user message is present in this DATA chunk.
Setting the bit \( E \) to 1 indicates the ending part of a user message within this chunk.

**Chunk length**  Chunks have a variable length. So this field is used to define the length of the user data.

**TSN**  Every DATA chunk is identified by its Transmission Sequence Number (TSN). The value of the TSN is used to ensure that the chunk arrived at the destination. The TSN counts the DATA chunks sent to the different targets. The TSN is comparable to TCPs Sequence Number field.

**Stream identifier**  This field identifies the stream to which the data chunk is directed.

**Stream Sequence Number**  Is a value for indicating what stream sequence number this user data contains. Messages within one stream are delivered in order to its destination.

**Payload Protocol identifier**  SCTP will not be responsible verifying any payload protocol identifiers. So SCTP receives the identifier from the upper layer, ignores this field and carries it with the corresponding payload data.

**User Data**  This is the user’s payload data. The size is limited to the PMTU. If the message is larger the sender has the possibility to fragment the message into parts and send these in separate DATA chunks.

### 2.5.2 Sending user Data

To send user data across the network the startup procedure, as described in 2.4.2.2, has to establish an association first. Afterwards user messages
have to be converted into DATA chunks. Depending on the size of the message, either the conversion is forming a single DATA chunk by adding a DATA chunk header to the message, or it splits the message into several smaller pieces and converts each of them into DATA chunks. Considering the definition of the Chunk length field within a DATA chunk as described in 2.5.1, some may ask why we have to split the message into several pieces.

The answer is: size matters. The size of each SCTP datagram is limited by the underlying network layer. In IP we have a defined limit of 65,535 Bytes. Due to the underlying physical network and its various conditions, this value, the so-called Maximum Transmission Unit (MTU), is not fixed and may strongly vary between some hundreds of bytes to several thousands of bytes.

Beside adding the DATA chunk header to the user message, each chunk is assigned a Stream Number, Stream Sequence Number and Transmission Sequence Number. First we have to define the stream that the outbound message has to be directed to. This is stored in the Stream identifier. SCTP data transfer streams are sending messages in an ordered way by default. In this case the sender has to define the Stream Sequence Number and store it in the Stream Sequence Number field which is responsible for ordered delivery. To protect against packet loss, the value for the TSN field is assigned as well.

Together the Stream Number, Stream Sequence Number and Transmission Sequence Number are responsible for assembling user data and guaranteeing the ordered delivery of messages within a stream. After all assignments are done, the DATA chunk, regardless of the stream it belongs to, is put into the chunk queue which is arranged according to the chunk TSN number. A DATA chunk must contain some user data. It is not allowed to send empty DATA chunks under any circumstances. Transmission of such a chunk will induce the receiving endpoint to abort the association.
2.5.3 Summary

The previous chapter gave an overview of the history and a theoretical background of SCTP, which is a general purpose reliable end-to-end message transportation protocol for IP based networks. We presented the structure and introduced services and features of this relatively new transport protocol. Furthermore we discussed the main differences between TCP and SCTP by examining the implemented advanced features of SCTP. Through the feature of multi-streaming, SCTP performs very well for data connections that exhibit packet loss. The multi-homing feature, which introduces path-redundancy, is a great improvement over the actual established Internet protocol suite. On the whole, SCTP has a lot of advantages over TCP and only very few drawbacks.
Chapter 3

Motivation, Design and Implementation

SCTP promises to be a well suited and attractive transport protocol to replace TCP for some applications. This especially applies to programs with a large number of independent transactions, due to its message oriented nature and the accomplishment of unordered but still reliable data delivery. The section 3.2 discusses the idea, design and implementation of a SCTP-library which can be used to replace a TCP connection with a multi-streamed and unordered SCTP data transfer where only slight changes within the code have do be done through the usage of the abstract factory pattern. We will also discuss the circumstances under which this protocol exchange can take place.

The next sections will give an overview of the advantages behind the idea of using unordered delivery and multi-streaming for data transmission.

3.1 Motivation

As we mentioned before, SCTP has many advantages over TCP through its use of advanced features. Nevertheless SCTP has not been widely deployed yet. The reason is the often cited and discussed “chicken and egg” type problem. On the one hand developer and software designers are not really interested in supporting SCTP instead of TCP as the transport layer proto-
col because the consumers do not ask for it. On the other hand, end users do not ask for it because nearly no application is written for SCTP support. So the reason to design and develop a Java SCTP library which makes some kind of translation from TCP to SCTP was to have a simple but powerful tool to convert existing Java based TCP applications into Java based SCTP applications. One cornerstone of this thesis is to speed up Java web applications and web service calls by completely avoiding head of line blocking. Therefore the library also potentially and automatically uses the advantages of multi-streaming and unordered delivery. Another nice side-effect of this library is that developers and users now have a tool to try out and take advantage of the features of SCTP without huge modifications of the source code. Before we go into more detail of the design of the library we want to introduce the phenomenon of the head-of-line blocking delay.

3.1.1 Head-of-line blocking in Transport Protocols

The classical Internet transport protocol provides either an ordered and reliable service like TCP or a service which does not guarantee order and reliability like UDP. In [12], the author precisely describes that ordered delivery and reliability are two orthogonal issues. Some applications need detection of message loss (reliability) but they do not categorically require an in-order delivery of the transferred data.

To fulfill the applications requirements, data transmission can be classified as ordered, partially-ordered, or unordered delivery. Strictly-ordered or partially-ordered services have the difficulty of potentially introducing unnecessary delays in case of packet loss. This phenomenon is called head-of-line blocking and occurs in buffered service systems. Related to IP based transport protocols, head-of-line blocking appears if a strict sequence maintenance like in TCP is offered. The loss of a single IP-packet can lead to long and unacceptable delays to the overall data delivery. Based on the offered strict ordering, all subsequent messages have to wait for a successful retransmission of the lost segment. Of course, packet loss in
networks are usually small but network links with a lot of traffic are still affected [13, 5].

The following example will illustrate the head-of-line blocking issue. If we are surfing the Internet and open a single webpage a typical HTTP request is made. A webpage is a complex resource of information and contains many kinds of different files including textual information, static graphics (gif, jpeg or png), animated images (gif or svg) audio files (midi or wav) and video files (mpg, mov, rm or wmv). To download information from a webpage, HTTP 1.0 opens several separate TCP connections for each different file and closes each after transmission. This kind of behavior induces a lot of overhead and so HTTP 1.1 tries to solve this problem by using persistent connections and pipelining. The issue requires that web servers are supporting pipelining [14].

Figure 3.1 illustrates a typical request to a web server to download a webpage. The left side of the Figure represents the web server and the right side the requesting client. For simplicity, we assume that every file is contained in three datagrams. For better comprehension we further use HTTP 1.0 for the schematic data transfer which also illustrates the behavior for other parallel file transfers.

Figure 3.1 a) shows a single TCP connection transferring two requested files. During the data transfer the first datagram was lost. Afterwards, the two remaining packets of the first file and the complete second file arrives correctly at the client. Due to the strict sequence maintenance in TCP, even though the second file was delivered entirely, it can not be passed to the upper layer until the first datagram of the first file was successfully resent. This behavior is the so called head-of-line blocking problem.

In Figure 3.1 b) we can see that the client opens a TCP connection for each requested file. In this case we are partially avoiding head-of-line blocking by using partial-ordered delivery. Again, after losing the first
datagram of the first file the remaining arrived packets have to be buffered until the lost packet is resent. The difference to a single connection is that the second file can be passed to the application as soon as it arrived at the client. However, for each single TCP connection we introduce some delay and overhead in opening and closing. Furthermore TCP does not scale well because the maximum of different simultaneous TCP connections is limited by the kernel. Also the amount of wasted resources to handle several open TCP connections at the same time is not negligible.

Figure 3.1 c) demonstrates one way SCTP handles the problem of the head-of-line blocking delay. While a single SCTP stream does not avoid the head-of-line blocking problem, the feature multi-streaming partially does. It follows the same principle as TCP. In case of packet loss, using different streams avoids transferring delays for other streams by using partial ordering. Each single file is transmitted in one own stream and SCTP ensures reliable and ordered data delivery within the same stream. In contrast to using several TCP connections, SCTPs multi-streaming is saving resources, avoids establishment delays and overhead because we still have only one association per client. Another big advantage is that the single association is using the same congestion avoidance mechanism. Several TCP connections are using multiple congestion avoidance instances and violate the TCPs axiom of using available bandwith in a fair way.

Another feature of SCTP shown in 3.1 d) is to transfer data unordered but still reliable. Using unordered delivery completely removes the head-of-line blocking issue. An unordered message is delivered to the upper layer as soon as it arrives at the destination. This possibility is similar to a transmission using UDP. However, the upper layer protocol must have a mechanism to deal with unordered and potentially out-of-sequence messages.
3.2 Speeding up Java Web Applications and Web Service calls

Decreasing delay and increasing throughput is the quintessence to improve data transmissions. As we mentioned before, head-of-line blocking indis-
putably has a big impact on the response time of message transactions. For speeding up web applications and web service calls, the solution is to avoid head-of-line blocking delays without increasing overhead as it happens when several TCP connections are used. SCTP seems to be a perfect candidate for this challenge due to its feature to send messages unordered but still reliable and the feature of multi-streaming.

3.2.1 Unordered Delivery

On the one hand data transfer with the transport protocol SCTP is delivered in an ordered way by default. This means that each message is delivered to the application at the receiver side in the same order as it has left the sender. On the other hand SCTP has the powerful ability to specify messages to be delivered unordered. This means that user messages are delivered to the upper layer as soon as they arrive at the destination regardless of their order. This functionality is the same as in UDP with the difference that reliability is still given.

When a message should be transmitted unordered, the data sender has to set the flag U in the Data chunk(s) carrying the message. This indicates to the receiver that this message is unordered and can be passed directly to the upper layer. This kind of data chunks do not use the Stream Sequence Number (SSN) field. If the U flag is set, the receiver simply ignores the SSN field regardless of the value which is filled in. Thanks to the Transmission Sequence Number (TSN) and the flags B and E, lost data can still be detected and retransmitted. A detailed description of the different kinds of flags can be found in section 2.5. Moreover, an entire stream can be made into an unordered stream by simply sending each message in unordered mode over that stream [5]. As we mentioned before, this kind of data transfer completely avoids head-of-line blocking.
3.2.2 Multi-Streaming

As already discussed in section 2.4.4, multi-streaming is another feature of SCTP. The following example will give a detailed explanation of how multi-streaming works. Figure 3.2 shows an association between two endpoints, the sender and receiver. Within this association we have three streams identified as 0, 1 and 2. In this example we want to send the messages A, B, C, D and E from the sender to the receiver. After establishing the connection, the application has to assign sequences of messages to an arbitrary stream. In our case we want to send the message A and B over stream 0, C and D over stream 1 and E over stream 2. After the decision which message is sent over which stream is made, the messages are passed to the SCTP layer and will be fragmented; this is the process where user messages are converted into SCTP data chunks. Depending on the size of each message, the conversion is simply adding the chunk header to the message or, if the user message size is bigger than the actual threshold, usually the current PMTU, the message has to be split into several parts before assigning a separate chunk header for each. Adding the SCTP chunk header implies that the sender has to assign a unique Transmission Sequence Number (TSN), a Stream Sequence Number (SSN) and the Stream Identifier(SI) value to every data chunk. As defined in 2.5.1, the TSN is responsible for sequencing the overall data chunks while the SSN is used for sequencing the data chunks within each stream. The SI assigns the specific stream number to the chunk. If a message has to be split into several data chunk parts the SSN number will be shared for all these data chunks. In our example B,D and E are split into more data chunks (the assigned values can be seen in the figure). Afterwards, the data chunks are placed in the chunk queue and are ready for delivery. By default, within each stream, SCTP guarantees in-order delivery. The ordering of the packets that are not within the same stream is undefined and can be changed. As one can see in our example, after assigning all values, the message A will always be delivered before B and message C before D. On the other hand it is not predictable if the receiver will first pick up message A, C or D due to the
unordered delivery between all streams.

![Multi-Streaming: An example with one SCTP association and three streams](image)

<table>
<thead>
<tr>
<th>Sender</th>
<th>SCTP Association</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 0</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>Stream 1</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Stream 2</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

**TSN ... Transmission Sequence Number**  
**SSN ... Stream Sequence Number**  
**SI ... Stream Identifier**

3.2.3 Together we are Strong

*Multi-streaming* and *unordered delivery* are both useful features on their own; we may ask ourselves how big their benefit is if they are combined? We will answer this question in the following sections and experiments. This thesis concentrates on solving the potential performance problems for applications which consist of a large number of independent transactions by using multi-streaming and unordered data delivery. As defined in [15], this kind of data transfer can be considered as a request/response oriented communication (like Web Service calls). A basic structure requires the following messages to be transferred:

**Request**  
Execute or request function X (with parameters Y)

**Response**  
Done: returning the result
It seems clear that, as described in 3.2.2, nearly every request/response oriented application can gain an advantage by sending its data through multi-streaming instead of a single stream. In this thesis we want to go one step further and combine multi-streaming and unordered delivery for such communication. Especially due to using unordered delivery, not every client-server application is able to use the library in a reasonable way. The upper layer protocol must have a mechanism to deal with data packets delivered out of order. Here we will present a small example of such an independent request/response transaction which can also deal with unordered delivery: let us assume that we want to request a download from a large progressive JPEG picture from a web server. This kind of JPEG is a file format which allows to preview the picture after receiving the first parts of the whole data. With each delivered part of the picture, the quality of the image increases. (This is called progressive JPEG, [16]) Finally, after receiving all pieces, the picture shows up in its whole glory. It seems clear that this kind of data transfer does not need any kind of in-order maintenance but still reliability. If a data chunk gets lost, the picture still gets a quality improvement through the other bits on the wire without introducing delays by waiting for immediate retransmission of the lost data chunk through the usage of unordered delivery. It is obvious that this combination of features makes no sense for transferring files which have to be received in order for further usage.

3.3 Library Design and Implementation

This section describes the implementation details of the library.

3.3.1 Preface

At the beginning of this work there was no Java library for SCTP available. So the plan for the first few months of my work was to develop such a library. Java does not support SCTP directly and due to the actual socket concept the only way to implement SCTP in Java is to use the Java Native Interface
JNI. JNI allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. Programmers use the JNI to write Java native methods to handle those situations when an application cannot be written entirely in Java [17]. During deployment of my own SCTP library another developer released a JNI based Java SCTP implementation. Under the circumstance that this library was already finished and probably well tested, the decision to work and do further development on the project with the new library, which can be found under [18], was taken.

3.3.2 Substructure

As described in [18], the Java SCTP library consists of both Java classes and JNI. The implementation is based on the LKSCTP library version 2.6.22-1.0.7, which is a kernel space implementation of SCTP for Linux based operating systems [19]. The actual implementation of the LKSCTP supports the RFCs 2960 [4], 3309 [20], 3758 [21] and the Internet Draft SCTPSocket-13 [22]. This thesis uses the actual SCTP library version 0.5.7 which has not implemented all functionality that the SCTP socket draft specifies. We will extend and embed missing functionalities within this library for our task.

3.3.3 Sockets API

The sockets API is a bidirectional interface to interact and communicate with a network in a uniform, system-independent and protocol-independent way. A Socket is the core of the sockets model that is a data construct for encapsulating all of the state information required for each endpoint.

At the beginning of this chapter we described the movement from TCP to SCTP as a kind of translation. The term translation is only used for better understanding but as a matter of principle this notation is not correct. At first glance it may seem that the library translates application level system
calls into corresponding SCTP calls and while this could probably be a way to implement the library, if we take a deeper look inside the sockets API for Java there is a more effective design principle for implementation. The sockets API gives us a lot of flexibility to solve the exchange between the protocols TCP and SCTP in an efficient way.

The \texttt{java.net} package in the Java platform provides a class, \texttt{Socket}, that implements one side of a two-way connection between a Java program and another application on the network. The \texttt{Socket} class sits on top of a platform-dependent implementation, hiding the details of any particular system from the Java program. By using the \texttt{java.net.Socket} class instead of relying on native code, the Java programs can communicate over the network in a platform-independent fashion. Additionally, \texttt{java.net} includes the \texttt{ServerSocket} class, which implements a socket that servers can use to \texttt{listen} for connection requests and \texttt{accept} connections for clients [23].

After calling a \texttt{Socket} Object, by default, Java returns a TCP Socket. Figure 3.3 illustrates the general architecture for two Java applications, where the Client is connected to the Server for message exchange. Within the Java code the function \texttt{Socket()} for the client-side and \texttt{ServerSocket()} for the server-side is called and a TCP connection will be established.

### 3.3.4 Library Architecture and Implementation

One of the countless \textit{design patterns}, which are object-oriented techniques to solve common recurring issues for more reusability and flexibility in software design, is the so called \textbf{Factory Pattern}. The abstract factory pattern deals with the problem of creating objects without specifying the exact class of the object that will be created before runtime. So it is responsible for returning an instance of one of several possible subclasses depending on the data provided to it.

Java supports the abstract factory pattern and we use this feature as a simple and efficient way to replace the TCP Socket with an SCTP Socket in an existing source code by using the new factory. As mentioned before,
each class that wants to connect to another endpoint over the network has to use the service of Sockets. The implementation of the behavior of the Socket is hidden in an appropriate SocketImpl class. Once a call within the java.net.Socket is done it is delegated to the actual SocketImpl class. In general, an instance of a Socket or ServerSocket just passes operations to their internal SocketImpl object, which does all the work. This encapsulation concept of consistent interfaces is responsible for a simple and fast exchange to another desired socket implementation. The functionality for returning such classes is integrated in the java.net.SocketImplFactory().

The SocketImpl/SocketImplFactory scheme is useful for, and was designed for, Java applications to be portable across environments that use different transport mechanisms. A client application that uses a java.net.Socket can work in the general case (where the runtime uses a PlainSocketImpl), as well as in environments where network connections need to have special behavior. Its utility is especially intended for extending or changing protocol
functionalities of the network transport. [24]

Our library implementation takes advantage of this underlying factory concept of Java and the following three design steps had to be implemented for the desired SCTP socket behavior.

- Deriving a class from the abstract class `SocketImpl`.
- Deriving a class from the interface `SocketImplFactory` and implementing the method `createSocketImpl()`.
- Defining the `SocketImplFactory` within the existing source code.

The remaining sections in this chapter will discuss and describe the implementation of the most important parts of the library.

### 3.3.4.1 SCTPSocketImpl class

The abstract class `SocketImpl` is a superclass that contains an interface of all methods for sockets implementation. To get an SCTP Socket with the behavior of sending data chunks unordered and over multiple streams implies the implementation of an own `SocketImpl` class. To accomplish this, the `SCTPSocketImpl` class has to extend and overwrite all methods of the class `SocketImpl`. Besides implementing all straightforward functionality we also have to extend and overwrite the classes `InputStream` and `OutputStream` to get the desired SCTP performance. Figure 3.4 shows the class diagram of the above described implementation steps for enhancing the underlying SCTP library.

**Multi-Streaming and unordered delivery**

The aim is not only to exchange TCP with SCTP, but also sending all data by default via multi-streaming and unordered delivery to avoid the head-of-line blocking problem as described in 3.1.1. To fulfill these conditions, the class `SCTPOutputStream` is designed to send all user data over multiple streams with the unordered flag set. By default, we are sending over
1000 streams within each established association. Applications that use more SCTP streams to send user data have to multiplex the data blocks over the given set of streams. The preconditions for choosing an algorithm for distributing the user data on an individual stream seem clear. It has to be a simple and fast algorithm which avoids stream starvation. The already won performance growth should not be lost by time consuming additional calculations. The perfect candidate for these requirements seems to be the Round-Robin algorithm which is one of the simplest schedulers. Figure 3.5 illustrates a short example of how the Round-Robin algorithm works in our case.

We want to send some data chunks over three streams within an SCTP association. In round robin scheduling, each flow queue is allowed to send one packet at a time in round robin fashion. To indicate over which stream...
a specific message has to be transported, the sending application is required to tag each of the outbound message with a stream identifier number. In our example, for outbound message number one, the application assigns the stream identifier number 0 and puts it into the sending queue of stream 0. If the application could still send data, controlled by the receiver window and the application's congestion window, we would put the next outbound message into the queue of stream 1. The next data chunk is selected and is assigned to the queue of stream 2. Afterwards the round-robin process starts at the beginning and redirects the next message for sending into the queue of stream 0 again. This process repeats until the connection is closed and the association is removed.

Listing 3.1 shows some parts of the implementation for using the features multi-streaming with the round-robin scheduling algorithm and unordered delivery together for sending the user data.

```java
private int nr_of_streams = 1000
private short current_stream = 0

// setting the flag for
// unordered delivery
private sctp_sndrcvinfo unorderedSetting =
new sctp_sndrcvinfo();
unorderedSetting.sinfo_flags =
sctp_sndrcvinfo.SCTP_UNORDERED;
...

/**
 * This method writes <code>len</code> bytes from the specified byte array
 * starting at offset <code>off</code> to this output stream.
 * @param b the data.
 * @param off the start offset in the data.
 * @param len the number of bytes to write.
 * @exception IOException if an I/O error occurs while writing.
 *          Happens if the output stream is closed.
 */
```
@Override
public void write(byte[] b, int off, int len)
throws IOException {
...
if (currentChunk == null)
  unorderedSetting.sinfo_stream = current_stream;
current_stream = Short.parseShort(
("" + ((current_stream + 1) % nr_of_streams)));
SCTPData sctpdata = new SCTPData(
  unorderedSetting, data);
  try {
    socket.send(sctpdata);
  } catch (WouldBlockException e) {
    e.printStackTrace();
  }
...}

Listing 3.1: Multi-Streaming and Unordered Delivery

3.3.4.2 Building user specific Data Chunks

Sometimes it is not enough to send data chunks through round robin scheduling over the assigned stream. Some applications may require to transfer a bunch of data chunks over the same stream. An example would be to have an application that has to send all control signals together sharing one stream. To satisfy this request we have implemented a feature to bundle data chunks and send them all over one assigned SCTP stream.

StartChunk() and EndChunk()

The two methods StartChunk() and EndChunk() within the SCTPOutputStream class are designed to bundle data chunks and send them all together over one stream. While the function StartChunk() indicates the beginning of such a data aggregation the method EndChunk() marks the end of the user specific SCTP Data Chunk and sends the whole data bundle over one stream. To use this feature the following three steps have to be included into the source code.
CHAPTER 3. MOTIVATION, DESIGN AND IMPLEMENTATION

• First of all, within the sourcecode, we have to instantiate an SCTPOutputStream like Listing 3.2 shows

```java
SCTPOutputStream os =
    (SCTPOutputStream) socket.getOutputStream();
```

Listing 3.2: Instantiate a SCTPOutputStream

• The second part is to set a flag that marks the start point for the gathering of all upcoming data chunks. Listing 3.3 illustrates the lines that have to be added to the source code. After using the method `startChunk()` for marking the beginn of the data collection, the function `write()` doesn’t send the data directly to the stream as expected. Calling the method `write()` now saves all data until the function `endChunk()` is invoked to finish this procedure and sends all data together over one assigned stream.

```java
try {
    os.startChunk();
} catch (SCTPException e) {
    System.err.println(e.getMessage());
}
```

Listing 3.3: Indicates the Begin of a data collection

• All data chunks will be gathered until the end of the aggregation is declared. To call the function `endChunk()` for sending the whole aggregated data bundle over the assigned stream requires the following line, as illustrated in listing 3.4, within the source code. After ending the user specific data wrapping, calling the method `write()` now sends the user data directly to the stream.

```java
os.endChunk();
```

Listing 3.4: Indicates the End of a data collection

To make sure that the transmitted user specific data chunk only includes the needed data, the user has to empty the wrapper streams before calling
these methods. The function `flush()` forces the buffered data out onto the stream. If the user fail to do that it would be possible that some unexpected data will also be transmitted over the stream which could lead to unintentional behavior on the receiver side application.

### 3.3.4.3 SCTPSocketImplFactory class

As described in section 3.3.4, the abstract factory method design pattern allows a system to determine the subclass from which to instantiate an object during runtime. An application is able to change the default socket factory that creates the SocketImpl subclass by supplying a SocketImplFactory. The user of the Java libraries can call the method `Socket.setSocketImplFactory` (for the client side) and `ServerSocket.setSocketFactory` (for the server side) with an object that will serve as a factory for socket implementations as parameter. The Socket constructor invokes the method `createSocketImpl` of the interface `SocketImplFactory` to create the new SocketImpl object. The downside of the Java socket factories concept is that the methods `setSocketFactory` and `setSocketImplFactory` can only be called once in an application.

To carry out the transport protocol change between TCP and SCTP, the developer only has to fill in the following lines within the the source code.

```java
// setting the new SocketFactory
try {
    Socket.setSocketImplFactory(new SCTPSocketImplFactory());
    ServerSocket.setSocketFactory(new SCTPSocketImplFactory());
} catch (IOException e1) {
    e1.printStackTrace();
}
```

Listing 3.5: Setting the new SocketFactory

Figure 3.6 shows the new communication architecture for two Java applications after setting the new `SocketImplFactory`, where the Client is connected to the Server for message exchange.

After calling a Socket Object, `Socket()` for the client-side and `ServerSocket()` for the server-side, Java now returns an SCTP Socket instead of the default TCP Socket.
3.4 Design Summary

Chapter 3 described the design and implementation principles of our library which is able to exchange the transport protocol of a Java application from TCP to SCTP. Our approach was to create a simple and efficient way to solve this problem without involving major changes within the source code. This task was solved by using the Abstract Factory design pattern. Factory objects represent a helpful technique for making a Java class library more flexible and extensible. The main focus of the work is to give legacy TCP applications the possibility to have access to SCTP for gaining performance enhancement due to SCTPs advanced features especially by sending user data by default via multi-streaming and unordered delivery to avoid the head-of-line blocking delay.

Figure 3.6: Java architecture for SCTP communication
Chapter 4

Experimental Evaluation

This chapter presents the experimental setup and measurements carried out to compare and evaluate the performance of SCTP against TCP. With our results we want to prove that the idea to improve the performance by exchanging TCP with SCTP using multi-streaming and unordered delivery as a combination not only exists theoretically but also in reality. The measurements will show that we attained a performance improvement of transporting messages over the network. Section 4.1 gives an overview of the used equipment and the experimental setup. Section 4.1.2.2 presents the results of the measurements and evaluates the performance of the implementation by comparing the results of SCTP and TCP.

4.1 Experimental Setup

The following sections of this chapter describe the equipment, setup and configuration for our experiments which are presented in section 4.1.2.2.

4.1.1 Hard- and Software configuration

The hardware and software setup for the performance measurements consists of three workstations. For simulating the server and client we used two identical AMD Athlon 64 X2 Dual-Core 4200 with 2.2 GHz. For introducing controlled loss rates we were running NistNet [25] which was installed and
Figure 4.1: Schematic of experimental setup for performance measurements configured on a HP Evo W6000 2.4 GHz workstation. All nodes are connected via 100 Mbit/s Ethernet connection as illustrated in Figure 4.1. All measurements were performed on the operating system ubuntu 7.10 using the lksctp version 2.6.22-1.0.7. [19] supporting SCTP.

4.1.2 Evaluating SCTP and File Transfers

The first part of the measurements is concentrating on transferring files with different sizes and droprates.

4.1.2.1 Experimental preparation

To evaluate our implementation we are first recording the performance of TCP and SCTP while transferring files of different sizes and different drop rates. For each experiment, we measure the total time required to transfer the files between server and the client nodes.

Consistency of the result All our experiments were executed several times to verify the consistency by reducing the variance of our results.
Each individual performance test, using any combination of filesize and packet loss, was running 100 times each. The only exception was the 400 MB file where the number of measurements was 50. The diagrams presented in section 4.1.2.2 are the intersection of all runs of the same combination of file size and packet loss.

**File Size** For the file transfer measurements we are using different *wave-files* of the size 500KB, 5MB, 25MB and 400MB. So we are able to find out if longer transfers have other performance affects than short transfers. The format *wave* is used due its ability to handle out of order packets to investigate the performance of a realistic environment. As we already described in 3.1.1 out of order packets could happen due the usage of *unordered-delivery*.

**Packet loss** To demonstrate the power of SCTP during packet loss we used different loss rates of 0%, 1%, 2%, 3%, 4% and 8% for our experiments. We are using a symmetric data loss which means that we have the same amount of packet loss from the client to the server and the server to the client. It seems a bit disproportionate to do some measurements with a drop rate of 8% but we argue that the tendency of SCTP can be demonstrated more clearly.

### 4.1.2.2 Results and Evaluation of the measurements

In this section we present and compare the performance of the SCTP library against TCP by transferring data as described in section 4.1.2.1. Beside reporting the time requirements of *native TCP applications* and the SCTP library using the combination of *unordered delivery* and *multi-streaming*, we are also observing and comparing SCTPs *multi-streaming* with *ordered delivery*. 
4.1.2.3 Size matters

The first observation of our experiments was that SCTP begins to perform better than TCP when the size of the files increase. We will see that SCTP outperforms TCP by using multi-streaming in ordered and unordered mode. The Figures 4.2, 4.3, 4.4, 4.5 illustrate the result of the measurements for short to long message sizes under different loss rates. For all file sizes smaller than 500KB we observed that TCP performs equal or better than SCTP independent of the amount of packet loss and the usage of SCTPs unordered or ordered delivery. The first interesting filesize where SCTP partially begins to outperform TCP is 500KB, as can be seen in 4.2. Particularly with regard to the data transfer of multi-streaming and unordered delivery, the tendency of achieving a better performance than TCP at higher packet loss rates is already recognizable.

![500 KB Transfer](image)

Figure 4.2: Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 500KB filesize versus packet loss

The trend that SCTP improves the performance against TCP can be monitored by increasing the filesize. In the diagrams 4.3, 4.4, 4.5 we can see
the results for increasing the filesize. SCTP improves its performance with longer file transfers.

![5 MB Transfer](image)

Figure 4.3: Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 5MB filesize versus packet loss

One interesting fact that can be seen in the results of the measurements is that transferring data *ordered* and with only *one* SCTP *stream* can be more efficient than sending the data in any other of the examined ways. This is especially the case for little packet loss (0%-2%) and small file sizes where SCTP also outperforms TCP. With increasing file sizes, the performance for small loss rates is becoming more and more the same as the performance when sending the data *multi-streamed* and *unordered*. The explanation for this behavior is that sending over multiple streams within our library requires some calculations and assigning time of the round-robin algorithm. The break even point for these additional costs seems to only be reached for increasing file sizes and higher packet loss. Another reason is probably that SCTP produces a smaller overhead for one stream than for multiple streams. Another issue that SCTP performance is not as good as TCP for smaller file sizes while sending over multiple streams, is to be found within the lksctp stack.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.4: Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 25MB filesize versus packet loss

Figure 4.5: Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 400MB filesize versus packet loss

The lsctp implementation is nowhere near as optimized as TCP. During the
experiments, one observation was that LKSCTPs buffer management has to be reengineered. Especially for measurements of using unordered but still reliable data transfer for small data, the buffer management on the receiver side often blocks and avoids better throughput.

4.1.2.4 Impact of packet loss

The real goal of this thesis was to gain advantage of combining SCTPs multi-streaming and unordered delivery to avoid head-of-line blocking delays. As described in 3.1.1 head-of-line blocking occurs if we have packet loss while transferring data. The following Figures 4.6, 4.7, 4.8, 4.9 are illustrating the impact of packet loss during file transfer. Impact of packet loss stands for the amount of transfer time that is additionally required if packet loss occurs while exchanging data. The measurements where again run at different loss rates for both short and long message sizes.

The first observation is that over all measurements without simulating any packet loss, TCP performs equivalent to or better than SCTP across all different file sizes. The reasons are probably the more complex association establishment of SCTP and the perfectly optimized TCP stack.

Our measurements verify again that in the case of short messages, we have no benefit of using SCTP over TCP. Diagram 4.6 shows the first file size where SCTP begins partly to overcome TCP.

The impact of packet loss can be observed at the diagrams 4.7, 4.8, 4.9 where SCTP outperforms TCP. At high loss rates (8%) the usage of SCTP and multi-streaming shows an increase in run-time of long file transfers of about 70%. This shows that head-of-line blocking has a substantial effect in applications.

Further within the results we compared the performance between SCTPs unordered delivery and ordered delivery. As we expected over all measurements we can monitor a performance boost by using multi-streaming and unordered delivery. The improvement due the usage of unordered delivery is within the range of 0%-5%. This does not fulfill our expectation at the be-
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.6: Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 500KB

Figure 4.7: Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 5MB
Figure 4.8: Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 25MB

Figure 4.9: Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 400MB
ginning of this master thesis but is a logical consequence while monitoring the measurements. Again the problem is solvable by rewriting and optimizing the lksctp library focusing on the buffer management.

4.1.3 Evaluating SCTP and Web Services

The second part of the measurements is concentrating on evaluating the library with the usage of a Web Service. The deployed Web Service does a simple scalar product calculation. The client is sending two Integer values to the webserver where the product is calculated and the result is sent back to the client. Afterwards the client adds all incoming values for the final result.

4.1.3.1 Experimental preparation

To evaluate the performance of our library implementation between TCP and SCTP for Web Services, we measured the total time between sending data to the Web Service, calculating the result and returning it to the client. This experiment not only gives us information about the performance of SCTP and Webservices but also how multiple independent transactions are influencing the performance.

It seems clear, that the usage of unordered delivery and the possibility of out of order packets leads to a small range of Web Services that are able to use this kind of data transfer. For this case in our library the features StartChunk() and EndChunk() are implemented. We used these features to wrap these two values for the multiplication at the Web Service. So there is no way to have two false values calculated. The ordering of the values that come back from the Web Service to the client does not matter since summing up all results requires no ordering.

Consistency of the result Again all our experiments were executed several times to verify the consistency by reducing the variance of our results. Each individual performance test was running 100 times with each packet loss setting. The diagrams presented in section 4.1.3.2 represent the intersection of all runs of the same packet loss setting.
CHAPTER 4. EXPERIMENTAL EVALUATION

Data transfer For each measurement we are sending 5000 Integer values to the Web Service. Afterwards the Web Service has to send 2500 results back to the client.

Packet loss To demonstrate the power of SCTP during periods of packet loss we used different loss rates of 0%, 1%, 2%, 3%, 4% and 8% for our experiments. We are again using a symmetric data loss which means that we have the same amount of packet loss from the client to the Web Service and the Web Service to the client. It seems a bit disproportionate to do some measurements with a drop rate of 8% but we argue that the tendency of SCTP can be demonstrated more clearly.

4.1.3.2 Results and Evaluation of the measurements

In comparison to the results of the file transfer experiments in 4.1.2, in the case of Web Services and independent transactions SCTP loses most of the performance advantage against TCP. Figure 4.10 and Figure 4.11 illustrates the result of the measurements.

Without packet loss all measurements are nearly the same. As far as packet loss happens, TCP outperforms SCTP sending ordered with one stream or 1000 streams. While we were observing the resource monitoring, it seems that the receiver side blocks further incoming data and so avoids better performance. Probably there is no more buffer space to save further out of order packets. This issue again should be solvable by overlooking the buffer management of SCTP.

The strengths of SCTP over TCP become apparent under loss conditions and the combination of multi-streaming and unordered delivery while transferring data as seen in the result. In this case we never noticed any blocking behavior on the receiver side. The performance improvement over TCP is not as clear as in the measurements of the file transfer in 4.1.2. Without heavy data loss (8%) the performance advantage is only in the range of some percent. The issue is the small data size. The required overhead of the data compared to any control information needed is too high and should be the explanation of
this behavior.

### 4.1.4 Number of Streams

As already described in section 2.4.4 within a single SCTP association up to 65,536 unidirectional streams can be used by either end for simultaneous data transfer. This ability has the same effect as using several different TCP connections but without its drawbacks. In our experiments we tried to figure out which number of streams would be the optimal one. The surprising result is, that we cannot deliver a guidance for this because this depends on more factors. In cases of little packet loss and/or small message size increasing the number of streams can lead to a decrease in performance. In this case, we recommend transferring the data over one stream. When increasing the filesize and loss rate, using more than one SCTP stream does indeed increase the performance. An explanation for the behavior in the scenario of little loss and small message size could be that the algorithm that

![Figure 4.10: Transfer time of TCP and SCTP using the Webservice](image-url)
Another problem could be the unoptimized buffer management of the lksctp library. Within an association, all streams belonging to a socket share the same receive buffer space. Through setting the `rcvbuf-policy` using `sysctl` we have the possibility to assign each stream its own receive buffer space with the result that instead of getting the expected performance gain we even experienced some performance loss.

### 4.2 Experimental Evaluation Summary

Chapter 4 presented the setup and results of our experiments and showed that the SCTP library works really efficient and performs very well. In particular for longer transfers and the incidence of packet loss, SCTP outperforms TCP by up to 70% in terms of throughput. For shorter file transfers, TCP performs better than SCTP also with the loss of packets. We also found out that using multi-streaming together with unordered delivery instead of ordered delivery
can cause an additional performance increase. Also for shorter messages sending between a client and a Web Service SCTP performs better than TCP, and it seems that the features of multi-streaming and unordered delivery play a major role in the increased performance. There is still a lot of room for improvement within the SCTP stack. The fact that SCTP has been standardized and the results prove that the performance is better than TCP, which is a good indication of more support of SCTP in the near future.
Chapter 5
Related Work

The performance improvement on the transport layer has been the topic of several research efforts—Especially for environments with high-loss rates and for applications with a large number of independent transactions, TCP often shows performance problems. There are a wide variety of projects that try to bypass the drawbacks of TCP by using SCTP as the transport protocol. Most of them are concentrating on solving the performance issue on a specific area like the usage of SCTP for MPI [26] or SCTP in FTP [26].

Only few researchers are trying to solve a wider range of performance issues as can be seen in [11, 27]. The rest of this chapter tries to give a short introduction and overview of related work. At the end we will discuss where this master thesis differs from the other presented works.

The authors of [26, 28] designed and evaluated a solution for using SCTP for the Message Passing Interface (MPI). MPI has become the de facto standard for constructing parallel applications in distributed environments due to the use of message passing. To improve the performance they have taken advantage of the multi-streaming feature of SCTP to provide a direct mapping from streams to MPI message tags. The context in an MPI program is to identify a set of processes that communicate with each other. This grouping of processes can be represented as a one-to-many socket in SCTP that establishes associations with that set of processes. SCTP can
map each association to the unique rank of a process within a context and thus use an association number to determine the source of a message arriving on its socket. Each association can have multiple streams which are independently ordered and this property directly corresponds with message delivery order semantics in MPI. In MPI, messages sent with different tag, rank and context to the same receiver are allowed to overtake each other. This permits direct mapping of streams to message tags that makes SCTP a perfect candidate due to its similarity at the conceptual level. Because of the implementation of the one-to-many sockets style they avoid the usage of expensive select system calls, which leads to increased scalability in large scale clusters. Additionally the MPI module also provides multi-homing for fault tolerance and resilience against network path failures.

Another project [29] focuses on improving data transfer on FTP using SCTPs multi-streaming feature. The authors pointed out that the current design of FTP includes a number of inefficiencies due to separate control- and data-connections and non-persistent data connection especially for multiple file transfers. They did some modifications to run FTP over SCTP which makes it more efficient with the available bandwidth and system resources. If an FTP client establishes an SCTP association with the server with two streams opened in each direction, the client and the server send control information (commands and replies) on their respective stream 0. All data (files, directory listings, and file namelists) are transferred over their respective stream 1. This approach maintains semantics for streams analogous to control and data connections in FTP over TCP. This results in the following improvements over TCP:

- Reduction of the number of connections by aggregating the control and data connections

- Reduction of the number of round-trip times for the setup/teardown procedure and command exchange
• Reduction of the number of connections a server must maintain.

• The usage of the bandwidth is more efficient by preserving the congestion window between file transfers.

Like the MPI project, this work again focuses on a single topic to gain better performance with ordered multi-streaming.

The probably most similar project to use SCTP insted of TCP can be found in [11]. The researcher designed a Transparent TCP-to-SCTP Translation Shim Layer. The concept is to translate TCP system calls into equivalent SCTP calls while the running application is unaware of the mapping from TCP to SCTP. To accomplish this, the protocol translation is inserted into the socket layer between the application and transport layers, leaving the structure of the existing network protocol stack intact. This project is primarily concentrating on the ability to provide fault tolerance to legacy applications by using SCTPs multi-homing support. The implementation of the shim layer library is located within the kernel. The actual version runs within the FreeBSD 4.10 kernel. So this project has the advantage that every application can use the library without changing and recompiling the source code. The drawback of a kernel space library includes a less portable design. For each operating system a separate shim implementation is needed. If an application generates a socket() system call in addition to creating the normal TCP socket a second hidden SCTP socket is created as well. It is termed hidden because it is created by the kernel but not exposed to the application. To have control over both sockets TCP-to-SCTP translation shim is controllable through a global on/off switch for the entire system. If SCTP is enabled the shim supports all standard socket API system calls and I/O functionality, including the ability to properly translate TCP socket options into their SCTP equivalents utilizing an implemented translation table within the library. As in our work, the shim layer library gives users and developers the ability to take
the advantage of SCTP without rewriting the sourcecode.

Recently another interesting and mentionable project has been published which can be found in [27]. The Structured Stream Transport (SST) is a new technology which appears to be promising. SST enhances the traditional stream abstraction with a hierarchical hereditary structure, allowing applications to create lightweight child streams from any existing stream supporting both reliable and best-effort delivery. SST is an experimental transport protocol designed to address the needs of modern applications that need to juggle many asynchronous communication activities in parallel, such as downloading different parts of a web page simultaneously and playing multiple audio and video streams at once. These lightweight streams incur neither 3-way handshaking delays on startup nor time-wait periods on close. Each stream offers independent data transfer and flow control, allowing different transactions to proceed in parallel without head-of-line blocking, but all streams share one congestion control context. The initial SST prototype takes the form of a user space library written in C++, which runs on Linux, BSD, Mac OS X, and Windows. It is very similar to SCTP but SST streams have individual flow control and can be dynamically created or destroyed.

The authors of [30, 31] discuss the usage of HTTP over SCTP as an alternative to the traditional HTTP over TCP protocol for the web traffic. They also focused on using multiple streams within an association. The tests were designed to simulate the behavior of a typical web browser downloading multiple embedded documents, pictures and links fetched by the browser. The results showed that SCTP using multiple streams improves the performance on heavily congested lines compared to SCTP using only a single stream.

So far SCTP uses multi-homing only for redundancy purposes, not
supporting simultaneous data transfer between two endpoints through the use of more end-to-end paths. As the authors in [9, 10, 32] describe, the technical requirements are already here to also use multi-homing to provide simultaneous transfer of data between two SCTP endpoints through more than one end-to-end path. Probably this feature will become another interesting performance improvement for SCTP in the future.

A drawback of these projects is that they are using either ordered multi-streaming [29, 33] for specific areas like MPI researching, or do not consider multiple SCTP streams because of a focus on fault tolerance using multi-homing. None of these projects take unordered delivery into account. This thesis, as the projects before, concentrates on increasing performance at the transport layer by taking advantage of SCTPs advanced features. Our work differs from the related work because we are the only ones using both multi-streaming and unordered delivery to increase performance for certain types of applications. Another fundamental difference to the other projects is the programming language. We wanted to design a library which is portable to all operating systems supporting SCTP. The perfect candidate for this is the JAVA programming language, which we used to implement our library.
Chapter 6

Conclusion and Future Challenges

6.1 Conclusion

We have just seen that SCTP has several features that make it a more suitable transport protocol than TCP for many applications. Even though SCTP is robust and provides extremely valuable features, it is still a relatively unknown and rarely used transport protocol. In this thesis we discussed the design, implementation and evaluation of an SCTP Java library that is able to exchange TCP with SCTP only demanding very little change within the source code, due to the usage of the abstract factory design pattern. The cornerstone of the library design was to avoid the head-of-line blocking problem present in many native TCP applications. To solve this issue we have combined the advantage of SCTPs features multi-streaming and unordered delivery.

In section 4 we have evaluated the library using real world applications and under typical network conditions. Afterwards the reported results of several experiments were compared to the performance of the native TCP application. The findings demonstrated that the migration from TCP to SCTP is not only a theoretical concept but also suitable for practical use. The strengths of SCTP over TCP through the usage of multi-streaming and unordered delivery become apparent under loss conditions as seen in the results.
With this Java library, developers and users now have a tool to try out and take advantage of the features of SCTP without rewriting the whole source code. While SCTP has many advantages over TCP and very few drawbacks we do not know if SCTP will replace TCP in the Internet. In the future it is very possible that the actual TCP/IP architecture will be replaced by another similar architecture where SCTP would be a perfect candidate for accepting this challenge.

6.2 And now? Upcoming challenges

While the implementation of the current library is working very well, there are some potential challenges for future work. On the one hand we have some performance issues and limitations to solve within the underlying Linux kernel library Lksctp, whose implementation is nowhere near as optimized as TCP. On the other hand there are some additional features that could lead to a nice improvement of the actual work.

6.2.1 Multi-homing

The library design does not take another great feature of SCTP into account, namely multi-homing. This feature opens up the possibility to assign more than one IP address to an SCTP endpoint. Up to date multi-homing is only used for redundancy purposes to support fault tolerance, not supporting simultaneous data transfer between two endpoints through the use of more end-to-end paths. If one connection breaks, SCTP automatically continues transferring traffic over an alternate path. Another service within the use of multi-homing is the possibility to completely hide a connection failure from the application. Even in situations where this feature is unavailable on one of the endpoints, due to the lack of a second network interface, it would be a nice extra for the actual library [34].
6.2.2 Half-closed-connection

The biggest drawback of SCTP over TCP is the unavailability of a half-closed state. The designer of SCTP simply removed this feature due to its extra complexity. There are some existing TCP applications that are in need of the half-closed state to operate correctly. One example is the file-transfer-protocol (FTP) that requires, under some circumstances, the state of a half-closed connection. Probably it is possible to simulate this half-closed state with the usage of multi-streaming.

6.2.3 PR-SCTP (Partially Reliable)

This thesis introduced a library to exchange TCP with SCTP. One enhancement might be to investigate the capability to use the library to exchange UDP with SCTP as well. The basic approach to fulfill this requirement is to extend the current library by implementing the availability of PR-SCTP. Like UDP, PR-SCTP has the ability to send data unordered and unreliable as well as ordered and unreliable which may be a benefit for some UDP applications. PR-SCTP uses the same congestion control mechanism avoidance as SCTP. Hence, PR-SCTP is TCP-friendly, while UDP is not. This kind of implementation for the actual library could be a very interesting topic for further research. In [35] the authors present an application already using PR-SCTP.
List of Figures

2.1 SCTP Data Unit with several chunks ...................... 7
2.2 Setting up a connection with TCP and SCTP .............. 10
2.3 Multi-homed SCTP connection ............................... 14
2.4 Multi-streaming: several streams within a SCTP association . 15
2.5 Shutting down a connection with TCP and SCTP ............ 16
2.6 The Data chunk ................................................. 17
3.1 Head-Of-Line blocking: Examples for TCP and SCTP ....... 25
3.2 Multi-Streaming: An example with one SCTP association and three streams ................................. 28
3.3 Java architecture for TCP communication ................... 32
3.4 Class diagram of the enhancement ........................... 34
3.5 Round-Robin scheduling ....................................... 34
3.6 Java architecture for SCTP communication .................. 39
4.1 Schematic of experimental setup for performance measurements 41
4.2 Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 500KB filesize versus packet loss .... 43
4.3 Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 5MB filesize versus packet loss .... 44
4.4 Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 25MB filesize versus packet loss .... 45
4.5 Transfer time of TCP and SCTP using 1000 streams ordered/unordered for 400MB filesize versus packet loss .... 45
LIST OF FIGURES

4.6 Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 500KB . . . . . . . . . . . . . . . . . . . . . 47
4.7 Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 5MB . . . . . . . . . . . . . . . . . . . . . 47
4.8 Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 25MB . . . . . . . . . . . . . . . . . . . . 48
4.9 Impact of packet loss TCP versus SCTP using 1000 streams ordered/unordered for filesize of 400MB . . . . . . . . . . . . . . . . . . . . 48
4.10 Transfer time of TCP and SCTP using the Webservice . . . . 51
4.11 Impact of packet loss for the Webservice . . . . . . . . . . . . 52
List of Tables

1.1 Layered architecture of the Internet Protocol suite . . . . . . 2

2.1 Services and features overview of SCTP, TCP and UDP . . . . 8
Listings

3.1  Multi-Streaming and Unordered Delivery .................. 35
3.2  Instantiate a SCTPOutputStream .......................... 37
3.3  Indicates the Begin of a data collection .................. 37
3.4  Indicates the End of a data collection .................... 37
3.5  Setting the new SocketFactory ............................ 38
Bibliography


Appendix A

Related Software

The Library was implemented, tested and written with the support of the following tools and environments.

**Operating System** Ubuntu 7.10 - the Gutsy Gibbon

**Kernel version** 2.6.22-14

**Lksctp** lksctp-tools 1.0.7

**Java** Java(TM) SE Runtime Environment build 1.6.0-03

**Dia** The diagramming software Dia [36] is an open source drawing application including several packages for different requirements like UML, flowing charts or network diagrams. The application supports exporting diagrams to a wide range of formats.

**Inkscape** Inkscape [37] is an Open Source vector graphics editor, with capabilities similar to Illustrator, CorelDraw, or Xara X, using the W3C standard Scalable Vector Graphics (SVG) file format.

**Eclipse** Eclipse [38] is an open source framework for software development including countless integrated functions. Due to its plug-in based structure Eclipse has a wide range of programming language support like Java, C/C++ or PHP.
APPENDIX A. RELATED SOFTWARE

**Gimp** The software *GIMP* [39] is a freely distributed program used for drawing and manipulating graphics. It supports all established file formats.

**Kile** *Kile* [40] is a LaTeX front end for KDE, that supports an easy way to write a LaTeX document.

**LaTeX** *LaTeX* [41] is a document preparation system for writing well designed work through the use of special formatting commands for the document.


**log4j** *log4j* [42] is a Java-based logging utility which is primarily used for debugging. It offers different standard logging levels that can be configured through files. The logs can be written to different locations.