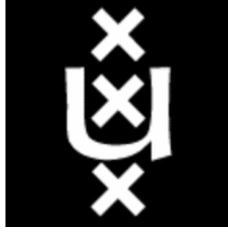


University of Amsterdam
System & Network Engineering

# Stream Control Transmission Protocol

by

Taarik Hassenmahomed

Coordinator: Dr. Ir. C. de Laat and Dr. Karst Koymans, University of Amsterdam.
Supervisor: Ronald van der Pol and Freek Dijkstra, SARA Computing and Networking Services.

Research Project 1

Academic year 2008–2009

# Contents

**Table of Contents** **ii**

**1 Introduction** **1**
1.1 Introduction . . . . . 1
1.2 Research . . . . . 2
1.3 Research Questions . . . . . 2

**2 History** **3**
2.1 Origin . . . . . 3
2.2 Standardizations in RFC . . . . . 3

**3 Features** **5**
3.1 What is it? . . . . . 5
3.2 Init . . . . . 7
3.3 Sack . . . . . 9
3.4 Gracefull shutdown . . . . . 9
3.5 Heartbeats . . . . . 9
3.6 Multistreaming . . . . . 9
3.7 Multihoming . . . . . 10
3.8 Dynamic Multihoming . . . . . 11

**4 Implementations** **13**
4.1 Operating systems . . . . . 13
4.2 Kernel API . . . . . 13

**5 The Test Set-Up** **15**
5.1 Hardware . . . . . 15
5.2 Operating system . . . . . 15
5.3 Software . . . . . 17
5.3.1 Dummynet . . . . . 17
5.3.2 Wireshark . . . . . 17

5.3.3 Netperf . . . . . . . . . . 18

**6 Test Methodology** **19**
6.1 Multistreaming . . . . . . . . . . 19
6.2 Multihoming . . . . . . . . . . 20
6.3 Dynamic Multihoming . . . . . . . . . . 20

**7 Test result and Analysis** **23**
7.1 Results SCTP vs TCP . . . . . . . . . . 23
7.1.1 Netperf . . . . . . . . . . 23
7.2 Results Multistreaming . . . . . . . . . . 23
7.2.1 Client-server programs . . . . . . . . . . 23
7.3 Results Multihoming . . . . . . . . . . 28
7.4 Results Dynamic Multihoming . . . . . . . . . . 30

**8 Conclusion** **33**
8.1 Technical conclusion . . . . . . . . . . 33
8.2 Research questions . . . . . . . . . . 33
8.3 Personal Conclusion . . . . . . . . . . 34

**Bibliography** **34**

# Chapter 1

# Introduction

## 1.1 Introduction

IANA[4] lists more than hundred different protocols able to operate in IP networks. For over thirty years TCP and UDP have been the standard transport layer protocols for connection oriented data transfer. Various variants of TCP and UPD have been propossed, which "improve" or combine features available in TCP and UPD, like the TCP variants Reno and Tahoe for better congestion avoidance and DCCP which provide access to congestion control mechanisms to datagrams and at the same time allows flow-based semantics like in TCP, but does not provide reliable in-order delivery.

In 1998 SCTP introduced itself as being a new protocol for Voice over IP, able to multistream data over a single connection and the first able to multihome to different addresses. The protocol also combined most features available in both TCP and UDP making it a new variant of TCP and UDP. This protocol was later altered to work directly on top of IP and got standardized by the IETF in RFC4960 as a new transport protocol and assigned protocol number 132 by IANA[4] .

This report is part of a research project for the Master Education in "System and Network Engineering at the University" of Amsterdam and has been carried out for "SARA Computing and Networking Services". This paper is the result of a practical investigation, with simple client and server programs, of some main features common to SCTP and not available in TCP or UDP.

## 1.2 Research

Because of the limited time of this research project and the various features of SCTP, the choice has been made to limit the investigation to the following main features of SCTP.

- Multistreaming
- Multihoming
- Dynamic Multihoming

This research has been carries out by writing simple client-server programs for the FreeBSD 7.0 operating system and analyzing the behavior and the performance of these main features.

## 1.3 Research Questions

From this limitation came the following research questions:

- Is it possible to use two separate connections with SCTP for load sharing with conservation of multihoming and multistreaming?
  - Does it matter if one of the lines has a larger latency, because of a longer round trip time?
  - How thus the performance of TCP and SCTP compare when using multiple streams?

# Chapter 2

# History

## 2.1 Origin

SCTP was originally not meant as an replacement to TCP, but developed to transport Voice over IP (VoIP). It all started with the public circuit-switched (SS7) telephone network (PSTN), which can be compared to the Internet which is a IP-based packet-switched networks, and the need to transport packet-based PSTN signaling over IP networks. PSTN and the Internet are very different from each other, which makes it difficult to transporting voice with IP networks. Because of this, a new working group, Signaling Transport (SIGTRAN) was founded in 1998 by the IETF, with the mission of addressing this problem. They started out with identifying the functionality and performance requirements needed for reliable telephone signals over IP. This resulted in the functional model of Fig. 2.1. Where SS7-IP gateways (SG,MGC,MG) handle the transport, translation and addressing of the SS7 signals through IP in a transparent message based way.

## 2.2 Standardizations in RFC

Initially, TCP was considered as the underlying protocol to transport VoIP data. UDP was not considered as a viable protocol, because it did not provide reliable connection oriented connections as TCP does, which was needed for the functional model. Analysis showed that TCP had some serious problems that made it unsuitable for PSTN signaling transport over IP networks. These problems had to do with at least it's lack or difficulty in multistreaming, multihoming, being stream oriented, lack of timer control, and vulnerability for SYN-attacks. Modifying or enhancing TCP to address these problems were considered to hard and various enhanced UDP versions were suggested, but none were accepted. In 1998

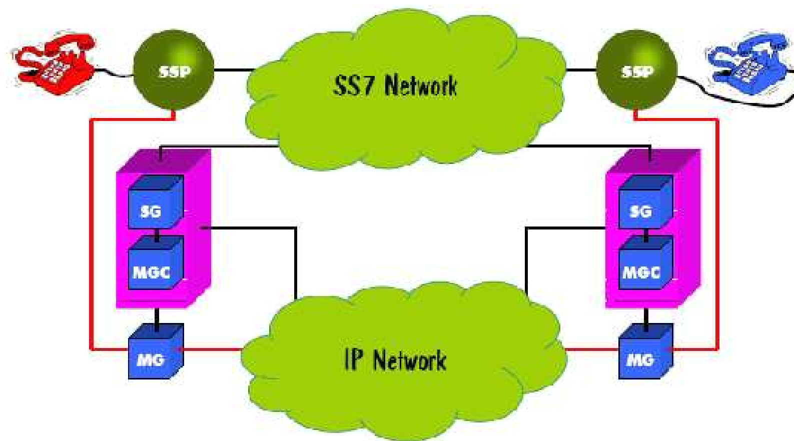



**Figure 2.1:** SIGTRAN functional mode [1]

Randall R. Stewart and Qiaobing Xie submitted a proposal "Multi-Network Datagram Transmission Protocol" (MDTP) which interested the SIGTRAN working group, because it had solutions for some of the TCP's weaknesses and in particular supported multihoming. After some big modifications it was used for the basis of SCTP. Which got revised to run directly on top of IP, giving it it's own protocol number space, moving it to the same (OSI) layer as TCP and giving it the ability to be used in other applications than only the telephony signaling transport. In October 2000 SCTP was published in the IETF as RFC2960. SCTP was designed to be easily extendible, and in September 2007, a revised version of RFC2960 was published which corrected many subjects and was registered as RFC4960.

# Chapter 3

# Features

The table in Fig. 3.1 gives an overview of features available in SCTP and compares the availability of these features with TCP and UDP. It shows that many features common to TCP or UPD are supported by SCTP.

## 3.1 What is it?

SCTP is, just like TCP, a reliable connection oriented transport protocol, except that instead of byte (or stream) oriented, SCTP is message oriented like UDP. It can send packet in order like TCP or unordered like UDP. Besides having the "best" of both worlds, it has a quite few new features of it own. What makes SCTP even more different from TCP and UDP is that it has it's own packet structure Fig. 3.2.

The packet has at least two parts, a common header (located in the first 64 bits) and one or more chunk parts. Multiple chunks can be bundled into one packet, except that data chunks can not be combined with INIT, INIT ACK, and SHUTDOWN COMPLETE chunks. This is because sending of data can only start with the packet comming after the INIT ACK, and not during or before. Data chunks can not be combined with SHUTDOWN COMPLETE as no data can be send after the SHUTDOWN COMPLETE chunk.

The source port and destination port identify the ports being used by sender and receiver,. The v-tag (short for Verification tag) can be seen as a signing mechanism to validate the sender of the packet and so preventing others from sending valid packets, without knowing the right v-tag. The Checksum is a CRC calculation over the whole packet. The receiver calculates the checksum to see if the packet is valid, otherwise it discards the packet.

A chunk contains either control information or user data, each chuck has a format

| Service/Feature | SCTP | TCP | UDP |
|---|---|---|---|
| Message-Oriented | yes | no | yes |
| Connection-oriented | yes | yes | no |
| Full duplex | yes | yes | yes |
| Reliable data transfer | yes | yes | no |
| Ordered data delivery | yes | yes | no |
| Unordered data delivery | yes | no | yes |
| Flow & Congestion control | yes | yes | no |
| Path MTU discovery | yes | yes | no |
| Fragmentation and bundling | yes | yes | no |
| ECN capable | yes | yes | no |
| Reach-ability check | yes | opt | no |
| Selective ACK's | yes | opt | no |
| Authentication | opt | opt | no |
| IPSec en TLS compatibility | yes | yes | yes |
| Partial-reliable data transfer | opt | no | no |
| Multistreaming | yes | no | no |
| Multihoming | yes | no | no |
| Dynamic Multihoming | opt | no | no |
| Stream Reset | opt | no | no |
| State Cookie | yes | no | no |
| Gracefull shutdown | yes | no | no |
| Pseudo-header for checksum | (uses v-tags) | yes | yes |
| CRC based checksum | yes | no | no |
| Time wait state | no | yes | n/a |

**Figure 3.1:** SCTP vs TCP vs UPD

| Bits | Bits 0 - 7 | 8 - 15 | 16 - 23 | 24 - 31 |
|---|---|---|---|---|
| +0 | Source port | | Destination port | |
| 32 | Verification tag | | | |
| 64 | Checksum | | | |
| 96 | Chunk 1 type | Chunk 1 flags | Chunk 1 length | |
| 128 | Chunk 1 data | | | |
| ... | ... | | | |
| ... | Chunk N type | Chunk N flags | Chunk N length | |
| ... | Chunk N data | | | |

**Figure 3.2:** SCTP packet structure [2]

containing a type, flag, length and data field. This is shown in Fig. 3.2 from bits 96 to 128.

At the time of writing there have been 19 registered chuck types (Fig. 3.3) The referred RFC's [8][9][10][11][12] contain detailed descriptions of each chunk type.

## 3.2 Init

During the initialization process SCTP uses a 4-way handshake instead of a 3-way used by TCP. (Fig. 3.4)

Host A sends an INIT chunk with an initial tag, and Host B acknowledges this message by echoing the Verification tag back to Host A. Host B includes its own initial tag in the INIT ACK. This mechanism validates the sender of the packet, as the one that received the initial tag, and doesn't start reserving resources for setting up a connection. This simple step protects SCTP against SYN attacks, which TCP can't handle. After an INIT ACK a valid connector will send a COOKIE parameter with all information needed to setup a connection which SCTP calls an association and asks for a COOKIE_ECHO. On return it will get a COOKIE_ACK containing the exact COOKIE that was send.

| ID Value | Chunk Type | Reference |
|---|---|---|
| 0 | Payload Data (DATA) | RFC4960 |
| 1 | Initiation (INIT) | RFC4960 |
| 2 | Initiation Acknowledgment (INIT ACK) | RFC4960 |
| 3 | Selective Acknowledgment (SACK) | RFC4960 |
| 4 | Heartbeat Request (HEARTBEAT) | RFC4960 |
| 5 | Heartbeat Acknowledgment (HEARTBEAT ACK) | RFC4960 |
| 6 | Abort (ABORT) | RFC4960 |
| 7 | Shutdown (SHUTDOWN) | RFC4960 |
| 8 | Shutdown Acknowledgment (SHUTDOWN ACK) | RFC4960 |
| 9 | Operation Error (ERROR) | RFC4960 |
| 10 | State Cookie (COOKIE ECHO) | RFC4960 |
| 11 | Cookie Acknowledgment (COOKIE ACK) | RFC4960 |
| 12 | Reserved for Explicit Congestion Notification Echo (ECNE) | RFC4960 |
| 13 | Reserved for Congestion Window Reduced (CWR) | RFC4960 |
| 14 | Shutdown Complete (SHUTDOWN COMPLETE) | RFC4960 |
| 15 | Authentication Chunk (AUTH) | RFC4895 |
| 128 | Address Configuration Acknowledgment (ASCONF-ACK) | RFC5061 |
| 132 | Padding Chunk (PAD) | RFC4820 |
| 192 | Forward TSN | RFC3758 |
| 193 | Address Configuration Change Chunk (ASCONF) | RFC5061 |

**Figure 3.3:** Chunk types[3]

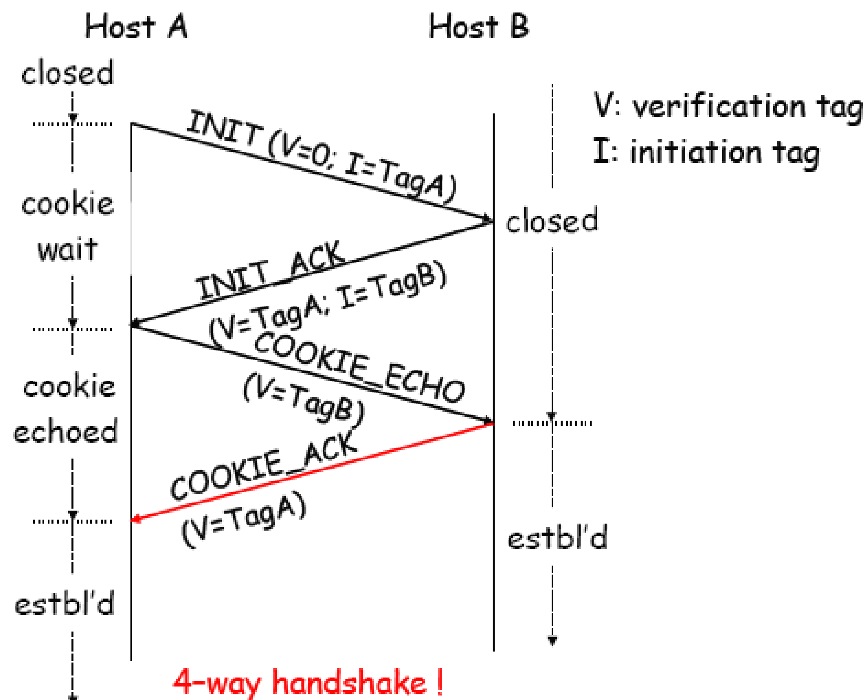



**Figure 3.4:** 4-way handshake[5]

## 3.3 Sack

SCTP has chunk numbers to identify a chunk in a packet, this chunk number is called a Transmission Sequence Number (TSN). Instead of acknowledging a packet, SCTP acknowledges chunks and does this with Selective Acks. These Acks don't get send for every packet, but for every other packet. De SACK contains cumulative TSN, to acknowledge all the chucks until the one listed. Beside this field, the SACK chunk also has 2 other fields, namely GAP BLOCK fields and Duplicate TSN fields. GAP BLOCK give the offset from the cumulative TSN, for missing chunks. So if cumulative TSN = 5 and GAP BLOCK = 2 this means chunks 3 is not acknowledged.

## 3.4 Gracefull shutdown

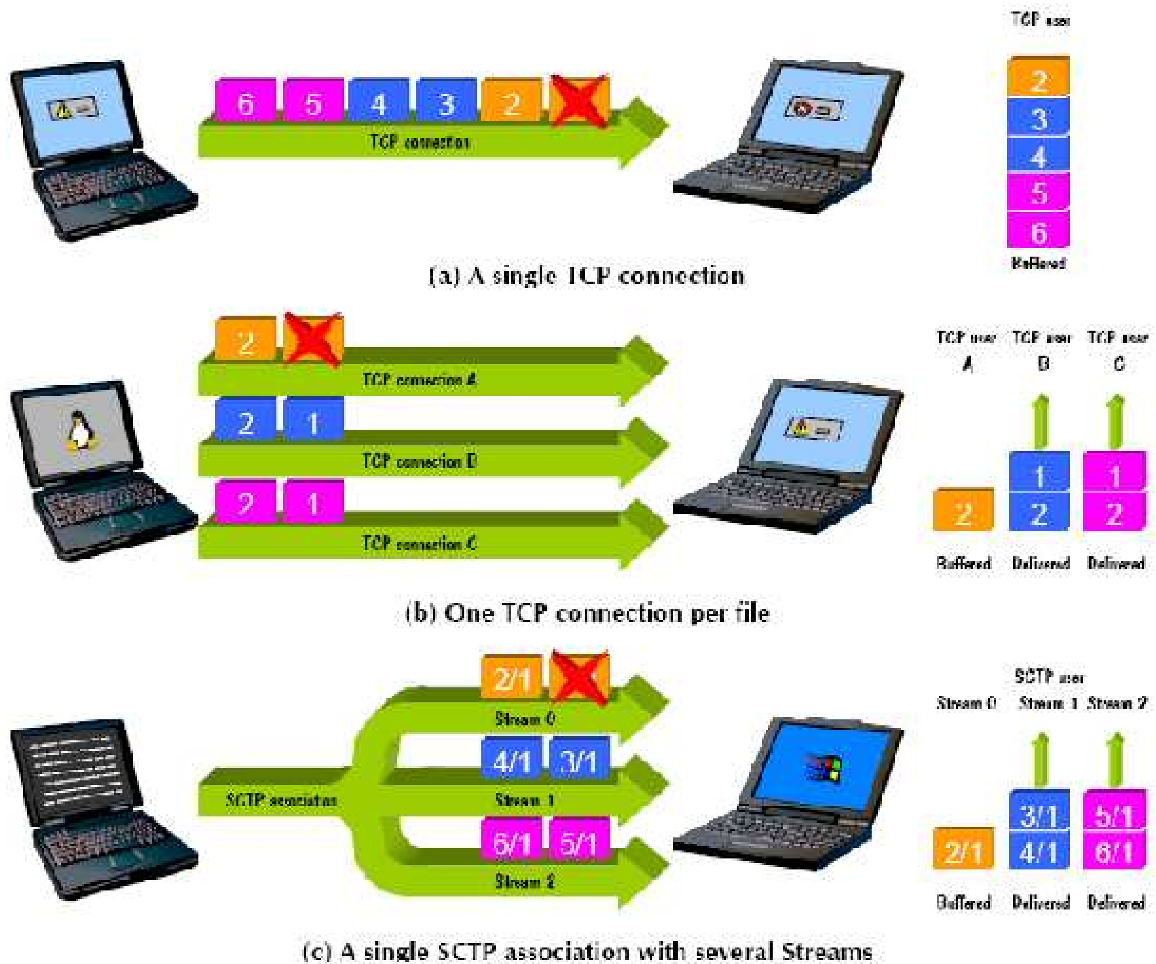
The shutdown process in SCTP uses a 3-message procedure for graceful shutdown, where every endpoint first acknowledges every arrived data chunk before ending the shutdown procedure. Because of this 3-message mechanism SCTP doesn't support half-open connections like TCP.

## 3.5 Heartbeats

SCTP periodically sends out HEARTBEAT chunks on alternative or non-active lines. Depending on the result of the return chunks, the status of a line is set to confirmed, unconfirmed or unreachable, with path failure (not using the line as long as it's unconfirmed) and endpoint failure (closing of the association) as possible result.

## 3.6 Multistreaming

Multistreaming is the ability to send multiple logically separated channel in one socket. This can be used to send multiple files in parallel with a single socket instead of using multiple sockets. This mechanism is mainly meant to avoid head-of-line blocking. In Fig. 3.5 you can see at (a) that the loss of the first packet containing data keeps the rest of the data in buffer until the correct arrival of the data and causing unnecessary delay. This happens because TCP works with byte streams and the packet missing could be any part of the data, since TCP doesn't manage boundaries. This problem is easily fixed by using a new connection for every file, but unfortunately this doesn't scale well and




(a) A single TCP connection

(b) One TCP connection per file

(c) A single SCTP association with several Streams

**Figure 3.5:** Multistreaming association (Head-of-Line Blocking)[1]

dependent on kernel limitations. Because SCTP is multistreaming, it doesn't need to open a new connection for every file, and just sends the files over different streams, and thereby solving the head-of-line blocking, at least as long as a stream is not being reused.

## 3.7 Multihoming

When a host has several network interfaces and can make use of multiple IP addresses at the same time, then that host is multihomed.. With SCTP you can connect multihomed hosts with each other and create a list of possible connections. The collection of these multiple IP addresses in combination with the port number are called the association. In a association only one primary connection can be active, the other connections are used as a redundant connections for when the primary should fail. An example of this van be seen

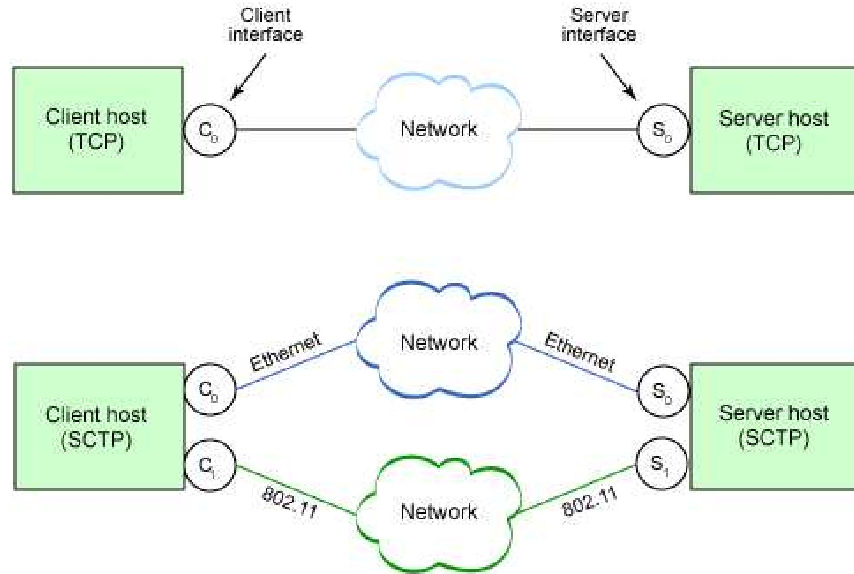



**Figure 3.6:** Multihoming association[13]

in Fig. 3.6.

## 3.8 Dynamic Multihoming

With the addition of Dynamic Multihoming you get the ability to add or remove IP addresses from the association, this making it possible to be more flexible when moving to another location. Another ability is to change the primary connection. This could be very handy for load sharing.

For more information on current features in SCTP not mention here I refer to their corresponding RFC.

# Chapter 4

# Implementations

## 4.1 Operating systems

SCTP is currently implemented in the kernel of the following operating systems

- AIX Version 5
- Crisco IOS 12
- Dragonfly BSD 1.4 and above
- FreeBSD 7.0 and above
- Linux 2.4/2.6
- HP-UX 11iv2 and above
- QNX Neutrino Realtime OS, 6.3.0 and above
- Sun Solaris 10

A user space library is available:

- The SCTP library (sctplib)

## 4.2 Kernel API

As the user space library did not compile on the test machines, the SCTP implementation in the FreeBSD 7.0 kernel was used for implementation of the client-server programs.

There were two versions of API programming styles namely:one-to-one and one-to-many.

- The one-to-one model is based on a one to one relationship between the socket and a SCTP association. This is the same as a standard TCP socket and connecting to a new association needs the disconnection of the first connection or the fork of a new socket.
- For the one-to-many model, there is a one-to-many relationship between the socket and the SCTP- associations making it a peer to peer type model, which resembles unconnected UDP sockets.

# Chapter 5

# The Test Set-Up

## 5.1 Hardware

I had the availability over two Dell servers to carry out different types of test with client en server programs. Each server had two dual core Intel Xeon processors and two network interfaces. Two Nortel switches provided the network connectivity. Each of the two network interfaces of the hosts was connected to a different switch. One of the switches was connected to the Internet and the other was connected to a private LAN. The private LAN could either connect the two hosts directly together Fig. 5.1 or via a transatlantic connection with a much longer round trip time (about 200 ms) illustrated in Fig. 5.2. The interfaces connected to the Internet had public IP addresses, while the interfaces connected to the LAN used private IP addresses.

## 5.2 Operating system

The operating system in use for these test is FreeBSD 7.0. FreeBSD is generally regarded as reliable and robust and the current version is, and [6] quotes:

> "the reference implementation for the new IETF Stream Control Transmission Protocol (SCTP) protocol, intended to support VoIP, telecommunications, and other applications with strong reliability and variable quality transmission through features such as multi-path delivery, fail-over, and multi- streaming."

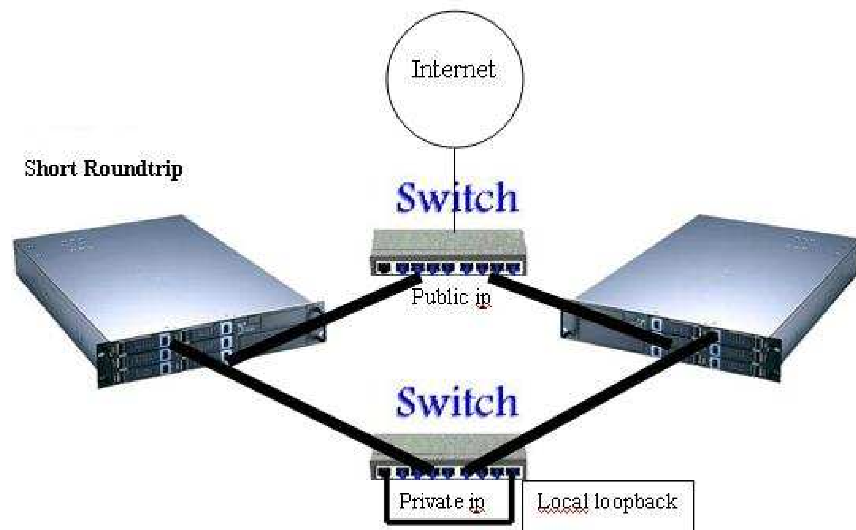



**Figure 5.1:** Scenarion 1: Short roundtrip

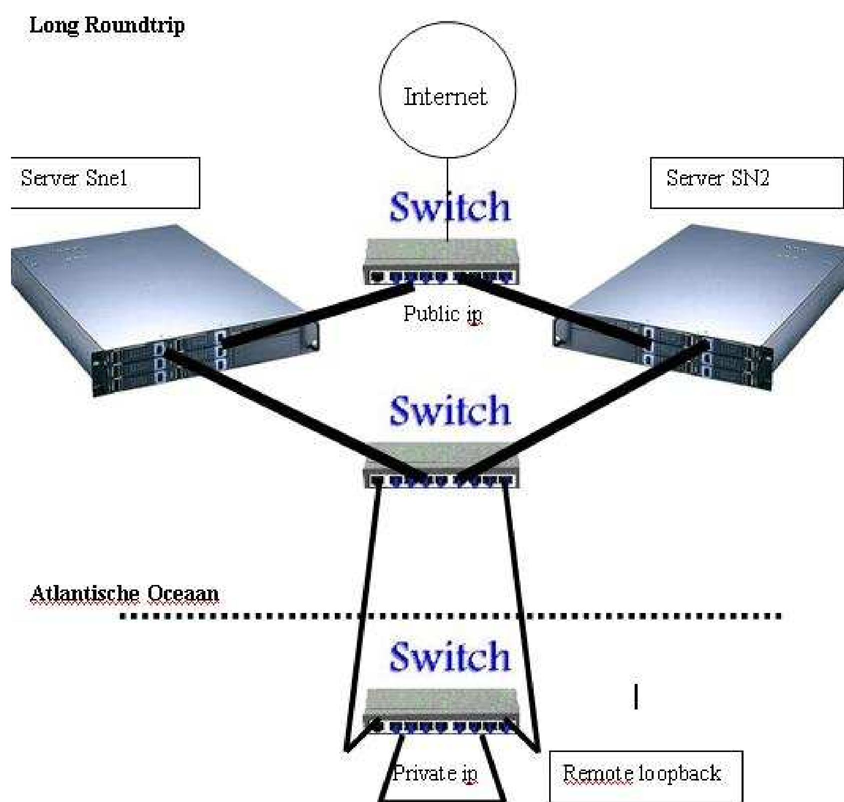



**Figure 5.2:** Scenarion 2: Long roundtrip

## 5.3 Software

### 5.3.1 Dummynet

For simulating lower bandwidth and random packet loss I made use of Dummynet [7]. This application gives you the ability to manipulate latency, packet loss and bandwidth for not only IP TCP and UDP, but also SCTP. FreeBSD 7.0 had Dummynet installed, but not enabled. To enable Dummynet I fist had to add the following lines to /etc/rc.conf:

```
firewall_enable="YES"
firewall_type="OPEN"
```

After this I had to reboot the system. After reboot I could load Dummynet with:

```
kldload dummynet
```

To use dummynet, you must use the ipfw command, with this command you can create new pipes, which represent connections and configures them the way you want. For example:

```
ipfw pipe 1 config plr 0.1
ipfw pipe 2 config bw 100Kbit/s
```

The first line configures a packet loss rate of 10% and the second line limits the bandwidth to 100Kbit/s To use these pipes you need to attach them to the FreeBSD firewall. For example:

```
ipfw add 1 pipe 1 sctp from 192.168.1.100 to any
ipfw add 2 pipe 2 sctp from any to 192.168.1.100
```

The first command gives only the SCTP packets going from 192.168.1.100 a packet loss of 10% and the second line reduces the bandwidth of SCTP coming through 192.168.1.100 only 100Kbit/s

### 5.3.2 Wireshark

To examine the SCTP packet I used Wireshark, a free packet sniffer. Wireshark can be used for network troubleshooting, analysis, software and communications protocol development, and education. SCTP is fully supported by wireshark and just like tcpdump it analyses packets, only Wireshark does this graphically and can do much more, like information sorting and filtering options. To use Wireshark I needed to enable X11 Forwarding and use xterm to run wireshark. After this you need to choose an interface and give the option to only capture SCTP.

### 5.3.3 Netperf

Netperf was used to test the performance of SCTP. Netperf is a benchmark that can be used with many different types of networking. Netperf supports SCTP. To use the Netperf to perform test you need to run the Netserver program on the remote server. After that you can you start

```
netperf -H remote_server -t protocol -other_parameters -- -test_parameters
```

You can also choose how long you want the test to take, the size of the files transmitted and buffer options.

# Chapter 6

# Test Methodology

## 6.1 Multistreaming

Multiple file transfer tests will be performed to investigate the throughput performance of SCTP in different situations. Making use of the scenario's in Fig. 5.1 and Fig. 5.1. In both scenarios we will only use the private IP interfaces. The variables for the different situations are:

- file size: 16KB, 4 MB, 100MB, 1GB
- file count: 1, 5 and 10
  - file are sent with same size
  - file are sent with different size
- stream count: 1, 5 and 10

With Dummynet we could adapt the following variables:

- bandwidth: 1 Mbps, 10 Mbps, 100 Mbps, 1 Gbps
- random packet loss: 0%, 5%, 10%

Netperf was used to test the throughput for each round trip time and was used to compare SCTP with TCP.

## 6.2 Multihoming

For the investigation of multihoming we again used the scenario's in Fig. 5.1 and Fig. 5.2. Only now we are using both the public and private IP interfaces. The multihoming feature of SCTP detect connection failure and uses the alternate connection. To introduce failure of one of the multihomed connection, we disconnected a cable from one of the interfaces. This disconnection has the same characteristics as network congestion or other problems where the other host (almost) isn't reachable. This disconnection can be done in various ways and are listed below.

For every scenario we analyzed with Wireshark what happens when:

1. Transfering files with the primary path
    (a) and this path is blocked.
        i. and comes back some time later.
    (b) the secondary path is blocked,
        i. and comes back some time later.
2. the primary path would fail during transmission,
    (a) and later the secondary line on the other side

## 6.3 Dynamic Multihoming

To investigate dynamic multihoming, we implemented the sctp_bindx, sctp_connectx function and SCTP_SET_PEER_PRIMARY_ADDR parameter for sctp_setsockopt() in the client server programs. Afterwards we wanted to analyze with Wireshark what happens when:

1. While transfering files with the primary connection
    (a) we add a secondary path to the association.
        i. afterwards the transmission on the primary path gets blocked.
        ii. or the primary address is changed to the other line
2. Besides investigating these options, we want to try load sharing,
    (a) By continuous changing the primary address in a round robin way, where we use each line for a limited time and change the primary address again. We can analyze the possible advantage and see what happens when a path gets blocked.

(b) Alternatively we could look at the possibilities of concurrent load sharing, instead of the proposed round robin strategy.

# Chapter 7

# Test result and Analysis

## 7.1 Results SCTP vs TCP

### 7.1.1 Netperf

The first test compared the performance of TCP and SCTP by transmitting files of different size across the 200ms link (see Fig. 5.2). Table 7.1 lists the measured throughputs and request/responds time. The same test was performed for the short roundtrip link (see Fig. 5.1) and put in table 7.2.

Analysis with Wireshark showed that Netperf isn't using the adventage of multistreaming and makes this test only represetative for one stream. With the default settings netperf tries to send larger files than SCTP can handle and causes to lag and eventually abort the transfer and end with an Netperf error. When using smaller files you get readable measurements. Netperf shows that SCTP performs roughly 3 times as bad as TCP. A possible explanation may be the relative immaturity of the SCTP implementation, but we have not investigated this any further.

## 7.2 Results Multistreaming

### 7.2.1 Client-server programs

We wrote a server and client program to test the multistreaming and multihoming features of SCTP. The source code of these programs is available in Appendix A. The code is based on example code published by HP [14], code available at [15] and the internet draft at [16] . The following log shows the output of a run of the programs and demonstrates the

| | Troughput in Mbps | | transmission rate in seconds | |
|---|---|---|---|---|
| | TCP | SCTP | TCP | SCTP |
| filesize in bytes | Throughput | Throughput | Request/Responds | Request/Responds |
| 1 | 3.7 | 0.1 | 4.7 | 4.7 |
| 100 | 15.3 | 2.1 | | |
| 1000 | 13.3 | 5.9 | | |
| 2000 | 14.3 | 5.9 | | |
| 2500 | 11.4 | error | | |
| 16000 | 13.6 | error | | |
| default | 13.9 | error | | |

**Table 7.1:** Performance of TCP vs SCTP 1 (as measured by netperf)

| | Troughput in Mbps | | transmission rate in seconds | |
|---|---|---|---|---|
| | TCP | SCTP | TCP | SCTP |
| filesize in bytes | Throughput | Throughput | Request/Responds | Request/Responds |
| 1 | 3.8 | 1.39 | 5432.6 | 3456.8 |
| 100 | 252.8 | 118.2 | | |
| 1000 | 764.4 | 227.1 | | |
| 2000 | 852.5 | 238.7 | | |
| 2500 | 937.1 | 328.1 | | |
| 16000 | 938.9 | error | | |
| default | 906.9 | error | | |

**Table 7.2:** Performance of TCP vs SCTP 2 (as measured by netperf)

multistreaming feature of SCTP

```
CLIENT
Connected to [192.168.138.2]
There are 1 remote addresses and they are:
145.146.100.136
There are 1 local addresses and they are:
145.146.100.135
SCTP_ASSOC_CHANGE: COMMUNICATION UP, assoc=0x28ced7cd, error=0, instr=3 outstr=3
SCTP_PEER_ADDR_CHANGE: SCTP_ADDR_CONFIRMED, addr=145.146.100.136, assoc=0x28ced7cd, error=0
<S>: [1]Send1 over stream 1
temp:0x28ced7cd new:0x28ced7cd:
From str:1 sqn:0 [assoc:0x28ced7cd]:
[1]Send1 over stream 1
<S>: [2]Send2 over stream 2
From str:2 sqn:0 [assoc:0x28ced7cd]:
[2]Send2 over stream 2
<S>: [2]Send3 over stream 2
From str:2 sqn:1 [assoc:0x28ced7cd]:
[2]Send3 over stream 2
SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=0x28ced7cd, error=0, instr=3 outstr=3
Over !!

SERVER
^C ./a.out 4001
{one-to-many}: Waiting for associations ...
There are 1 remote addresses and they are:
145.146.100.135
There are 1 local addresses and they are:
145.146.100.136
SCTP_ASSOC_CHANGE: COMMUNICATION UP, assoc=0x1fec53a3, error=0, instr=5 outstr=5
SCTP_PEER_ADDR_CHANGE: UNKNOWN, addr=145.146.100.135, assoc=0x20000000, error=3276800
unknown type: 50851
Send on str:1 sqn:0 [assoc:0x1fec53a3]:
[1]Send1 over stream 1
Send on str:2 sqn:0 [assoc:0x1fec53a3]:
[2]Send2 over stream 2
Send on str:2 sqn:1 [assoc:0x1fec53a3]:
[2]Send3 over stream 2
SCTP_SHUTDOWN_EVENT: assoc=0x1fec53a3
SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=0x1fec53a3, error=0, instr=5 outstr=3
```

The log shows that a connection is setup with 5 incoming and 5 outgoing streams (instr=5 outstr=5), we send different messages over the different streams from the client

| Packetloss% | 1 stream | 3 streams | 5 streams |
|---|---|---|---|
| 0 | 0.008s | 0.008s | 0.005s |
| 10 | 1s | 1s | 1s |
| 30 | 28s | 35s | 50s |
| 50 | abort | abort | abort |

**Table 7.3:** Multiple stream performance (random packetloss)

and get an echo back from the sever. We see that the messages are received on the chosen stream (indicated by str:x). The messages for the same stream get a different sequence number (indicated by sqn:x).

Analysis with Wireshark shows the following data chunks:

```
DATA chunk(ordered, complete segment, TSN: 1394501593, SID: 1, SSN: 0, PPID: 0, payload length: 23 bytes)

DATA chunk(ordered, complete segment, TSN: 1394501594, SID: 2, SSN: 0, PPID: 0, payload length: 23 bytes)

DATA chunk(ordered, complete segment, TSN: 1394501595, SID: 2, SSN: 1, PPID: 0, payload length: 23 bytes)
```

Here you can see that in the packets every data chunk has it own Transmission Sequence Number (TSN) and that it contains SID (Stream ID numbers) and SSN (Stream Sequence numbers).

The next test performed was a transfer test of 5 x 16Kb bin file over different stream with random SCTP packetloss and SCTP bandwidth limitations initiated with Dummynet.

Being able to avoid head-of-line blocking (because of packetloss), which slow the delivery of data to the user in TCP, five streams should theoretically be faster than a single stream. The test result in table 7.3 shows that with 16Kbit files five streams seem to have a higher bandwidth than with a single stream. With the introduction of random packetloss this changes and it seem that with five streams it takes longer for packets to be successfully handled than before introduction of random packetloss. Which contradicts the theory that multiple streams working faster when introduction of packetloss.

The next table 7.4 shows the bandwidth test The test with different bandwidth showed that the number of streams does not affect the total used bandwidth, it remains lower than the bandwidth used by TCP.

When sending files you need load them in a buffer, when trying to load buffers of larger files than 200Kbit for storing larger files in the program memory gives an error. The only thing you could do is send smaller buffers containing fragments of the file. So instead of letting SCTP handle the fragmentation, it was needed to program this into the client

| Bandwidth(Mbps) | 1 stream | 3 streams | 5 streams |
|---|---|---|---|
| 100 | 0.01s | 0.01s | 0.01s |
| 10 | 0.08s | 0.08s | 0.08s |
| 1 | 0.8s | 0.8s | 0.8s |
| 0.1 | 8s | 8s | 8s |
| 0.01 | 94s | 93s | 94s |

**Table 7.4:** Multiple stream performance (bandwidth)

server code. Self fragmentation of files make the stream sequence number change for every fragment. This could be solved by sending the data with an UNORDERED flag.

How this fragmentation works in SCTP with ordered and unordered chunks can be seen next by an analysis of the packets in Wireshark.

```
<...>
DATA chunk(ordered, first segment, TSN: 1339775518, SID: 0, SSN: 0, PPID: 0, payload length: 1452 bytes)
DATA chunk(ordered, middle segment, TSN: 1339775519, SID: 0, SSN: 0, PPID: 0, payload length: 1452 bytes)
DATA chunk(ordered, middle segment, TSN: 1339775528, SID: 0, SSN: 0, PPID: 0, payload length: 1452 bytes)
DATA chunk(ordered, last segment, TSN: 1339775529, SID: 0, SSN: 0, PPID: 0, payload length: 412 bytes)
<..>
----
<...>
DATA chunk(ordered, first segment, TSN: 1339775530, SID: 1, SSN: 0, PPID: 0, payload length: 1452 bytes)
<..>
----
<...>
DATA chunk(ordered, last segment, TSN: 1339775589, SID: 5, SSN: 0, PPID: 0, payload length: 412 bytes)
<..>
```

Here you can see how SCTP fragments the chunks in ordered transfer, by setting the fragmentation flag to either first, middle or last segment. And keeps the SSN the same for every chunk.

```
<..>
DATA chunk(ordered, first segment, TSN: 2767140403, SID: 0, SSN: 0, PPID: 0, payload length: 1452 bytes)
DATA chunk(ordered, last segment, TSN: 2767140540, SID: 0, SSN: 0, PPID: 0, payload length: 1076 bytes)
<..>
----
<...>
DATA chunk(ordered, last segment, TSN: 2767143162, SID: 0, SSN: 19, PPID: 0, payload length: 1076 bytes)
<..>
```

When sending larger files files, you need to fragment the file a buffer, when sending the buffers you receive the chunks with new sequence numbers.

```
<...>
DATA chunk(unordered, first segment, TSN: 2017087117, SID: 0, SSN: 0, PPID: 0, payload length: 1452 bytes)
DATA chunk(unordered, last segment, TSN: 2017089876, SID: 0, SSN: 0, PPID: 0, payload length: 1076 bytes)
<...>
```

| | short roundtrip connection | | long roundtrip connection | |
|---|---|---|---|---|
| filesize in bytes | packet count | time in seconds | packet count | time in seconds |
| 16K | 26 | 0.01 | 21 | 0.63 |
| 4M | 4050 | 0.31 | 4412 | 7.3 |
| 100M | 94193 | 8.83 | 110032 | 127 |
| 1G | 1012522 | 132.45 | not tested | not tested |

**Table 7.5:** short vs long roundtrip

When using the UNORDERED flag you still have fragmentation by SCTP, only now the SSN doesn't change, which keeps the sending file intact.

The next test was sending large files over the short round trip connection and the long roundtrip connection, the result in table 7.5.

from this test we can see that SCTP send larger files at a lower speed than smaller files. With a longer roundtrip it even takes on average 15 times longer in sending the same file on a short round trip connection. With Wireshark can be seen that all packet arrived correctly and none had to be resend, they just arrive with larger intervals than with the short connection. Why this happens is not clear, since both connections are at least gigabit.

Because of time lost in getting the client and server programs to work properly, there wasn't time left to do the first set of test for large files and have enough time left to investigate multihoming. Therefore the choice was made to continue with multihoming.

## 7.3 Results Multihoming

To initiate multihoming, all you need is to program the server to bind with and ADDR_ANY address, this enables the program to bind the SCTP socket with all the interfaces on the current host. Connecting with another host will automatically enable multihoming. The test setup has a public and a private IP. Connecting with the private IP (e.g. IP addresses in the 10.0.0.0/8 or 192.168.0.0/16 ranges) will create a connection with the public IP address (routable via the Internet) and add this connection as a redundant alternate IP address for the association. Connecting with the public IP, will unfortunately discard the private address. An example of this can be seen in the client-server output of section 7.2.1 where only one address pair is established.

To investigate mulithoming Dummynet was used to artificially create a SCTP packetloss of 100%. This to remove the need to disconnect cables from interfaces.

With the multihomed version of the client and server code (available in Appendix A) we did the following tests. We made the client loop messages every couple of seconds and did the following steps:

To make the primary connection fail we introduced -Packet loss 100% after 20 seconds To set the primary connection back -Packet loss 0% after 65 seconds And then letting it fail again -Packet loss 100% at 105 seconds

The corresponding client and server gave the following log. Here you can see that SCTP notifies changes in the availability of the addresses.

```
CLIENT
Connected to [172.17.2.136]
There are 2 remote addresses and they are:
172.17.2.136
145.146.100.136
There are 2 local addresses and they are:
172.17.2.135
145.146.100.135
<S>: From str:1 sqn:3 [assoc:0x5b046bc3]:Send10
<S>: SCTP_PEER_ADDR_CHANGE: ADDRESS UNREACHABLE, addr=172.17.2.136
<S>: From str:0 sqn:7 [assoc:0x5b046bc3]:Send21
<S>: SCTP_PEER_ADDR_CHANGE: ADDRESS AVAILABLE, addr=172.17.2.136
<S>: From str:2 sqn:13 [assoc:0x5b046bc3]:Send41
<S>: SCTP_PEER_ADDR_CHANGE: ADDRESS UNREACHABLE, addr=172.17.2.136 assoc=0x5b046bc3, error=4
SCTP_SHUTDOWN_EVENT: assoc=0x5b046bc3
SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=0x5b046bc3, error=0, instr=3 outstr=3

SERVER
{one-to-many}: Waiting for associations ...
unknown type: 58187
There are 2 remote addresses and they are:
172.17.2.135
145.146.100.135
There are 2 local addresses and they are:
172.17.2.136
145.146.100.136
Send on str:1 sqn:3 [assoc:0x26b54cb4]:Send10
SCTP_PEER_ADDR_CHANGE: UNKNOWN, addr=172.17.2.135, assoc=0x20000000
Send on str:0 sqn:7 [assoc:0x26b54cb4]:Send21
SCTP_PEER_ADDR_CHANGE: UNKNOWN, addr=172.17.2.135, assoc=0x20000000
Send on str:2 sqn:13 [assoc:0x26b54cb4]:Send41
SCTP_PEER_ADDR_CHANGE: UNKNOWN, addr=172.17.2.135, assoc=0x20000000
Send on str:0 sqn:14 [assoc:0x26b54cb4]:Send42
```

| Wireshark | Time line of messages send/received | | | |
|---|---|---|---|---|
| | | | | |
| client bg0 | | 6->21 | | 37->42 |
| server bg0 | | 6->21 | | 37->42 |
| client bg1 | 1->5 | | 22->36 | |
| server bg1 | 1->5 | | 22->36 | |

**Table 7.6:** wireshark timeline of multihoming test

```
SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=0x26b54cb4, error=0
```

With Wireshark you can see what actually happens. In table 7.6 you can see a simplified table of the message numbers that were send every 3 seconds. The actual connection can be seen in the table 7.7.

What table 7.7 shows is that when SCTP on the client interface 1 (on the left) detects a failure, it immediately send the data through the alternate client interface 2 (on the right),which can be seen at time 22. In the mean time SCTP is sending heartbeat chunks (indicated by HB in the table) to interface 1 to try to update the status of the connections. In the log of the client-server program in section 7.3 you can see that the connection gets a different status, namely:ADDRESS UNREACHABLE. Once the primary interface is back online the status changes to ADDRESS AVAILABLE and it takes over the transfer. The secondary goes back to his idle mode and receives heartbeats periodically.

Doing the same investigation over the secondary link doesn't abort to the connection, the only thing that happens is that SCTP keeps sending heartbeats out, and keeps waiting for it's reply. The status of the secondary link changed just as the primary to ADDRESS UNREACHABLE when blocking the connection and to ADDRESS AVAILABLE after deblocking.

## 7.4 Results Dynamic Multihoming

For dynamic multihoming, function sctp_bindx was needed. The implementation of this function is supported in FreeBSD, the sctp_bindx man page states you need to supply a pointer to an array of addresses of the appropriate structure. Trying this results in a invalid argument. The same thing happens with the sctp_connectx function. To change the primary address we need to set the SCTP_SET_PEER_PRIMARY_ADDR parameter for sctp_setsockopt() (set socket option function). This function needs an association id

and a adress to set primary. The result of this function also returned an error. One of my supervisors tried to implement the same in a one2one version (available in Appendix A), but this resulted in the same errors. The code used for dynamic multihoming is available in Appendix A

| Time | CLIENT(interface 1) | | SERVER | Time | CLIENT(interface 2) | | SERVER |
|---|---|---|---|---|---|---|---|
| | | | | 1.1 | 5129 | HB —> | 6000 |
| | | | | 1.3 | 5129 | <— HB_ACK | 6000 |
| 5.2 | 5129 | SACK —> | 6000 | | | | |
| 22.5 | 5129 | <— DATA | 6000 | 22.5 | 5129 | DATA —> | 6000 |
| 24.5 | 5129 | <— DATA | 6000 | 22.5 | 5129 | SACK —> | 6000 |
| 36.3 | 5129 | <— HB | 6000 | 38.9 | 5129 | <— SACK | 6000 |
| 38.9 | 5129 | <— DATA | 6000 | 40.9 | 5129 | <— DATA | 6000 |
| 41.9 | 5129 | <— DATA | 6000 | 41.9 | 5129 | <— HB | 6000 |
| 56.8 | 5129 | <— DATA | 6000 | 41.9 | 5129 | SACK —> | 6000 |
| 61.8 | 5129 | <— DATA | 6000 | 41.9 | 5129 | HB_ACK –> | 6000 |
| | | | | 56.8 | 5129 | DATA —> | 6000 |
| | | | | 56.8 | 5129 | <— SACK | 6000 |
| | | | | 60.8 | 5129 | <— DATA | 6000 |
| | | | | 61.8 | 5129 | <— HB | 6000 |
| | | | | 61.8 | 5129 | HB_ACK —> | 6000 |
| | | | | 61.8 | 5129 | SACK —> | 6000 |
| | | | | 62.8 | 5129 | <— DATA | 6000 |
| | | | | | | | |
| 69.3 | 5129 | <— HB | 6000 | 69.3 | 5129 | HB —> | 6000 |
| 69.3 | 5129 | HB_ACK —> | 6000 | 69.3 | 5129 | <— HB_ACK | 6000 |
| 75.6 | 5129 | DATA —> | 6000 | | | | |
| 75.6 | 5129 | <-SACK DATA | 6000 | | | | |
| 76.6 | 5129 | SACK —> | 6000 | | | | |
| | | | | | | | |

**Table 7.7:** wireshark output of multihoming test

# Chapter 8

# Conclusion

## 8.1 Technical conclusion

This investigation has examined and implemented the workings of SCTP multihoming and multistreaming. Measurements with Netperf showed that the performance of SCTP in single stream mode is much lower than that of TCP, although we could not determine if this was a implementation issue or caused by the protocol itself. This could also not be determined because of the lack of test tools that work with SCTP on FreeBSD. Trying multistreaming with self made client server programs have shown the simple use of multistreaming where you simply supply a stream number and the message gets send on that stream. Unfortunatly it didn't show the advantage of multiple streams for file transfer, but a disadvantage when introducing packet loss and long round trip connections. For multihoming it has shown the flexibility of SCTP when it comes to path failure and the use of multiple local and peer IP addresses.

This investigation has not been successful in finding out what happens when multiple files of different sizes are send with multistreaming. How SCTP reacts to multiple large files when sending these files as one message without using the UNORDERED flag. Or succesfully implementing dynamic multihoming and investigating it's workings.

## 8.2 Research questions

- With the current version of SCTP is not possible to use two separte connection for concurrent load sharing, because there is only one connections active during transmission. This could be changed, to the secondary connection with the SCTP_SET_PEER_PRIMARY_ parameter, but could not be confirmed, due to implementation problems.

- When faced with longer roundtrip connections SCTP apparently slows down, just like TCP.
- To test the performance of SCTP we need better test tools, currently only netperf was found to test this on FreeBSD for single single streams. There is still a need for a tool that can handle multiple streams.

## 8.3 Personal Conclusion

SCTP has been an interesting protocol, the extra features makes it more flexible than TCP, but because of it's short age SCTP is not much used in applications or sites. Since the birth of SCTP there have been a lot of extensions to TCP and many more other transport protocols, making SCTP fade away in the background. I regret the fact that I could not successfully investigate all features of SCTP, but this would make it become a hobby instead of a research project. In conclusion I would like to thank Freek Dijkstra and Ronald van der Pol for their support in getting through the problems I had with writing the server and client code.

# Bibliography

[1] Iván Arias Rodríguez Stream Control Transmission Protocol The design of a new reliable transport protocol for IP networks 12th of February, 2002

[2] Colin M.L. Burnett `http://en.wikipedia.org/wiki/SCTP`

[3] `http://www.iana.org/assignments/sctp-parameters`

[4] `http://www.iana.org/assignments/protocol-numbers`

[5] Randall Stewart & Prof. Paul Amer Stream Control Transmission Protocol (SCTP) `http://www.sctp.org/bsdasia_sctp_intro.pdf`

[6] `http://www.freebsd.org/features.html`

[7] Luigi Rizzo `http://info.iet.unipi.it/~luigi/ip_dummynet/`

[8] `http://tools.ietf.org/html/rfc3758`

[9] `http://tools.ietf.org/html/rfc4820`

[10] `http://tools.ietf.org/html/rfc4960`

[11] `http://tools.ietf.org/html/rfc4895`

[12] `http://tools.ietf.org/html/rfc5061`

[13] M. Tim Jones Better networking with SCTP 28 Feb 2006 `http://www.ibm.com/developerworks/linux/library/l-sctp/`

[14] `http://docs.hp.com/en/5992-4578/apas02.html`

[15] W. Richard Stevens, Bill Fenner and Andrew M. Rudoff

`UNIXÂ® Network Programming Volume 1, Third Edition: The Sockets Networking API`

[16] R. Stewart The Resource Group K. Poon Sun Microsystems, Inc. M. Tuexen Univ. of Applied Sciences Muenster V. Yasevich HP P. Lei Cisco Systems, Inc. November 3, 2008 Sockets API Extensions for Stream Control Transmission Protocol (SCTP) http://tools.ietf.org/html/draft-ietf-tsvwg-sctpsocket-18

# Appendix A

The first part with functions are used in the one2manyclient/server code (need to insert in original code, to make program work)

```
}//print address from double pointer
static void
print_addr(struct sockaddr **addr, int numaddr)
{
int i=0;
int num = numaddr;
char *s;
struct sockaddr *sa;
while(i<num)
{
sa =(struct sockaddr *)addr;
if(sa->sa_family == AF_INET)
    inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr), s, INET6_ADDRSTRLEN);
else
    inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr), s, INET6_ADDRSTRLE
printf("%s\n",s);
i+=1;
addr = addr + sizeof(sa);
}
}

//Get peer and local adresses
static void
get_pandl_adress(int fd, struct sctp_assoc_change *sac)
{
```

```
int num_rem, num_loc;
struct sockaddr **sal,**sar;
/* Gather and print peer addresses */
num_rem = sctp_getpaddrs(fd,sac->sac_assoc_id,(struct sockaddr **)&sar);
printf("There are %d remote addresses and they are:\n", num_rem);
if(num_rem>0)
{
print_addr(sar,num_rem);
sctp_freepaddrs((struct sockaddr *)sar);
}
/* Gather and print local addresses */
num_loc = sctp_getladdrs(fd,sac->sac_assoc_id,(struct sockaddr **)&sal);
printf("There are %d local addresses and they are:\n", num_loc);
if(num_loc>0)
{
print_addr(sal,num_loc);
sctp_freeladdrs((struct sockaddr *)sal);
}
}

//handle notifications
static void
handle_event(int fd, void *buf)
{
struct sctp_assoc_change *sac;
struct sctp_send_failed *ssf;
struct sctp_paddr_change *spc;
struct sctp_remote_error *sre;
struct sctp_shutdown_event *sse;
union sctp_notification *snp;
char addrbuf[INET6_ADDRSTRLEN];
const char *str;
const char *ap;
struct sockaddr_in *sin;
struct sockaddr_in6 *sin6;
```

```
snp = buf;
switch (snp->sn_header.sn_type)
{
case SCTP_ASSOC_CHANGE:
sac = &snp->sn_assoc_change;
switch(sac->sac_state)
{
case SCTP_COMM_UP:
  str = "COMMUNICATION UP";
  get_pandl_adress(fd,sac);//Get peer and local adresses
  break;
case SCTP_COMM_LOST:
  str = "COMMUNICATION LOST";
  break;
case SCTP_RESTART:
  str = "RESTART";
  get_pandl_adress(fd,sac);//Get peer and local adresses
  break;
case SCTP_SHUTDOWN_COMP:
  str = "SHUTDOWN COMPLETE";
  break;
case SCTP_CANT_STR_ASSOC:
  str = "CAN'T START ASSOC";
  break;
default:
  str = "UNKNOWN";
  break;
}
printf("SCTP_ASSOC_CHANGE: %s, assoc=0x%x, error=%hu, instr=%hu "
"outstr=%hu\n", str,(uint32_t)sac->sac_assoc_id, sac->sac_error,
sac->sac_inbound_streams, sac->sac_outbound_streams);
break;

case SCTP_SEND_FAILED:
ssf = &snp->sn_send_failed;
printf("SCTP_SEND_FAILED: assoc=0x%x error=%d\n",(uint32_t)ssf->ssf_assoc_id, ssf->ss
```

```
break;

case SCTP_PEER_ADDR_CHANGE:
spc = &snp->sn_paddr_change;
switch(spc->spc_state)
{
case SCTP_ADDR_AVAILABLE:
  str = "ADDRESS AVAILABLE";
  break;
case SCTP_ADDR_UNREACHABLE:
  str = "ADDRESS UNREACHABLE";
  break;
case SCTP_ADDR_REMOVED:
  str = "ADDRESS REMOVED";
  break;
case SCTP_ADDR_ADDED:
  str = "ADDRESS ADDED";
  break;
case SCTP_ADDR_MADE_PRIM:
  str = "ADDRESS MADE PRIMARY";
  break;
case SCTP_ADDR_CONFIRMED:
  str = "SCTP_ADDR_CONFIRMED";
  break;
default:
  str = "UNKNOWN";
  break;
}
if (spc->spc_aaddr.ss_family == AF_INET)
{
sin = (struct sockaddr_in *)&spc->spc_aaddr;
ap = inet_ntop(AF_INET, &sin->sin_addr, addrbuf, INET6_ADDRSTRLEN);
}
else
{
```

```
sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
ap = inet_ntop(AF_INET6, &sin6->sin6_addr, addrbuf, INET6_ADDRSTRLEN);
}
printf("SCTP_PEER_ADDR_CHANGE: %s, addr=%s, assoc=0x%x, error=%d\n", str, ap,(uint32_
break;

case SCTP_REMOTE_ERROR:
sre = &snp->sn_remote_error;
printf("SCTP_REMOTE_ERROR: assoc=0x%x error=%d\n",(uint32_t)sre->sre_assoc_id, sre->s
break;

case SCTP_SHUTDOWN_EVENT:
sse = &snp->sn_shutdown_event;
printf("SCTP_SHUTDOWN_EVENT: assoc=0x%x\n",(uint32_t)sse->sse_assoc_id);
break;

default:
printf("unknown type: %hu\n", snp->sn_header.sn_type);
break;
}
}
```

one2manyclient_filetransfer.c

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

#define ISTREAMS 5
```

```
#define OSTREAMS 5
#define SLEEPINT 0

int debug=0;

int main(argc, argv)
int argc;
char **argv;
{
int fd, sz, len, msg_flags,istreams,ostreams;
sctp_assoc_t associd;
int check = 0;
struct sockaddr_in serv_addr;
struct sctp_event_subscribe event;
char buf[256];
struct sctp_sndrcvinfo sri;
struct sctp_initmsg initmsg;
union sctp_notification *snp;
struct sctp_assoc_change *sac;

//retry?
if (argc < 3) {
printf ("\nUsage: <%s> <remote-ip> <port> \n\n", argv[0]);
return -1;
}

//setup one2many socket
fd = socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
if (fd == -1) {
printf ("\nUnable to create socket \n");
return -1;
}

/* Specify that a maximum of streams that will be available per socket */
  memset( &initmsg, 0, sizeof(initmsg) );
  initmsg.sinit_num_ostreams = OSTREAMS;
```

```
  initmsg.sinit_max_instreams = ISTREAMS;
  sz = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,&initmsg, sizeof(initmsg) );

//initialise and connect remote ip en chosed port
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons (atoi(argv[2]));
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);

if (connect (fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
printf ("\nUnable to connect to remote server [%s] !! \n\n", inet_ntoa(serv_addr.sin_
return -1;
} else
{
printf ("\nConnected to [%s]\n", inet_ntoa(serv_addr.sin_addr));
}

/* Enable some events */
memset (&event, 0, sizeof(event));
event.sctp_data_io_event = 1;
event.sctp_association_event = 1;
event.sctp_address_event = 1;
event.sctp_send_failure_event = 1;
event.sctp_shutdown_event = 1;

if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0)
{
               perror("setevent failed");
               exit(1);
        }

memset (&sri, 0, sizeof(sri));
memset (buf, 0, sizeof(buf));


// declare a FILE pointer
```

```
FILE *file;

// open a file for reading
   file = fopen(argv[3], "rb");

if(file==NULL)
{
printf("Error: can’t open file.\n");
return 1;
}
fseek (file , 0 , SEEK_END);
int fileSize_s1 = ftell (file);
rewind (file);

char *buf_s1[fileSize_s1+1];
int result_s1 = fread(buf_s1, 1, fileSize_s1, file);


//get comm up notification
sleep(5);
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
{
    handle_event(fd, buf);
    snp = buf;
    if(snp->sn_header.sn_type==SCTP_ASSOC_CHANGE)
    {
sac = &snp->sn_assoc_change;
associd = sac->sac_assoc_id;
istreams = (u_int) sac->sac_inbound_streams;
ostreams = (u_int) sac->sac_outbound_streams;

    }
}
if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
```

```
/* Loop while waiting for data */
while (1) {
sleep(SLEEPINT);

                sri.sinfo_stream = check;//strtol(buf+1, NULL, 0);
if((u_int)sri.sinfo_stream >= ostreams)
{
  sri.sinfo_stream = ((u_int)sri.sinfo_stream % ostreams);
}
printf("ost%d\n",ostreams);

int result = strlen(buf);
len = sizeof(serv_addr);

//stop after "check" sends with MSG_EOF else send normal message
if (check == 5)
{
printf ("Sending shutdown event");
strncpy (buf_s1, "exit", 4);
sctp_sendmsg(fd, buf_s1,result_s1,(struct sockaddr *)&serv_addr, len, sri.sinfo_ppid,
}
else
sz = sctp_sendmsg (fd, buf_s1, result_s1, (struct sockaddr *)&serv_addr, len, 0, 0,sr
if (debug)
printf ("sctp_sendmsg:[%d,e:%d]\n", result_s1, errno);

//store and later compare the association_id, to see if the association has been shut
if(associd != 0)
{
associd = sri.sinfo_assoc_id;
printf ("temp:0x%x new:0x%x:\n",(u_int)associd, (u_int)sri.sinfo_assoc_id);
}
else
{
if(associd != sri.sinfo_assoc_id)
{
```

```
printf ("temp:0x%x new:0x%x:\n",(u_int)associd, (u_int)sri.sinfo_assoc_id);
printf("Association closed by server\n");
break;
}

}

//exit clause for loop
if (!strncmp(buf_s1, "exit", 4))
{
do
{
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
    handle_event(fd, buf);
}
while(strcmp(buf , "exit", 4));

if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
if (sz <= 0)
printf("Break");
printf ("From str:%d sqn:%d [assoc:0x%x] %d bytes:\n", sri.sinfo_stream,sri.sinfo_ssn
sleep(5);
break;
}
else
printf ("<S>: ");
check += 1;
memset (buf, 0, sizeof(buf));

}

printf ("\nOver !!\n");
//close socket
close(fd);
```

```
return 0;
}
```

one2manyserver_filetransfer.c

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

#define ISTREAMS 5
#define OSTREAMS 5
#define SLEEPINT 0
#define BUFFLEN  200000

int debug=1;

int main(argc, argv)
int argc;
char **argv;
{
int fd, sz, len, msg_flags,istreams,ostreams;
sctp_assoc_t associd;
int check = 0;
struct sockaddr_in serv_addr;
struct sctp_event_subscribe event;
char buf[256];
char buf_s1[BUFFLEN];
struct sctp_sndrcvinfo sri;
struct sctp_initmsg initmsg;
union sctp_notification *snp;
```

```
struct sctp_assoc_change *sac;

//retry?
if (argc < 3) {
printf ("\nUsage: <%s> <remote-ip> <port> \n\n", argv[0]);
return -1;
}

//setup one2many socket
fd = socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
if (fd == -1) {
printf ("\nUnable to create socket \n");
return -1;
}

/* Specify that a maximum of streams that will be available per socket */
  memset( &initmsg, 0, sizeof(initmsg) );
  initmsg.sinit_num_ostreams = OSTREAMS;
  initmsg.sinit_max_instreams = ISTREAMS;
  sz = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,&initmsg, sizeof(initmsg) );

//initialise and connect remote ip en chosed port
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons (atoi(argv[2]));
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);

if (connect (fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
printf ("\nUnable to connect to remote server [%s] !! \n\n", inet_ntoa(serv_addr.sin_
return -1;
} else
{
printf ("\nConnected to [%s]\n", inet_ntoa(serv_addr.sin_addr));
}

/* Enable some events */
```

```
memset (&event, 0, sizeof(event));
event.sctp_data_io_event = 1;
event.sctp_association_event = 1;
event.sctp_address_event = 1;
event.sctp_send_failure_event = 1;
event.sctp_shutdown_event = 1;

if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0)
{
               perror("setevent failed");
               exit(1);
        }

memset (&sri, 0, sizeof(sri));
memset (buf, 0, sizeof(buf));
memset (buf_s1, 0, sizeof(buf_s1));


// declare a FILE pointer
FILE *file;

// open a file for reading
   file = fopen(argv[3], "rb");

if(file==NULL)
{
printf("Error: can't open file.\n");
return 1;
}
//fseek (file , 0 , SEEK_END);
//int fileSize_s1 = ftell (file);
//rewind (file);

//char *buf_s1[fileSize_s1+1];
//int result_s1 = fread(buf_s1, 1, fileSize_s1, file);
```

```
//get comm up notification
sleep(5);
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
{
    handle_event(fd, buf);
    snp = buf;
    if(snp->sn_header.sn_type==SCTP_ASSOC_CHANGE)
    {
sac = &snp->sn_assoc_change;
associd = sac->sac_assoc_id;
istreams = (u_int) sac->sac_inbound_streams;
ostreams = (u_int) sac->sac_outbound_streams;

    }
}
if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
/* Loop while waiting for data */
while (1) {
//gives wrong association id, look at function at top
//get_associds(fd);

//loop message
//strncpy (buf, "Sendx", 4);
//sprintf(buf+4, "%d", check);

//set and loop stream number

                sleep(SLEEPINT);

                sri.sinfo_stream = check;//strtol(buf+1, NULL, 0);
if((u_int)sri.sinfo_stream >= ostreams)
{
  sri.sinfo_stream = ((u_int)sri.sinfo_stream % ostreams);
```

```
}
printf("ost%d\n",ostreams);

int result = strlen(buf);
int result_s1 = sizeof(buf_s1);
len = sizeof(serv_addr);

//stop after "check" sends with MSG_EOF else send normal message
if (check == 1)
{
printf ("Sending shutdown event");
strncpy (buf, "exit", 4);
sctp_sendmsg(fd, buf,result,(struct sockaddr *)&serv_addr, len, sri.sinfo_ppid,(sri.s
}
else
{
while(fread(buf_s1, sizeof(buf_s1), 1, file)!= 0 )
{
sz = sctp_sendmsg (fd, buf_s1, result_s1, (struct sockaddr *)&serv_addr, len, 0, (sri
if (debug)
printf ("sctp_sendmsg:[%d,e:%d]\n", result_s1, errno);
}
}
/*
//reveive messages, and handle notification messages
do
{
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
    handle_event(fd, buf);
}
while(msg_flags & MSG_NOTIFICATION);
if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
if (sz <= 0)
printf("Break");
```

```
printf ("From str:%d sqn:%d [assoc:0x%x] %d bytes:\n", sri.sinfo_stream,sri.sinfo_ssn
//printf("%.*s\n",sz,buf);
*/
//store and later compare the association_id, to see if the association has been shut
if(associd != 0)
{
associd = sri.sinfo_assoc_id;
printf ("temp:0x%x new:0x%x:\n",(u_int)associd, (u_int)sri.sinfo_assoc_id);
}
else
{
if(associd != sri.sinfo_assoc_id)
{
printf ("temp:0x%x new:0x%x:\n",(u_int)associd, (u_int)sri.sinfo_assoc_id);
printf("Association closed by server\n");
break;
}

}

//exit clause for loop
if (!strncmp(buf, "exit", 4))
{
do
{
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
    handle_event(fd, buf_s1);
if(sz>0 && !strncmp(buf, "exit", 4))
break;
}
while(msg_flags & MSG_NOTIFICATION);

if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
if (sz <= 0)
```

```
printf("Break");
printf ("From str:%d sqn:%d [assoc:0x%x] %d bytes:\n", sri.sinfo_stream,sri.sinfo_ssn

sleep(5);
break;
}
else
printf ("<S>: ");
check += 1;
memset (buf, 0, sizeof(buf));
memset (buf_s1, 0, sizeof(buf_s1));
}

printf ("\nOver !!\n");
//close socket
close(fd);
return 0;
}
```

one2manyserver_filetransfer_unordered.c

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

#define BUFLEN 20000
#define IDLETIME 500
#define ISTREAMS 5
#define OSTREAMS 5
```

```
int debug=1;

int main(int argc, char **argv)
{
int fd, sz, len, msg_flags;
sctp_assoc_t temp_associd;
int idleTime = IDLETIME;
struct sockaddr_in sin[1];
struct sockaddr cli_addr;
struct sctp_event_subscribe event;
char buf[BUFLEN];
struct sctp_sndrcvinfo sri;
struct sctp_initmsg initmsg;

if (argc < 2)
{
printf ("\nUsage: <%s> <port>\n\n", argv[0]);
return -1;
}

//setup one2many socket
if ((fd = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) == -1)
{
perror("socket");
exit(1);
}

/* Specify that a maximum of streams that will be available per socket */
  memset(&initmsg, 0, sizeof(initmsg));
  initmsg.sinit_num_ostreams = OSTREAMS;
  initmsg.sinit_max_instreams = ISTREAMS;
  sz = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,&initmsg, sizeof(initmsg) );

//initialise and bind any ip en chosed port
sin->sin_family = AF_INET;
sin->sin_port = htons(atoi(argv[1]));
```

```
sin->sin_addr.s_addr = inet_addr("145.146.100.136");

if (bind(fd, (struct sockaddr *)sin, sizeof (*sin)) == -1)
{
perror("bind");
exit(1);
}



/* Enable all events */
memset (&event, 0, sizeof(event));
event.sctp_data_io_event = 1;
event.sctp_association_event = 1;
event.sctp_address_event = 1;
event.sctp_send_failure_event = 1;
event.sctp_peer_error_event = 1;
event.sctp_shutdown_event = 1;

if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0) {
perror("setevent failed");
exit(1);
}

/* Set associations to auto-close in 20 seconds of
* inactivity
*/
if (setsockopt(fd, IPPROTO_SCTP, SCTP_AUTOCLOSE,&idleTime, 4) < 0) {
perror("setsockopt SCTP_AUTOCLOSE");
exit(1);
}

/* Allow new associations to be accepted */
if (listen(fd, 1) < 0) {
perror("listen");
exit(1);
}
```

```
memset (&sri, 0, sizeof(sri));
memset (buf, 0, sizeof(buf));

printf ("{one-to-many}: Waiting for associations ...\n");

// Wait for new associations
while(1)
{
len = sizeof (struct sockaddr);
//reveive messages, and handle notification messages
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&cli_addr, &len, &sri, &m

if (debug)
printf ("sctp_recvmsg:[%d,e:%d,fl:%X]:\n ", sz, errno, msg_flags);
if (sz <= 0)
break;

//check for nofticiation
if (msg_flags & MSG_NOTIFICATION)
{
handle_event(fd, buf);
continue;
}
printf ("Received on str:%d sqn:%d [assoc:0x%x] %d bytes:\n", sri.sinfo_stream, sri.s

if (!strncmp(buf, "exit", 4))
{
 sz = sctp_sendmsg (fd, buf, 4, (struct sockaddr *) &cli_addr, len, sri.sinfo_ppid, s
}
memset (buf, 0, sizeof(buf));
}
close(fd); //close socket (comes here when there is an error)
}
```

one2manyserver_multihoming.c

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

int debug=1;

int main(argc, argv)
int argc;
char **argv;
{
int fd, sz, len, msg_flags,istreams,ostreams;
sctp_assoc_t temp_associd,associd;
int check = 0;
struct sockaddr_in serv_addr;
struct sctp_event_subscribe event;
char buf[256];
struct sctp_sndrcvinfo sri;
struct sctp_initmsg initmsg;
union sctp_notification *snp;
struct sctp_assoc_change *sac;

//retry?
if (argc < 3) {
printf ("\nUsage: <%s> <remote-ip> <port> \n\n", argv[0]);
return -1;
}
```

```
//setup one2many socket
fd = socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
if (fd == -1) {
printf ("\nUnable to create socket \n");
return -1;
}

/* Specify that a maximum of streams that will be available per socket */
  memset( &initmsg, 0, sizeof(initmsg) );
  initmsg.sinit_num_ostreams = 5;
  initmsg.sinit_max_instreams = 5;
  sz = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,&initmsg, sizeof(initmsg) );

//initialise and connect remote ip en chosed port
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons (atoi(argv[2]));
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);

if (connect (fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
printf ("\nUnable to connect to remote server [%s] !! \n\n", inet_ntoa(serv_addr.sin_
return -1;
} else
{
printf ("\nConnected to [%s]\n", inet_ntoa(serv_addr.sin_addr));
}

/* Enable some events */
memset (&event, 0, sizeof(event));
event.sctp_data_io_event = 1;
event.sctp_association_event = 1;
event.sctp_address_event = 1;
event.sctp_send_failure_event = 1;
event.sctp_shutdown_event = 1;

if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0)
```

```
{
               perror("setevent failed");
               exit(1);
        }

memset (&sri, 0, sizeof(sri));
memset (buf, 0, sizeof(buf));

//get comm up notification
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
{
    handle_event(fd, buf);
    snp = buf;
    if(snp->sn_header.sn_type==SCTP_ASSOC_CHANGE)
    {
sac = &snp->sn_assoc_change;
associd = sac->sac_assoc_id;
istreams = (u_int) sac->sac_inbound_streams;
ostreams = (u_int) sac->sac_outbound_streams;

    }
}
  sleep(5);
/* Loop while waiting for data */
while (1) {

//loop message
strncpy (buf, "Sendx", 4);
sprintf(buf+4, "%d", check);

//set and loop stream number
sri.sinfo_stream = check;//strtol(buf+1, NULL, 0);
if((u_int)sri.sinfo_stream >= ostreams)
{
  sri.sinfo_stream = ((u_int)sri.sinfo_stream % ostreams);
```

```
}

sz = strlen(buf);
len = sizeof(serv_addr);

//stop after "check" sends with MSG_EOF else send normal message
if (check == 50)
{
strncpy (buf, "exit", 4);
sctp_sendmsg(fd, buf,sz,(struct sockaddr *)&serv_addr, len, sri.sinfo_ppid,(sri.sinfo
}
else
sz = sctp_sendmsg (fd, buf, sz, (struct sockaddr *)&serv_addr, len, 0, 0,sri.sinfo_st
if (debug)
printf ("sctp_sendmsg:[%d,e:%d]\n", sz, errno);

//reveive messages, and handle notification messages
sz = sctp_recvmsg (fd, buf, sizeof(buf), (struct sockaddr *)&serv_addr, &len, &sri, &
if(msg_flags & MSG_NOTIFICATION)
    handle_event(fd, buf);
if (debug)
printf ("sctp_recvmsg:[%d,e:%d] ", sz, errno);
if (sz <= 0)
printf("Break");
printf ("From str:%d sqn:%d [assoc:0x%x]:", sri.sinfo_stream,sri.sinfo_ssn,(u_int)sri
printf("%.*s\n",sz,buf);

//store and later compare the association_id, to see if the association has been shut
if(check == 0)
{
temp_associd = sri.sinfo_assoc_id;
printf ("temp:0x%x new:0x%x:\n",(u_int)temp_associd, (u_int)sri.sinfo_assoc_id);
}
else
{
if(temp_associd != sri.sinfo_assoc_id)
```

```
{
printf ("temp:0x%x new:0x%x:\n",(u_int)temp_associd, (u_int)sri.sinfo_assoc_id);
printf("Association closed by server\n");
break;
}

}

//exit clause for loop
if (!strncmp(buf, "exit", 4))
break;
else
printf ("<S>: ");
check += 1;
memset (buf, 0, sizeof(buf));

//just wait to see message
/*
if (check == 1)
                {
                        struct sockaddr_in addr[1];
                        addr->sin_family = AF_INET;
                        addr->sin_port = htons (atoi(argv[2]));
                        addr->sin_addr.s_addr = inet_addr("192.168.138.18");
                        sz = sctp_connectx(fd,(struct sockaddr *)&addr,1,(sctp_assoc_
if(sz == -1)
{
perror("sctp connectx");
exit(2);
}
                }
*/
sleep(3);
}

printf ("\nOver !!\n");
```

```
//close socket
close(fd);
return 0;
}
```

one2manyserver_multihoming.c

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

#define BUFLEN  100
#define THQ_ADD 0xc001
#define THQ_REM 0xc002

int debug=1;

//try to do dynamic mulithoming, gives a invallid argument everytime.
static void
bind_address(int fd, int port)
{
int numb = 1;
//struct sockaddr *tempaddr;
struct sockaddr_in sen[1];
//socklen_t sock_ln;
sen->sin_family = AF_INET;
sen->sin_port = htons(port);
//sen->sin_addr.s_addr = inet_addr("145.146.100.136");
sen->sin_addr.s_addr = inet_addr("192.168.138.2");
//getsockname(fd, tempaddr, &sock_ln);
```

```
//sen->sin_port = ((struct sockaddr_in *)tempaddr)->sin_port;
int sz = sctp_bindx(fd,(struct sockaddr *)&sen, numb, THQ_REM);
printf("adress:%s,type:0x%x\n",inet_ntoa(((struct sockaddr_in *)&sen[0])->sin_addr),S

printf("bindSZ=%d, errno=%d\n",sz, errno);
if (sz == -1)
{
perror("sctp bindx");
exit(2);
}
}



static void
change_primary(int fd, sctp_assoc_t assoc_id, int port)
{

struct sockaddr_storage ssp;
struct sockaddr_in *sip;
sip = (struct sockaddr_in*)&ssp;

        sip->sin_port = htons(port);
sip->sin_addr.s_addr = inet_addr("145.146.100.136");
//sip->sin_addr.s_addr = inet_addr("192.168.138.2");
struct sctp_setprim newaddr;
newaddr.ssp_assoc_id = assoc_id;
newaddr.ssp_addr = ssp;
if (setsockopt(fd, IPPROTO_SCTP, SCTP_PRIMARY_ADDR, &newaddr, sizeof(newaddr)) < 0)
{
perror("setsockopt SCTP_PRIMARY_ADDR");
exit(1);
}
}

int main(int argc, char **argv)
{
```

```
int fd, sz, len, msg_flags,in;
sctp_assoc_t temp_associd;
int idleTime = 500;
struct sockaddr_in sin[1];
struct sockaddr cli_addr;
struct sctp_event_subscribe event;
char buf[100];
struct sctp_sndrcvinfo sri;
struct sctp_initmsg initmsg;
struct sctp_status status;

if (argc < 2)
{
printf ("\nUsage: <%s> <port>\n\n", argv[0]);
return -1;
}

//setup one2many socket
if ((fd = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) == -1)
{
perror("socket");
exit(1);
}

/* Specify that a maximum of streams that will be available per socket */
  memset(&initmsg, 0, sizeof(initmsg));
  initmsg.sinit_num_ostreams = 3;
  initmsg.sinit_max_instreams = 3;
  sz = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,&initmsg, sizeof(initmsg) );

//initialise and bind any ip en chosed port
sin->sin_family = AF_INET;
sin->sin_port = htons(atoi(argv[1]));
sin->sin_addr.s_addr = htonl(INADDR_ANY);
//sin->sin_addr.s_addr = inet_addr("192.168.138.2");
//sin->sin_addr.s_addr = inet_addr("145.146.100.136");
```

```
if (bind(fd, (struct sockaddr *)sin, sizeof (*sin)) == -1)
//if (sctp_bindx(fd, (struct sockaddr *)sin, 1, SCTP_BINDX_ADD_ADDR) == -1)
{
perror("bind");
exit(1);
}


/* Enable all events */
memset (&event, 0, sizeof(event));
event.sctp_data_io_event = 1;
event.sctp_association_event = 1;
event.sctp_address_event = 1;
event.sctp_send_failure_event = 1;
event.sctp_peer_error_event = 1;
event.sctp_shutdown_event = 1;
event.sctp_partial_delivery_event = 1;
event.sctp_adaptation_layer_event = 1;

if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0) {
perror("setevent failed");
exit(1);
}

/* Set associations to auto-close in 20 seconds of
* inactivity
*/
if (setsockopt(fd, IPPROTO_SCTP, SCTP_AUTOCLOSE,&idleTime, 4) < 0) {
perror("setsockopt SCTP_AUTOCLOSE");
exit(1);
}

/* Allow new associations to be accepted */
if (listen(fd, 1) < 0) {
perror("listen");
```

```
exit(1);
}

//clear sctp_sndrcvinfo
memset (&sri, 0, sizeof(sri));

printf ("{one-to-many}: Waiting for associations ...\n");

// Wait for new associations
while(1)
{
//dynamic multi-homing after message
if (!strncmp(buf, "Send1", 5))
{
// bind_address(fd, atoi(argv[1]));
}

//Echo back any and all data
memset (buf, 0, sizeof(buf));
len = sizeof (struct sockaddr);


//reveive messages, and handle notification messages
sz = sctp_recvmsg (fd, buf, sizeof(buf),(struct sockaddr *) &cli_addr, &len, &sri, &m

if (debug)
printf ("sctp_recvmsg:[%d,e:%d,fl:%X]: ", sz, errno, msg_flags);
if (sz <= 0)
break;
//check for nofticiation
if (msg_flags & MSG_NOTIFICATION)
{
handle_event(fd, buf);
continue;
}
printf ("Send on str:%d sqn:%d [assoc:0x%x]:", sri.sinfo_stream,sri.sinfo_ssn,(u_int)
```

```
printf("%.*s\n",sz,buf);

if (!strncmp(buf, "Send3", 5))
{

//change_primary(fd, sri.sinfo_assoc_id, (int) atoi(argv[1]));
}



//echo back all messages
sz = sctp_sendmsg (fd, buf, sz,(struct sockaddr *) &cli_addr, len, sri.sinfo_ppid, sr
if (debug)
printf ("sctp_sendmsg:[%d,e:%d]\n", sz, errno);
}

close(fd); //close socket (comes here when there is an error)

}
```

one2oneclient.c

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

int debug = 1;
```

```
int sendmesg(int sock_fd, long int stream, char *msg) {
    int sz, msg_flags;
    struct sockaddr_in serv_addr;   /* warning: only IPv4 */
    size_t adrlen;
    char readbuf[256];
    struct sctp_sndrcvinfo sri;

    memset(&sri, 0, sizeof(sri));
    memset(readbuf, 0, sizeof(readbuf));
    printf("%d", sizeof(readbuf));
    sri.sinfo_stream = stream;
    sz = strlen(msg);
    sz = sctp_sendmsg (sock_fd, msg, sz, NULL, 0,
            0, 0, sri.sinfo_stream, 0, 0);
    //        sri.sinfo_ppid, sri.sinfo_flags, sri.sinfo_stream, 0, 0);
    if (debug) printf ("sctp_sendmsg:[l:%d, e:%d]\n", sz, errno);
    if (sz <= 0)
        return -1;

    adrlen = sizeof(serv_addr);
    msg_flags = 0;
    sz = sctp_recvmsg (sock_fd, readbuf, sizeof(readbuf),
            (struct sockaddr*) &serv_addr, &adrlen, &sri, &msg_flags);
    //        NULL, NULL, &sri, &msg_flags);
    if (debug) printf ("sctp_recvmsg:[l:%d, e:%d] ", sz, errno);
    if (sz <= 0)
        return -1;
    printf ("<-- %s      from %s on stream %d\n", readbuf, inet_ntoa(serv_addr.sin_ad
    // printf ("<-- %s on str %d\n", readbuf, sri.sinfo_stream);
    return 0;
}

int getcurport(int sock_fd) {
    // find the currently used port of the server, and return that port
```

```
    struct sockaddr* addressArray;
    int count;
    int port = -1;
    count = sctp_getpaddrs(sock_fd, 0, &addressArray);
    if(count > 0) {
        // Warning: IP specific. Use getaddrinfo()
        switch(addressArray->sa_family) {
            case AF_INET:
                port = htons(((struct sockaddr_in*)addressArray)->sin_port);
            break;
            case AF_INET6:
                port = htons(((struct sockaddr_in6*)addressArray)->sin6_port);
            break;
        }
        sctp_freepaddrs(addressArray); // free allocated memory
    }
    return port;
}

int addconn(int sock_fd, char *iptext) {
    struct sockaddr_in serv_addr;  // warning: IPv4 only. replace with sockaddr_stora
    int err, port;
    sctp_assoc_t id;

    // Warning: IPv4 only. replace with getaddrinfo
    port = getcurport(sock_fd);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = port; // htons(port);
    serv_addr.sin_addr.s_addr = inet_addr(iptext);

    // serv_addr.sin_addr.s_addr = ip
    err = sctp_connectx(sock_fd, (struct sockaddr*)&serv_addr, 1, &id);
    if (err == 0) {
        printf ("Added %s:%d as association #%d \n", inet_ntoa(serv_addr.sin_addr), p
    } else {
        printf ("failed to add association %s:%d [l:%d, e:%d]\n", inet_ntoa(serv_addr
```

```
    }
    return 0;
}

int addip(int sock_fd, char *iptext) {
    struct sockaddr_in serv_addr;  // warning: IPv4 only. replace with sockaddr_stora
    int err, port;

    // Warning: IPv4 only. replace with getaddrinfo
    port = getcurport(sock_fd);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = port;
    serv_addr.sin_addr.s_addr = inet_addr(iptext);

    // serv_addr.sin_addr.s_addr = ip
    err = sctp_bindx(sock_fd, (struct sockaddr*)&serv_addr, 1, SCTP_BINDX_ADD_ADDR);
    if (err == 0) {
        printf ("Added %s:%d as association\n", inet_ntoa(serv_addr.sin_addr), port);
    } else {
        printf ("failed to add association %s:%d [l:%d, e:%d]\n", inet_ntoa(serv_addr
    }
    return 0;
}

int delip(int sock_fd, char *iptext)
{
    int err,count;

    struct sockaddr* addressArray;

    count = sctp_getladdrs(sock_fd, 0, &addressArray);
    printAddressArray(addressArray, count);
    // serv_addr.sin_addr.s_addr = ip
    err = sctp_bindx(sock_fd, addressArray, 2, SCTP_BINDX_REM_ADDR);
sctp_freepaddrs(addressArray);
count = sctp_getladdrs(sock_fd, 0, &addressArray);
```

```
printAddressArray(addressArray, count);
    if (err == 0) {
        //printf ("Removed %s:%d as association\n", inet_ntoa(serv_addr.sin_addr), po
    } else {
        //printf ("failed to remove association %s:%d [l:%d, e:%d]\n", inet_ntoa(serv
    }
if(count > 0) {
        sctp_freepaddrs(addressArray); // free allocated memory
}
    return 0;
}

void printAddressArray(struct sockaddr* addressArray, int len) {
    int i, error;
    struct sockaddr *address;
    char host[100];
    char port[10];

    for(i = 0;i < len; i++) {
        printf("%d/%d: ",i + 1, len);
        memset(host, 0, sizeof(host));
        memset(port, 0, sizeof(port));
        address = &addressArray[i];
        error = getnameinfo(address, address->sa_len,
                            host, sizeof(host),
                            port, sizeof(port),
                            NI_NUMERICHOST);
        if (error != 0) {
            fprintf(stderr, "getnameinfo error\n");
            printf("(unknown address)\n");
        } else {
            printf("%s:%s\n", host, port);
        }
    }
}
```

```
int listip(int sock_fd) {
    struct sockaddr* addressArray;
    int count;

    count = sctp_getladdrs(sock_fd, 0, &addressArray);
    if(count > 0) {
        printf("SCTP association local addresses:\n");
        printAddressArray(addressArray, count);
        sctp_freepaddrs(addressArray); // free allocated memory
    }
    count = sctp_getpaddrs(sock_fd, 0, &addressArray);
    if(count > 0) {
        printf("SCTP association remote addresses:\n");
        printAddressArray(addressArray, count);
        sctp_freepaddrs(addressArray); // free allocated memory
    }
    return 0;
}


int main(int argc, char **argv)
{
    int sock_fd, r;
    // int idleTime = 2;
    // struct sockaddr_in sin[1];
    struct sockaddr_in serv_addr;  // warning: IPv4 only. replace with sockaddr_stora
    struct sctp_event_subscribe event;
    char buf[256];
    char *text;
    const char sep = ' ';
    long stream;

    if (argc < 3) {
        printf ("\nUsage: <%s> <remote-ip> <port> \n\n", argv[0]);
        return -1;
    }
```

```
// Warning: IPv4 only. replace with getaddrinfo
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons (atoi(argv[2]));
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);

sock_fd = socket (AF_INET, SOCK_STREAM, IPPROTO_SCTP);
if (sock_fd == -1) {
    printf ("\nUnable to create socket \n");
    return -1;
}

if (connect (sock_fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    printf ("\nUnable to connect to remote server [%s] !! \n\n",
                inet_ntoa(serv_addr.sin_addr));
    return -1;
} else {
    // warning: inet_ntoa is ipv4 only. replace with inet_ntop
    printf ("\nConnected to [%s]\n", inet_ntoa(serv_addr.sin_addr));
}

memset (&event, 0, sizeof(event));
if (setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0) {
    perror("setevent failed");
    exit(1);
}

memset (buf, 0, sizeof(buf));

while (1) {
    printf ("<S>: ");
    memset (buf, 0, sizeof(buf));
    if (!fgets(buf, 256, stdin))
        break;

    text = buf;
```

```
    /* Split buffer into first word (at *buf) and remaining text (at *text).
    Do this by replacing the first space with a NUL character, and setting *text
    while ((*text != sep) && (*text != '\0'))
        text++;
    if (*text == sep) {
        *text = '\0';
        text++;
    }

    if (isdigit(*buf)) {
        stream = strtol(buf, NULL, 0);
        r = sendmesg(sock_fd, stream, text);
    } else if (!strncmp(buf, "exit", 4)) {
        break;
    } else if (!strncmp(buf, "conn", 4)) {
        r = addip(sock_fd, text);
    } else if (!strncmp(buf, "add", 3)) {
        r = addip(sock_fd, text);
    } else if (!strncmp(buf, "del", 3)) {
        r = delip(sock_fd, text);
    } else if (!strncmp(buf, "list", 4)) {
        r = listip(sock_fd);
    } else {
        printf ("Error, line must be of the form 'command text'. Allowed commands
        printf ("    0, 1, 2, ... <text>: send message on specified stream.\n");
        printf ("    add, del <ip>: add or remove IP address from the association
        printf ("    conn <ip>: add IP address from the association (using connec
        printf ("    list: list all IP addresses of the current association.\n");
        printf ("    exit: Thanks for the fish.\n");
        continue;
        r = 0;
    }
    if (r < 0) {
        break;
    }
}
```

```
    shutdown(sock_fd, SHUT_WR);
    printf("\nOver !!\n");
    close(sock_fd);

    return 0;
}
```

one2oneserver.c

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>
#include <sys/wait.h>

int dofork = 0;
int debug  = 1;
int reportevents = 1;

static void
handle_event(void *buf)
{
    struct sctp_assoc_change *sac;
    struct sctp_send_failed *ssf;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    union sctp_notification *snp;
    char addrbuf[INET6_ADDRSTRLEN];
    const char *ap;
```

```
struct sockaddr_in *sin;
struct sockaddr_in6 *sin6;

snp = buf;

switch (snp->sn_header.sn_type) {
case 0:
    break;

case SCTP_ASSOC_CHANGE:
    sac = &snp->sn_assoc_change;
    printf("--- assoc_change: state=%hu, error=%hu, instr=%hu "
    "outstr=%hu\n", sac->sac_state, sac->sac_error,
    sac->sac_inbound_streams, sac->sac_outbound_streams);
    break;

case SCTP_SEND_FAILED:
    ssf = &snp->sn_send_failed;
    printf("--- sendfailed: len=%hu err=%d\n", ssf->ssf_length,
    ssf->ssf_error);
    break;

case SCTP_PEER_ADDR_CHANGE:
    spc = &snp->sn_paddr_change;
    if (spc->spc_aaddr.ss_family == AF_INET) {
    sin = (struct sockaddr_in *)&spc->spc_aaddr;
    ap = inet_ntop(AF_INET, &sin->sin_addr,
            addrbuf, INET6_ADDRSTRLEN);
    } else {
    sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
    ap = inet_ntop(AF_INET6, &sin6->sin6_addr,
            addrbuf, INET6_ADDRSTRLEN);
    }
    printf("--- intf_change: %s state=%d, error=%d\n", ap,
      spc->spc_state, spc->spc_error);
    break;
```

```
    case SCTP_REMOTE_ERROR:
        sre = &snp->sn_remote_error;
        printf("--- remote_error: err=%hu len=%hu\n",
        ntohs(sre->sre_error), ntohs(sre->sre_length));
        break;

    case SCTP_SHUTDOWN_EVENT:
        printf("--- shutdown event\n");
        break;

    case SCTP_ADAPTATION_INDICATION:
        printf("--- Adaptation event\n");
        break;

    case SCTP_PARTIAL_DELIVERY_EVENT:
        printf("--- partial delivery event\n");
        break;

    case SCTP_STREAM_RESET_EVENT:
        printf("--- stream resest event\n");
        break;

    default:
        printf("unknown type: %hu\n", snp->sn_header.sn_type);
        break;
    }
}

/* Handle incoming connections */
int handleconnection(int sd)
{
    int sz, msg_flags;
    size_t adrlen;
    struct sockaddr_storage msg_addr;
    char readbuf[100];
```

```
    char host[100];
    struct sctp_sndrcvinfo sri;
    struct sctp_event_subscribe event;

    memset (&sri, 0, sizeof(sri));

    if (reportevents) {
        /* Enable all events */
        event.sctp_data_io_event = 1;
        event.sctp_association_event = 1;
        event.sctp_address_event = 1;
        event.sctp_send_failure_event = 1;
        event.sctp_peer_error_event = 1;
        event.sctp_shutdown_event = 1;
        event.sctp_partial_delivery_event = 1;
        event.sctp_adaptation_layer_event = 1;
    } else {
        /* Not interested in any events for now */
        memset (&event, 0, sizeof(event));
    }
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_EVENTS, &event, sizeof(event)) != 0) {
        perror("setevent failed");
        exit(1);
    }

    while (1) {
        /* Echo back any and all data */
        memset (readbuf, 0, sizeof(readbuf));

        adrlen = sizeof(msg_addr);
        sz = sctp_recvmsg (sd, readbuf, sizeof(readbuf),
                    (struct sockaddr*) &msg_addr, &adrlen, &sri, &msg_flags);
        if (debug) printf ("sctp_recvmsg:[l:%d, e:%d, fl:%X]: ", sz, errno, msg_flags
        if ((reportevents) && (msg_flags & MSG_NOTIFICATION)) {
            handle_event(readbuf);
            if (sz <= 0)
```

```
                return 0;
            continue;
        }
        if (sz <= 0)
            return 0;

        // Warning: inet_ntop may return NULL pointer
        printf ("<-- %s      from %s on stream %d\n", readbuf, inet_ntop(msg_addr.ss_
        sz = sctp_sendmsg (sd, readbuf, sz, NULL, 0,
                0, 0, sri.sinfo_stream, 0, 0);
        //        sri.sinfo_ppid, sri.sinfo_flags, sri.sinfo_stream, 0, 0);
        if (debug)
            printf ("sctp_sendmsg:[l:%d, e:%d]\n", sz, errno);
    }
    shutdown(sd, SHUT_RDWR);
    close(sd);
}

/* main function. Listen on the given port, and wait for incoming connection. */
int main(int argc, char **argv)
{
    int fd, client_sd;
    size_t adrlen;
    // int idleTime = 20;
    struct sockaddr_in sin[1];  // warning: IPv4 only
    struct sockaddr_in cli_addr; // warning: IPv4 only. replace with sockaddr_storage
    fd_set fdset;
    pid_t pid;
    int rv;

    if (argc < 2) {
        printf ("\nUsage: <%s> <port> \n\n", argv[0]);
        return -1;
    }

    // warning: this is IPv4 only.
```

```
if ((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
    perror("socket");
    exit(1);
}

// warning: IPv4 only
sin->sin_family = AF_INET;
sin->sin_port = htons(atoi(argv[1]));
sin->sin_addr.s_addr = INADDR_ANY;
if (bind(fd, (struct sockaddr *)sin, sizeof (*sin)) == -1) {
    perror("bind");
    exit(1);
}

/* Allow new associations to be accepted */
if (listen(fd, 1) < 0) {
    perror("listen");
    exit(1);
}

printf ("{one-to-one}: Waiting for associations ...\n");

FD_ZERO(&fdset);
FD_SET(fd, &fdset);

/* Wait for new associations */
while(1) {
    if (select (fd+1, &fdset, 0, 0, 0) <= 0)
        continue;

    adrlen = sizeof(cli_addr);
    client_sd = accept(fd, (struct sockaddr *) &cli_addr, &adrlen);
    if (client_sd >= 0 ) {
        // inet_ntoa is IPv4 only. replace with inet_ntop
        printf ("\n Connection from: %s\n", inet_ntoa(cli_addr.sin_addr));
        /* handle incoming connections */
```

```
            if (dofork) {
                pid = fork();
                if (pid < 0) {
                    perror("fork");
                    close(client_sd);
                    close(fd);
                    exit(1);
                } else if (pid > 0) { /* parent */
                    close(client_sd);
                    // wait for the child to exit, otherwise we fill up the process t
                    // and awake the sysadmin from hell
                    wait(&rv);
                    continue;
                } else { /* child */
                    handleconnection(client_sd);
                }
            } else { /* do not fork, simply handle connection */
                handleconnection(client_sd);
            }
        } else { /* client_sd < 0: error */
            if (errno == EINTR)           /* EINTR might happen on accept(), */
                continue;                   /* try again */
            perror("accept");             /* bad */
            exit(1);
        }
    }

    /* unreachable */
    close(fd);
    return 0;
}
```

# List of Figures

2.1 SIGTRAN functional mode [1] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.1 SCTP vs TCP vs UPD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
3.2 SCTP packet structure [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.3 Chunk types[3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.4 4-way handshake[5] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.5 Multistreaming association (Head-of-Line Blocking)[1] . . . . . . . . . . . . . 10
3.6 Multihoming association[13] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

5.1 Scenarion 1: Short roundtrip . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
5.2 Scenarion 2: Long roundtrip . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

# List of Tables

7.1 Performance of TCP vs SCTP 1 (as measured by netperf) . . . . . . . . . . 24
7.2 Performance of TCP vs SCTP 2 (as measured by netperf) . . . . . . . . . . 24
7.3 Multiple stream performance (random packetloss) . . . . . . . . . . . . . . . 26
7.4 Multiple stream performance (bandwidth) . . . . . . . . . . . . . . . . . . . 27
7.5 short vs long roundtrip . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
7.6 wireshark timeline of multihoming test . . . . . . . . . . . . . . . . . . . . . 30
7.7 wireshark output of multihoming test . . . . . . . . . . . . . . . . . . . . . . 32