Department of Informatics
University of Oslo

Improvements of the Linux SCTP API

Master thesis

Geir Ola Vaagland

May 27, 2014

Improvements of the Linux SCTP API

Geir Ola Vaagland

May 27, 2014

Acknowledgments

I wish to acknowledge the help provided by my supervisors Ernst G. Gran and Thomas Dreibholz for giving great advice, and patient guidance during my work with this thesis.

I would also like to thank Michael Tüxen and Vladislav Yasevich for providing me with valuable and constructive suggestions during the implementation phase.

Finally, I would like to thank my good friend and co-student Øystein for helping me gather my thoughts when things were difficult.

Abstract

This master thesis outlines the changes that need to be made to get the current Linux implementation of the Stream Control Transmission Protocol (SCTP) up to date with recently released "Sockets Application Programming Interface Extensions for the Stream Control Transmission Protocol" (RFC 6458). The thesis contains a thorough review of the discovered changes, and describes the work done in this thesis with regards to implementing some of the new features. SCTP is a transport layer communication protocol that serves a similar role to popular protocols like the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), but SCTP has unique features like multistreaming, multihoming and better validation and acknowledgement mechanisms that further enforce security.

Contents

List of Figures

List of Tables

Code listings

Chapter 1

Introduction

Since the early days of computer networking, even before the Internet as we know it today was introduced, there has been constant research on finding a good way of letting computers communicate fast, reliably and in a "universal language", known as a protocol. Newcomers to the network, like mobile devices and all other new types of electronical gadgets should be able to learn these protocols, and the protocols should be able to cater for their needs. Even though the Internet is becoming more and more grown up these days, some people are always trying to come up with something better, faster and more clever than what is currently available. One such new protocol is the Stream Control Transmission Protocol (SCTP) [1]. The SCTP implementation for the Linux operating system [2] is lagging behind compared to the general progress of the standardization work performed by the Internet Engineering Task Force (IETF). To help Linux developers who want to use SCTP as their communication protocol, it is necessary to understand what measures that need to be taken to bring SCTP for Linux up to the same level of quality and support as with the other operating systems. The Request For Comments (RFC) no. 6458 [3], defines the mappings between SCTP and a socket application programming interface (API). This includes compatibility with existing APIs for the Transmission Control Protocol (TCP) [4] and access to new SCTP features such as an error/event notification scheme and a new control data scheme.

1.1 The Internet Protocol Suite

The Internet protocol suite, commonly known as the TCP/IP model, is maintained by the IETF. This model was named after the first two protocols to be defined in this stan-

Figure 1.1: The Internet Protocol Suite

Layers	**Common protocols**
Application	SMTP, HTTP, etc.
↑ Sockets API interface ↓	
Transport	TCP, UDP, **SCTP**
Network	IPv4, IPv6
Link	Ethernet, Serial, etc.

dard, which also is the two most widely used and known: the Transmission Control Protocol (TCP) and the Internet Protocol (IP) [5]. The idea of having a layered networking model was conceived in the late 1960s by the Defence Advanced Research Projects Agency (DARPA), an agency funded by the United States Department of Defence.

The model specifies how data should be formatted, addressed, transmitted, routed and received at the destination. It is split into four layers (Figure 1.1), where the basic idea is to allow one application running on one host to talk to another application on a different host. Put simply, a packet from the application layer travels down the layers and each layer places control information on the packet and passes it on. At the receiving end the opposite happens. The packet is stripped of its control information and its content received by the application layer of the receiving part.

The **link layer** is responsible for communication with the physical interfaces, the **network layer** handles movement of packages through the network, specifically routing, and the **transport layer**, which is where SCTP resides, regulates the flow of packets between endpoints[1]. It also presents the application its endpoint for communication, known as ports. The application layer makes use of the underlying architecture by sending the actual data and giving it meaning through the use of the operating system's network system calls. These system calls makes communication with the lower layers a lot less complex. For Linux, and most other operating systems, this implementation of system calls is known as the sockets API.

1.1.1 The sockets API

A socket API is provided by an operating system, and allows applications to use the concept of network sockets. A socket is an endpoint of a two-way communication link between two entities. Like the opening of a tube that data gets pushed or pulled out of. A socket is bound to a port number, so that the transport layer knows which application the data is destined to or from.

The BSD sockets (or Berkeley sockets[2]) are very similar to the Portable Operating System Interface (POSIX) sockets that are being used in Linux. The Socket API has functions to create a socket and to receive and write data to it. It also contains functions to bind a socket to a specific port (*bind()*), getting and setting socket options with *get-/setsockopt()* and means to accept connection requests and connect sockets to other sockets (*accept()/connect()*). These functions and their related data structures make up a complete socket API.

1.2 What is SCTP?

SCTP is a reliable, general-purpose transport layer protocol intended for use on IP networks, serving a similar role as popular protocols such as the User Datagram Protocol (UDP) [6] and TCP, lending ideas from both. SCTP is message-oriented, similar to UDP,

[1]Think of an endpoint as one of the parties communicating with SCTP. E.g. a network interface or even better, a single IP address.

[2]The Socket API originated with the 4.2BSD Unix operating system released in 1983.

and further ensures reliability and in-sequence message transport with congestion control, similar to TCP. For completeness a short section that outlines the most important similarities and differences between TCP and SCTP will be provided in Section 1.4.

Although TCP had provided excellent services as the primary means of reliable data transfer in IP networks over the past decades, an increasing number of applications found TCP too limiting in certain areas. Thus people started making their own reliable data transfer protocols based on UDP. It was decided that a new protocol was needed, and it had to satisfy the following requirements:

- **Reliable message delivery**
  The recipient of a message[3] acknowledges that it has received it, or the sender should make sure to retransmit if something went wrong.

- **Network-failure tolerance**
  A failure in the network should be detected and handled in a reasonable way. E.g. wait to retransmit or figure out an alternative path to the destination.

- **Avoid the head-of-line [7] problem**
  Avoid packets queuing up while waiting for a blocked packet to get out of the way. See Subsection 1.4.2.

- **Better security**
  Security is always an important issue when designing protocols. It is never desirable to not know whether someone else can intercept your e-mails or tamper with your bank transactions. Although several mechanisms to improve security have been developed, TCP is fundamentally more vulnerable to denial of service attacks [8] than SCTP.

1.2.1 The new SCTP API - RFC 6458

The main design goals of the SCTP API are three-fold, and are summarized as follows in RFC 6458:

- Maintain consistency with the existing sockets API.
- Support a one-to-many UDP-style interface.
- Support a one-to-one TCP style interface.

As the two latter goals are not completely compatible, RFC 6458 defines two different modes of operation. Although they share some data structures and operations, they require different programming styles. The decision of which style to use depends on the indent of the application. This API for Linux has never been completely implemented.

[3]TCP is also a reliable protocol, but handles streams of bytes rather than messages. This will be covered in Section 1.4 on page 5.

1.3 Background

The SCTP implementation for Linux is developed by the *Linux Kernel SCTP project* (LKSCTP) [9]. LKSCTP provides both a userspace library and a kernel part. But since the development of the "final" RFC 6458 has taken so long, more than 10 years and 33 draft versions, it seems fair to assume that the developers have been a bit discouraged about keeping their APIs' up to date. On the other hand, the FreeBSD [10] SCTP implementation has been maintained by the authors of the RFC, and is thus far more up to date. However, the largest deployment of SCTP is by various Linux[4] distributions, so having SCTP up to date for this platform would surely be very useful.

1.3.1 History of SCTP

SCTP was defined by the IETF Signaling Transport (SIGTRAN) [11] working group in 2000, and is currently maintained by the IETF Transport Area (TSVWG) working group [12]. The SIGTRAN group is concerned with the transport of telephony signalling data over IP. They concluded that none of the existing transport protocols satisfied the transport requirements of signalling data, and decided that they required a transport protocol that met the needs mentioned in Section 1.2.

To solve this, SIGTRAN selected a proposed standard from Randall R. Stewart and Qiaobing Xie, two Motorola employees, as a starting point. Stewart and Xie had been developing a Distributed Processing Environment called Quantix [13], which was aimed at telephony applications. This environment had been successfully demonstrated at Geneva Telecom in 1999. Quantix brought support for multihoming, multistreaming and message framing. These concepts will be explained in Section 1.4. Eventually, the Internet Engineering Steering Group (IESG) [5], decided that the protocol was robust enough to be elevated from a specialised transport for telephony signalling to a more general purpose transport protocol like TCP and UDP.

[4]The Linux kernel has had built-in support for SCTP since version 2.6.x

[5]The IESG (Internet Engineering Steering Group) are the ones who make final reviews of proposed IETF standards

1.4 Short comparison of TCP and SCTP

When forming SCTP, the working group took care to incorporate lessons learnt from TCP, such as:

- **Selective ACKs**
  Selective ACKs well tuned retransmission scheme. Basically it involves the receiver being able to say "I have not received packet 3, but I have gotten packet 4,5 and 6" instead of just saying "I still only have gotten packet 2.", thus forcing the sender to retransmit all packets that have been sent from that point.

- **Message fragmentation and bundling**
  Fragmentation is the technique of splitting up messages larger than the link's Maximum Transmission Unit (MTU) into smaller fragments, and bundling is pretty much the opposite. Smaller messages get bundled together to keep the "message header to payload"-ratio at a reasonable level.

- **Congestion control**
  Congestion control is accomplished using the same model as for TCP, but SCTP has some specifics with regards to its multihoming traits. E.g. slow-starts[6] for each possible destination address. Congestion control is meant to detect and avoid bottlenecks in the network flow.

In addition to these similarities, TCP and SCTP are both **connection oriented**[7]. While TCP has its connections, SCTP operates with associations. A listen()-connect()-accept() cycle is needed to transfer data, although it works a bit differently for SCTP than for TCP.

Although SCTP has inherited these features from TCP, there are some notable differences that sets them apart. Subsection 1.4.1-1.4.4 presents a quick look at the key differences.

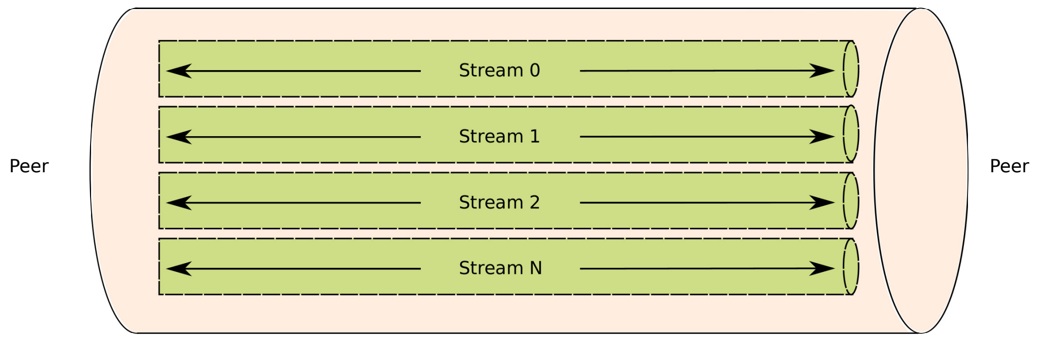
1.4.1 Multihoming

An essential property of SCTP is its ability to bind to several addresses on a single node. If allowed to use several addresses, SCTP can e.g. use the extra path simply for redundancy. Making use of the extra network path as a destination for failed messages, can make SCTP more resistant to network failure. All SCTP endpoints monitor the state of its path in an association by sending a heartbeat at some configurable interval. As one side sends a HEARTBEAT chunk (More on chunks will be covered in Subsection 2.6.2), the other responds with a HEARTBEAT_ACK, thus allowing the sender to detect a possible path failure, and proceed sending over another path.

[6]Slow start basically means sending a small chunk of data initially, and then slowly increasing the amount of transferred data until the other end chokes, and then settling on some value less than the choke point.

[7]SCTP's one-to-many/UDP style is also connection oriented, since the association is setup implicitly/on the fly.

Figure 1.2: Multistreaming: An SCTP association with multiple streams.



In Linux there are currently three ways to handle multihoming:

- Ignore it, only use one address.
- Bind all addresses.
- Using the *sctp_bindx()*-function to bind a specific subset of addresses.

The latter option is the most flexible, as it allows binding additional addresses to a socket after it has been bound with *bind()*.

1.4.2 Multistreaming

Another interesting feature of SCTP is that its associations support multiple streams. In simple terms, an SCTP packet header gets annotated with a stream number that identifies which stream the packet belongs to. These streams can e.g. be used for separating control packets from data packets, so that a stuck data packet will not delay more important control packets. Traditionally, with the single stream TCP approach, a packet awaiting retransmission because of a packet loss would imply that all other packets that were scheduled to move over the same channel would have to wait. This is called **head-of-line** blocking, which is what SCTP multistreaming is designed to solve.

Also contrary to having multiple TCP connections open, the SCTP multistream will not require the use of multiple ports. This is an important point, as the establishment of brand new connection would typically require a much larger amount of system resources compared to the multistreaming approach.

Multistreaming enables an association to have subflows inside the overall SCTP message flow and choose whether or not to enforce message ordering by the use of **unordered message delivery** [14].

1.4.3 Unordered message delivery

SCTP can be configured on a per stream basis, or for a single message within a stream, to send messages reliably, but unordered. An unordered message is "unordered" with respect to any other message, both ordered and unordered. It might be delivered before or after an ordered message sent on the same stream. This is useful for a message oriented protocol when it is dealing with independent transactions where ordering is not important. Essentially, without unordered message delivery enabled, an endpoint delivers user data messages to the upper layer according to the message's **Stream Sequence Number**. And if a message arrives out of order, it will be held back by the SCTP stack until enough messages have arrived. Another brief look at the more technical aspects of unordered messages will be given in Subsection 2.6.2.

1.4.4 Message framing

SCTP data transportation is message-oriented, while TCP is byte-oriented. This means that TCP guarantees that every chain of bytes that is sent will get to the recipient in the correct order, but with no conservation of any message boundaries. An application using TCP must thus often include length information within the message to tell the receiver how much to read. The receiver of a TCP message needs a reassembly buffer, since every time data is received either more or less data than is expected may show up. SCTP will not only deliver the messages in correct order, it will also indicate to the receiver both the beginning and the end of the received data. This relieves the application developer of the complex task of doing the buffering and framing of the messages manually. The SCTP approach also strips away the overhead of including length information in each data transmission.

1.5 Problem Definition and limitations

This section will briefly outline the main goals, and also the scope and limitations of this thesis. The goal is to present a thorough investigation of the latest SCTP API described in RFC 6458 up against the current SCTP implementation in Linux. Identifying out-of-date functions and related features that have been neglected by the Linux SCTP community the past few years. As FreeBSD already is up to date, it will be used as inspiration. So, although an attempt will be made at implementing some of the new concepts from RFC 6458, the main goal will be to identify all the changed functionality. A secondary goal is to get some of the features that are implemented during the work this thesis submitted to the Linux kernel developers and possibly also included in future Linux deployments.

Limitations

Due to time constraints and complexity, this thesis will not necessarily serve as a good step-by-step recipe covering every aspect of how to update the API. The task is not trivial, and requires a certain set of skills and good knowledge of the SCTP stack as it is implemented today. This document will however hopefully serve as a good starting point for anyone, with time and knowledge, who wants to commit to the task of developing the remaining features that are not present today.

Changing things that already works, i.e. taking away features that have been deprecated in RFC 6458, will not be the intent in this thesis. The current goal of the implementations is rather to have the new functionality reuse the deprecated features.

1.6 Research Method

A lot of the work done in this thesis has revolved around getting acquainted with the SCTP implementation, both at the user space level and at the kernel level. There has been a lot of browsing through source code and searching for needles that are not necessarily present. Some missing functionality was found by the use of *grep* [15] to search both the userspace library and the kernel for what was present and what was not. Often a more thorough search was required, as some functionality could e.g. be "hidden" through the use of different names than the ones used in RFC 6458.

Eventually, a list was compiled containing all found changes. It was then necessary to write test applications to verify that the userspace library failed to handle the changes from RFC 6458. Finally, some of the new functionality was implemented and added to the existing source code of the Linux kernel. As part of the work with this thesis, the implementations were also submitted to the Linux Kernel SCTP mailing list for further review, and possible inclusion in the mainline Linux kernel one day in the future. This process will be described in Chapter 5.

1.7 Main Contributions

The main contributions implemented as part of the work with this thesis are:

- The functions *sctp_recvv()* and *sctp_sendv()* should work as intended now.
- Implemented the socket options SCTP_RECVV_NXTINFO and SCTP_RECVV_-RCVINFO to enable retrieval of these types of ancillary data.
- Implemented the socket option SCTP_DEFAULT_SNDINFO.
- The structure type *sctp_recvv_rn* is used to enable receival of both *struct sctp_-rcvinfo* and *struct sctp_nxtinfo* in one call to *sctp_recvv()*.
- Renamed the state types SCTP_STATE_* to just SCTP_* according to RFC 6458.
- Identified missing functionality.
- Set the SCTP_COMPLETE flag for complete messages.
- Implemented a simple draft solution to the SCTP_SENDALL flag, to send messages to all associations established on a socket with *sctp_sendv()*.

1.8 Outline

This thesis is outlined like this: First, chapter 2 will give an overview of the state of the current Linux implementation, how things are done today, regardless of whether RFC 6458 will change it or not. A brief, but slightly technical introduction to some of the key concepts of SCTP will also be given here, like the concepts of chunks, associations, notifications and ancillary data. Chapter 3 will describe the shortcomings and missing pieces that were found in the current implementation of Linux. It will present the new functions introduced in RFC 6458, changes to how ancillary data is to be handled and which new socket options and notifications that have been introduced. Chapter 4 contains a presentation of what was implemented during the work with this thesis. This includes all the changes mentioned in the previous section. Most code listings will be found in this chapter. Chapter 5 presents a retrospective view with thoughts about the implementation process, a description of the submittal process of the code to the LKSCTP-developers, and a critical view on how the new features were implemented. Finally, chapter 6 will conclude the thesis by listing the key components of project, and propose a path for future work on SCTP.

1.9 Summary

This chapter has introduced the goals and methods that will be used in this thesis. It has presented the SCTP protocol, and outlined the purpose and scope of the thesis. The next chapter will look a bit closer at the current state of the SCTP implementation on the target platform, Linux.

Chapter 2

Overview of the current Linux SCTP architecture

As mentioned in Section 1.3, the Linux kernel has had support for SCTP since version 2.6[1]. In this chapter we will take a look at how some things have been implemented in Linux. This includes how multihoming works, how ancillary (control) data is handled and a closer look at the data types that are used to tie it all together.

2.1 Ancillary data

Ancillary data is metadata used to give information about the state of the SCTP subsystem. For instance, a developer can by using the functions *sendmsg()* and *recvmsg()*, make use of the ancillary data structures to decide which association or stream number a message should be sent to, or which stream it belongs to. Although ancillary data is crucial for multistreaming, it can be useful for other things as well. For one-to-many style communication, the ancillary data can be used to manage the associations as pleases. Ancillary data can also be used for requesting that a message should be delivered unordered, as was described in Subsection 1.4.3.

2.1.1 The msghdr structure

The msghdr structure is used to pass messages around with *sendmsg()* and *recvmsg()*. As is shown in Listing 2.1, the *msg_control* field is of type void, and can thus point to structures of any kind. Subsection 2.1.2 will show how this can come in handy when sending SCTP specific control structures.

Table 2.1 shows the only types of ancillary data that is being used by Linux in its current state. The new types that have been introduced in RFC 6458 are shown in Table 3.1.

The last field, the *msg_flags*, is set to MSG_NOTIFICATION (defined in sctp.h) when the message contains a notification. More details about notifications will be covered in Section 2.2.

[1]Version 2.6 of the Linux kernel was released in December 2003

Table 2.1: Types of ancillary data

Description	**cmsg_type**	**cmsg_data[]**
SCTP Initiation Structure	SCTP_INIT	struct sctp_initmsg
Header Information Structure	SCTP_SNDRCV	struct sctp_sndrcvinfo

```c
struct msghdr {
    void *msg_name;         /* optional address */
    socklen_t msg_namelen; /* size of address */
    struct iovec *msg_iov; /* scatter/gather array */
    size_t msg_iovlen;      /* # elements in msg_iov */
    void *msg_control;      /* ancillary data, see below */
    size_t msg_controllen; /* ancillary data buffer len */
    int msg_flags;          /* flags on received message */
};
```

Listing 2.1: struct msghdr

2.1.2 The cmsghdr structure

The cmsghdr structure [16] is used to specify SCTP options for *sendmsg*(), and to describe SCTP header information when receiving a message. Note that both this structure and *struct msghdr* is defined in the header file *include/linux/socket.h* in the Linux kernel source code. It is thus not a structure specific to SCTP, but is used to pass control data regardless of the protocol used.

```c
struct cmsghdr {
    socklen_t cmsg_len; /* data byte count, including
                           header */
    int cmsg_level;     /* originating protocol */
    int cmsg_type;      /* protocol-specific type */

    /* followed by unsigned char cmsg_data[]; */
};
```

Listing 2.2: struct cmsghdr

As mentioned, the *msg_control* field of the *struct msghdr* in Listing 2.1 can point to anything. Most commonly it will point to a *struct cmsghdr*, shown in Listing 2.2, which in turn contiains one of the SCTP specific control structures. So, for SCTP the *cmsg_data* field contains one of the SCTP-specific structures in Table 2.1, and the *msg_control* field of the *struct msghdr* is set to point to this *struct cmsghdr*.

2.1.3 struct sctp_sndrcvinfo

A single structure, *struct sctp_sndrcvinfo* (described in section 5.3.2. [3]) is used for both sending options to the SCTP stack (with *sendmsg()*) , and receiving configuration parameters and control information (with *recvmsg()*). This structure was split with RFC 6458, and two new structures have been introduced instead: *struct sctp_rcvinfo* and *struct sctp_sndinfo*. These will be covered in Chapter 3. The options from *struct sctp_sndrcvinfo* that were only relevant to sending was placed in *struct sctp_sndinfo* and vice versa. Listing 2.3 shows this structure as it is today.

```
struct sctp_sndrcvinfo {
    uint16_t sinfo_stream;
    uint16_t sinfo_ssn;
    uint16_t sinfo_flags;
    uint32_t sinfo_ppid;
    uint32_t sinfo_context;
    uint32_t sinfo_timetolive;
    uint32_t sinfo_tsn;
    uint32_t sinfo_cumtsn;
    sctp_assoc_t sinfo_assoc_id;
};
```

Listing 2.3: struct sctp_sndrcvinfo

An example on how to send ancillary data with *sendmsg()* and how to receive it with *recvmsg()* has been included in Section A.1 and Section A.2 of the Appendix.

2.2 Notifications

When SCTP applications receive messages, the SCTP stack can "piggyback" notifications related to non-data events. When a notification arrives, *sctp_recvmsg()* (defined in *net/sctp/socket.c*) sets the MSG_NOTIFICATION flag, and sends the message to the application layer as it normally would. The available notifications are listed in Table 2.2. Each one of these notification types has a structure that holds data related to the notification. A notification is represented by the union shown in Listing 2.4. Note that a union means that it will only contain *one* of its members.

Table 2.2: Notification types

Notification type	Description
SCTP_ASSOC_CHANGE	An SCTP association has started/ended.
SCTP_PEER_ADDR_CHANGE	A multihomed peer has changed state.
SCTP_REMOTE_ERROR	Can indicicate various error conditions. [14]
SCTP_SEND_FAILED	SCTP cannot deliver message. Deprecated.
SCTP_SHUTDOWN_EVENT	One of the peers have shut down.
SCTP_ADAPTATION_INDICATION	Adaptation Layer Indication received. [17]
SCTP_PARTIAL_DELIVERY_EVENT	Various events related to partial delivery.
SCTP_AUTHENTICATION_EVENT	Can report various events related to authentication. [18]
SCTP_SENDER_DRY_EVENT	No more user data to send or retransmit.
SCTP_NOTIFICATIONS_STOPPED_EVENT	Indicates that the stack is out of buffer space, and stops further notifications.

```c
union sctp_notification {
    struct sctp_tlv {
        uint16_t sn_type; /* Notification type. As shown in
                             Table~2.2. */
        uint16_t sn_flags;
        uint32_t sn_length;
    } sn_header;
    struct sctp_assoc_change sn_assoc_change;
    struct sctp_paddr_change sn_paddr_change;
    struct sctp_remote_error sn_remote_error;
    struct sctp_send_failed sn_send_failed;
    struct sctp_shutdown_event sn_shutdown_event;
    struct sctp_adaptation_event sn_adaptation_event;
    struct sctp_pdapi_event sn_pdapi_event;
    struct sctp_authkey_event sn_auth_event;
    struct sctp_sender_dry_event sn_sender_dry_event;
    struct sctp_send_failed_event sn_send_failed_event;
};
```

Listing 2.4: union sctp_notification

```c
struct sctp_event_subscribe {
    uint8_t sctp_data_io_event;
    uint8_t sctp_association_event;
    uint8_t sctp_address_event;
    uint8_t sctp_send_failure_event;
    uint8_t sctp_peer_error_event;
    uint8_t sctp_shutdown_event;
    uint8_t sctp_partial_delivery_event;
    uint8_t sctp_adaptation_layer_event;
    uint8_t sctp_authentication_event;
    uint8_t sctp_sender_dry_event;
};
```

Listing 2.5: struct sctp_event_subscribe

```c
     struct sctp_event_subscribe events;

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

     events.sctp_data_io_event = 1;
     events.sctp_association_event = 1;

     setsockopt(sd, IPPROTO_SCTP, SCTP_EVENTS, &events,
        sizeof(events));
```

Listing 2.6: Example: Enabling notifications with SCTP_EVENTS.

2.2.1 Notification Interest Options

No notifications are enabled by default, so in order to be able to receive notifications from the SCTP stack, an application must set the appropriate socket option. Notifications can be enabled by setting the SCTP_EVENTS socket option. By passing along a *struct sctp_event_subscribe* the developer can choose exactly which events from the SCTP stack that are desired. The events could e.g. be a new association coming up or an address transport failure. Notifications are distinguished from other data since they have a MSG_NOTIFICATION flag set in the *msg_flags* field of the *struct msghdr*.

One or more of these notifications can be enabled with a single *setsockopt()*-call. By setting the fields to 1 the SCTP stack will inform the upper layer whenever one of those respective events happen. The *data_io_event* field is of particular interest for this thesis, as it is how SCTP prior to RFC 6458 knew when to include *struct sctp_sndrcvinfo* ancillary data.

An example borrowed from section 6.2.1. of RFC 6458 [3] is provided in Listing 2.6

Table 2.4: The states in the SCTP state machine and their meanings

State	Meaning
CLOSED	No connection
COOKIE-WAIT	Waiting for a cookie
COOKIE-ECHOED	Waiting for a cookie acknowledgment
ESTABLISHED	Connection is established; data are being transferred
SHUTDOWN-PENDING	Sending data after receiving *close*
SHUTDOWN-SENT	Waiting for SHUTDOWN acknowledgment
SHUTDOWN-RECEIVED	Sending data after receiving SHUTDOWN
SHUTDOWN-ACK-SENT	Waiting for termination completion

2.3 The state machine

The kernel part of SCTP maintains a finite state machine that converts header information and produces a set of side-effects that it then processes, and creates actions. The state machine is queried to determine which actions to perform during association establishment, association termination and data transfer. There are four types of events (as defined in *net/sctp/constants.h* in the Linux kernel) that can cause a transition in the state machine:

- SCTP_EVENT_T_CHUNK
- SCTP_EVENT_T_TIMEOUT
- SCTP_EVENT_T_OTHER
- SCTP_EVENT_T_PRIMITIVE

These events are processed by the function *sctp_do_sm()* in sm_sideeffect.c. It creates a *struct sctp_sm_retval* that contains a description of the side effects of the given event. The side effect processor function *sctp_side_effects()* then converts the *struct sctp_sm_retval* into actions.

More information about the state machine can be found in the LKSCTP paper [13].

2.4 The smart pipes

The smart pipe is described as an oven. Raw input is injected in one end, and it serves "cooked" output in the other end. There are four types of smart pipes being used in LKSCTP, which will all be explained here.

SCTP_inqueue and SCTP_ULPqueue

These two smart pipes are used to carry data from "the wire" to the user. *SCTP_inqueue* turns packets into chunks (See Subsection 2.6.2), reassembles fragmented messages, tracks received Transmission Sequence Numbers (TSN) for acknowledgments, and manages the receiving window-size for congestion control. Each endpoint has an inqueue for handling unassociated messages, and another for each association.

The *SCTP_ULPqueue* (Upper-layer-protocol) accepts events (data messages or notifications) from the state machine and delivers these to the upper layer through the sockets layer. It is responsible for delivering streams of messages in order.

SCTP_outqueue

The *SCTP_outqueue* is responsible for the bundling logic, transport selection, outbound congestion control, fragmentation, and any necessary data queueing. Every outbound chunk goes through a queue like this, although the state machine is able to put a chunk directly on the wire. Currently, only ABORT uses this feature.

The SCTP_packet queue

The *SCTP_packet* queue is called a "lazy packet transmitter". It blindly bundles chunks and transmits. It does not accept packets that need fragmenting, nor does it handle any congestion logic. An example packet created by this queue is shown in figure 2.3.

2.5 SCTP Associations

As SCTP is a connection-oriented transport protocol, the two SCTP endpoints must create an association before exchanging data. One specific pair of endpoints can never have more than one association between them. However, the endpoints can always have associations to other endpoints simultaneously. Figure 2.1 shows how this relation works. Machine A has two network interfaces, and each has its own, unique association to machine B's single network interface. The association provides a feature of particular importance, the multiple stream feature. This enables an endpoint to transfer multiple separate sequences of reliable messages simultaneously and independently. An association can be setup to have up to 65536 different streams which can be created and used for simultaneous data transfer. In order to indicate over which specific stream a message is to be sent, the developer must "tag" the message with a stream identifier through the use of the *snd_sid* / *sinfo_stream* field mentioned in Section 2.1.

Figure 2.1: Example of an SCTP association

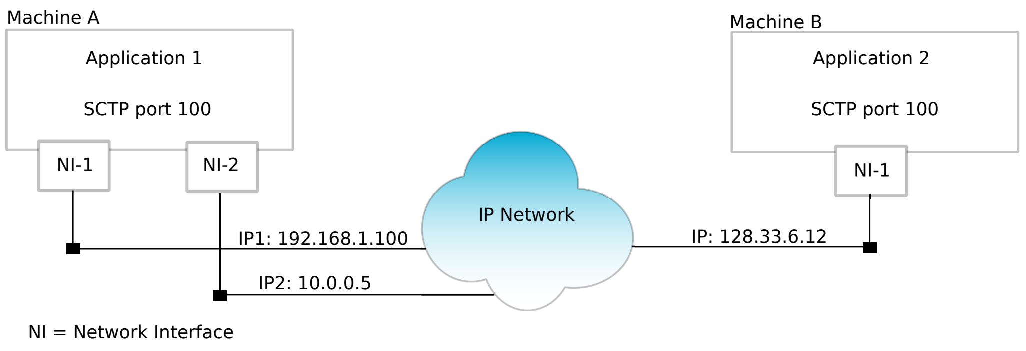
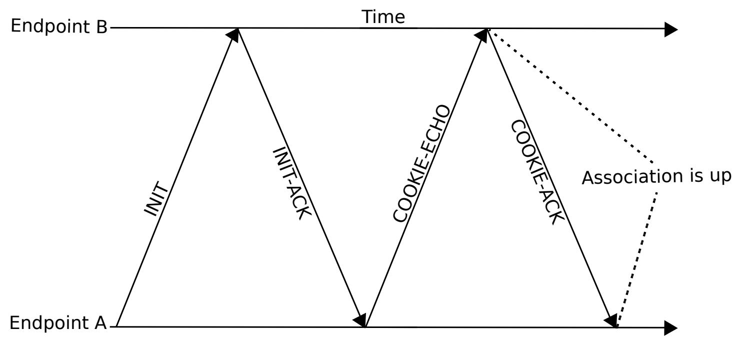


Figure 2.2: A view of the association setup phase.



2.5.1 The association setup phase

SCTP uses a 4-way handshake in contrast to TCP's 3-way handshake [4]. This was chosen as means to avoid TCP's vulnerability to a SYN packet flooding denial of service attack [2]. SCTP sends a signed state cookie [1] to protect against this form of attack[3].

The signed state cookie should contain:

- A timestamp indicating when the cookie was created.
- A Message Authentication Code (MAC) [20].
- Lifespan of the state cookie
- Information necessary to establish the association.

While an additional packet exchange in the handshake often indicates additional overhead, it is worth noting that two of the SCTP packets in the handshake can carry other types of information, such as user data. This is done to minimize the delay burden for the application without compromising the improved security. As is shown in Figure 2.2, there are four chunks involved in a SCTP association setup.

The cookie is embedded inside the INIT-ACK chunk and is echoed back to endpoint Z. When this is received, a COOKIE-ACK chunk is sent back and the association is set up.

[2]"*A SYN flooding attack is one of a number of denial-of-service attacks that have been used on the Internet. It is usually executed by a malicious host (the attacker) sending a targeted host (the victim) a large number of SYN messages. (The SYN message is the first setup message in a TCP connection, similar to SCTP's INIT message)*" [19]

[3]Note that more recent implementations of TCP have incorporated a similar system, but they do not use a signed cookie.

Here is a short review of each chunk involved in the handshake:

- INIT
  This is the client initiating an association to another endpoint. It contains information (in the *a_rwnd* field) about:

  - How much buffer resources the initiator has dedicated to the association.
  - How many streams the local user is requesting to open to the remote peer.
  - The maximum number of inbound streams it is capable of supporting.
  - The *initiation tag* value. This value serve as a mechanism to verify that an SCTP packet truly belongs to this association.
  - A list of addresses available to the association.

  Before the chunk is transmitted, the sender starts a timer called the *T1-init* timer at his end so that if the chunk is lost, in other words if no INIT-ACK returns to the initiator, it can be retransmitted in the event that the timer expires. The retransmission also restarts the timer. A counter keeps track of how many retransmissions have been performed, and if it eventually reaches some threshold the upper layer will be informed that the destination host is unreachable, and the initiator will give up.

- INIT-ACK
  This is generated at the receiving side of an INIT chunk. The receiver will **not** allocate memory to store information about this association yet (this would make it vulnerable to exhausting its resources in the event of a denial of service attack), but it must decide what values would go there if it were to do it. In particular it will form a cookie. It contains the receiving side's *a_rwnd* field, how many streams to open and a corresponding list of reachable addresses as was included in the INIT chunk. The generated cookie should contain a timestamp of when it was created, a time to live value and a signature for authentication, to help the next step ensure that the cookie has not been tampered with.

- COOKIE-ECHO
  When receiving the INIT-ACK chunk, the initiator will stop its *T1-init* timer and reset any retransmission counter. Then it will proceed to update its address list with the addresses read from the INIT-ACK chunk. The host then starts a new timer called the *T1-cookie* timer. Finally the host will pack the received cookie into a COOKIE-ECHO chunk and send it back to the sender. The COOKIE-ECHO chunk can as mentioned earlier also contain data, so in the event of data waiting to be transmitted, it can be added here.

- COOKIE-ACK
  When the COOKIE-ECHO is received, the recipient will verify the cookie's authenticity and build a Transmission Control Block (TCB). The TCB is an internal data structure containing a set of information that an endpoint must maintain in order to manage a SCTP association, the exact structure of the TCB is implementation dependent. If there is data to be read in the COOKIE-ECHO chunk,

Figure 2.3: An SCTP packet

IP Header
SCTP common header
Chunk 1
Chunk 2
Chunk 3
...
Chunk N

Figure 2.4: The SCTP common header

Bits 0-7	**8-15**	**16-23**	**24 - 31**
Source port		Destination port	
Verification tag			
Checksum			

it will be processed and the host will send back a COOKIE-ACK chunk. The COOKIE-ACK is just meant to tell the initiator that the cookie has been received and accepted, that the peer can turn off its *T1-cookie* timer and that the peer can change the internal state of the new association to ESTABLISHED.

2.6 SCTP Message structure

As can be seen in Figure 2.3 an SCTP packet consists of a **SCTP common header** which contains properties needed to control and maintain an association. A packet also contains a variable number of **chunks** which will be covered in Subsection 2.6.2.

2.6.1 The SCTP common header

Figure 2.4 shows the SCTP common header. It provides three basic services:

- A method to associate a SCTP packet with an association - The source/destination port
- Verification that the SCTP packet belongs to the current instance of this association - The verification tag
- Transport-level verification that the data is intact and unaltered by inadvertent network errors. - The checksum

The common header contains a source port number, a destination port number, a verification tag and a checksum. The verification tag ensures that the packet does not belong to an earlier SCTP association between the two peers, and it makes it more difficult for an attacker to inject data into an existing association. The chunks in the message, and the header itself form the basis for the checksum, which is used to verify the itegrity of the packet (i.e. help ensure that the packet has not been tampered with).

Figure 2.5: The chunk header

Bits 0-7	**8-15**	**16-23**	**24 - 31**
Chunk 1 type	Chunk 1 flags	Chunk 1 length	

Table 2.5: Chunk types

Chunk number	Chunk type
0	DATA
1	INIT
2	INIT ACK
3	SACK
4	HEARTBEAT
5	HEARTBEAT ACK
6	ABORT
7	SHUTDOWN
8	SHUTDOWN ACK
9	ERROR
10	COOKIE ECHO
11	COOKIE ACK
12	ECNE
13	CWR
14	SHUTDOWN COMPLETE
15-62	Reserved by IETF
63	IETF-defined chunk extensions
64-126	Reserved by IETF
127	IETF-defined chunk extensions
128-190	Reserved by IETF
191	IETF-defined chunk extensions
192-254	Reserved by IETF
255	IETF-defined chunk extensions

2.6.2 Chunks

Chunks are the basic building blocks meant to carry information in SCTP. They come in two main types:

- Control chunks
  The control chunks carry information, for controlling and maintaining an association.

- Data chunks
  The data chunks carry user messages across an association.

Each chunk comes with a chunk header that describes what type of chunk it is, and a chunk-type specific flags field. The chunk length says how long the chunk is, including the chunk header itself (See Figure 2.5). That means that for a chunk that has no data, the chunk length will still be 4 bytes.

Figure 2.6: The DATA chunk

Bits 0-7	8-12	13	14	15	16-31
Chunk type = 0	Reserved	U	B	E	Chunk length
Transmission Sequence Number (TSN)					
Stream identifier (SID)					Stream sequence number (SSN)
Payload protocol identifier (PPID)					
Data					

RFC 2960 [1] defines 16 chunk types (See Table 2.5), leaving space for an additional 240 chunk types that may be defined in the future. The concept of chunks was chosen for its extensibility, and new chunk types can be added as fits.

A closer look at the DATA chunk

The DATA chunk (shown in Figure 2.6) carries user messages. A list of the options and their respective meaning is given below:

- U - If set to 1, this chunk is unordered.
- B - Beginning fragment bit, indicates that this is the first fragment of a fragmented message.
- E - Ending fragment bit, indicates that this is the last fragment of a fragmented message.

Each chunk is assigned a 32-bit **Transmission Sequence Number (TSN)**. It allows the endpoint to detect duplicate deliveries and tell the sending part that the chunk has been received successfully.

The **Stream Sequence Number (SSN)** is a 16-bit value that ensures sequenced delivery of a message within a given stream. Unordered messages do not have a SSN, and fragments of a message all carry the same SSN. This is basically what makes an unordered message "unordered". Normally, with ordered messages, if a DATA chunk (See Figure 2.6) arrives out of order with the U bit set to 0, it must be held back until the full message can be reassembled. On the other hand, DATA chunks with the U bit set to 1 will tell the endpoint to bypass its ordering mechanism, and just deliver the chunk to the upper layer as soon as it arrives.

Finally, the **Payload Protocol Identifier (PPID)** is just carried through the SCTP stack. It has no functionality in SCTP by itself, but can be used by network entities and applications as necessary.

2.7 Summary

This chapter has looked at the current state of the Linux implementation of SCTP and introduced some key concepts that will be of interest for the remainder of this thesis. Especially ancillary data and notifications will be touched upon in the next chapter, which will present all discrepancies between the Linux implementation of SCTP and the API defined in RFC 6458 that were found as part of the work with this thesis.

Chapter 3

Changes in RFC 6458

A big part of the research conducted while working on this thesis has revolved around figuring out which aspects of SCTP that would need to be changed due to changes in RFC 6458. Thus, this chapter will provide an overview of the discrepancies found to be missing from the current Linux implementation of SCTP. There are mainly four types of changes that have been found.

1. New helper functions to handle ancillary data.
2. Various changes related to the structure of ancillary data.
3. Changes to how notification interest is specified.
4. New socket options, and ways to set socket options on a more fine-grained level.

3.1 New functions

The old API functions are implemented in the userspace library known as *libsctp*. Of all the functions introduced in RFC 6458, only three functions have not been found in *libsctp*:

- *sctp_sendx()*
- *sctp_sendv()*
- *sctp_recvv()*

Note that *sctp_sendx()* is deprecated, and should be replaced with *sctp_sendv()*. Thus, it will not be covered in this thesis.

sctp_sendv()

The function *sctp_sendv()* provides an extensible way for an application to send various attributes to the SCTP stack when sending a message. In this case, extensible means that it has been designed to be very "open" as to what types of ancillary data that can be attached to a message. It makes it easy to add new types of ancillary data in the future.

```
ssize_t sctp_sendv(
    int sd,
    const struct iovec *iov,
    int iovcnt,
    struct sockaddr *addrs,
    int addrcnt,
    void *info,
    socklen_t infolen,
    unsigned int infotype,
    int flags);
```

Listing 3.1: Prototype of *sctp_sendv()*.

According to RFC 6458 it can be implemented as a library function or a system call. The full prototype of this function as it is defined in RFC 6458, is shown in Listing 3.1. Most notably, there are two things that sets *sctp_sendv()* apart from the bare *sendmsg()*-approach[1]:

1. First, the combination of the parameters *info*, *infolen* and *infotype* will together indicate whether the message includes any of the following types:

    - *struct sctp_sndinfo* - General send parameters as will be described in Section 3.2
    - *struct sctp_prinfo* - Parameters related to Partial Reliability SCTP [21].
    - *struct sctp_authinfo* - Parameters related to AUTH SCTP [18].
    - *struct sctp_sendv_spa* - The *struct sctp_sendv_spa* is a collection structure used when more than one setting is to be set at the same time. Listing 3.2 shows how this structure is defined.

    ```
    struct sctp_sendv_spa {
        uint32_t sendv_flags;
        struct sctp_sndinfo sendv_sndinfo;
        struct sctp_prinfo sendv_prinfo;
        struct sctp_authinfo sendv_authinfo;
    };
    ```

    Listing 3.2: struct sctp_sendv_spa

    For one-to-many style sockets it is necessary to always include a *struct sctp_sndinfo* to specify which association(s) to affect[2]. This is an example of when the *struct sctp_sendv_spa* is needed. A developer would copy structs to their respective spots in the struct shown in Listing 3.2, and then proceed to set the *sendv_flags* field to a bitwise OR of either SCTP_SEND_SNDINFO, SCTP_SEND_PRINFO or

[1]In fact, *sctp_sendv()* and *sctp_recvv()* are just a convenient wrappers around *sendmsg()* and *recvmsg()*. The examples in Section A.1 and A.2 in the Appendix shows how this can be accomplished.

[2]Except when the *sctp_sendv()* call is used to setup an implicit association (RFC 6458 [3], Section 7.5.)

SCTP_SEND_AUTHINFO to indicate which of the three fields that should be included by *sctp_sendv()*.

2. Secondly, the caller can provide a list of addresses in the *addrs* parameter shown in Listing 3.1. These addresses can be used to set up an association or send to a specific address. If NULL is passed, the message will be sent to whichever other endpoint the socket is connected to.

sctp_recvv()

Like *sctp_sendv()*, *sctp_recvv()* provides a way to receive attributes from the SCTP stack to an application. Listing 3.3 shows its prototype, which is defined in RFC 6458 in a similar fashion as the one shown earlier for *sctp_sendv()*.

```
ssize_t sctp_recvv(int sd,
    const struct iovec *iov,
    int iovlen,
    struct sockaddr *from,
    socklen_t *fromlen,
    void *info,
    socklen_t *infolen,
    unsigned int *infotype,
    int *flags);
```

Listing 3.3: Prototype of *sctp_recvv()*.

RFC 6458 defines two types of attributes that can be returned by this function. The attributes of the received message, and/or those of the next message. Before receiving either of these (or both), the RECVRCVINFO and RECVNXTINFO socket options must be enabled to tell the SCTP stack which one(s) are desired. As with *sctp_sendv()*, there is a collection structure defined if both socket options are on. This is the *struct sctp_-recvv_rn* shown in Listing 3.4.

```
struct sctp_recvv_rn {
    struct sctp_rcvinfo recvv_rcvinfo;
    struct sctp_nxtinfo recvv_nxtinfo;
};
```

Listing 3.4: struct sctp_recvv_rn

These structures will be shown in more detail in the next section. Before making a call to *sctp_recvv()* the caller must prepare a buffer for the message and point the *iov_base* field of the *struct iovec* to it. Also a pointer to a buffer to store the address of the sender should be provided in the *from*-parameter to *sctp_recvv()*. The *info* pointer is where the ancillary data will be stored, and the *infolen* and *infotype* will be filled appropriately.

3.2 Ancillary data

The new ancillary data types are listed in Table 3.1. As was mentioned in Section 2.1 the *struct sctp_sndrcvinfo* was split into two smaller structs, *struct sctp_sndinfo* and *struct sctp_rcvinfo*. Listing 3.5 shows how these are defined in RFC 6458.

In short, the fields in Listing 3.5 give the application developer the possibility to change various settings. A few examples will be provided here.

The field *snd_sid* specifies which stream number to send the message on. An example of how this can be done with *sctp_sendv()* is shown in Listing 3.6. Conversely, *rcv_sid* is used to check which stream a message came in on. As long as the stream number is within a valid range, i.e. within the range of available in/out streams of the communicating parties,[3] this would send the message on stream number 5.

The *snd_flags* field can be used to tell the SCTP stack various things. The field contains a bitwise OR of one or more of the following options:

- SCTP_UNORDERED
  Setting this flag requests the message to be delivered unordered.

- SCTP_ADDR_OVER
  Setting this requests that the SCTP stack overrides the primary address destination address with the one found in the call.

- SCTP_ABORT
  Setting this flag causes the specified association to abort by sending an ABORT message to the peer.

- SCTP_EOF
  Setting this flag invokes a graceful shutdown procedure on the specified association. A graceful shutdown makes sure all untransmitted data is transmitted before closing the association.

- SCTP_SENDALL
  Setting this flag will cause a one-to-many style socket to send the message to all associations currently established on the socket.

As most of these flags are directions to the SCTP stack, only SCTP_UNORDERED can be seen from the receivers end. Hence, only SCTP_UNORDERED can possibly end up in *rcv_flags* after a call to *sctp_recvv()*, telling the receiver that the received message was sent out of order.

More information on what each individual field in *sctp_sndinfo* and *sctp_rcvinfo* can be used for can be found in Section 5.3.4. and 5.3.5. of RFC 6458.

Note that the only field in Listing 2.3 that has been changed its name is the *sinfo_-stream* field which has been renamed to *snd/rcv_sid* for the two new structures. Only the *sinfo_timetolive* flag was left out. Setting this value was moved to Partial Reliability SCTP (PR-SCTP) with the policy SCTP_PR_SCTP_TTL. The next sections will contain a brief explanation of some of these fields.

[3]Section 8.2.1. of RFC 6458 [3] shows how to get an endpoints available in/out streams.

Table 3.1: New ancillary data types in RFC 6458.

Description	**cmsg_type**	**cmsg_data[]**
SCTP Send Information Structure	SCTP_SNDINFO	struct sctp_sndinfo
SCTP Receive Information Structure	SCTP_RCVINFO	struct sctp_rcvinfo
SCTP Next Receive Information Structure	SCTP_NXTINFO	struct sctp_nxtinfo
SCTP PR-SCTP Information Structure	SCTP_PRINFO	struct sctp_prinfo
SCTP AUTH Information Structure	SCTP_AUTHINFO	struct sctp_authinfo
SCTP Destination IPv4 Address Structure	SCTP_DSTADDRV4	struct in_addr
SCTP Destination IPv6 Address Structure	SCTP_DSTADDRV6	struct in6_addr

```c
struct sctp_sndinfo {
    uint16_t snd_sid;
    uint16_t snd_flags;
    uint32_t snd_ppid;
    uint32_t snd_context;
    sctp_assoc_t snd_assoc_id;
};

struct sctp_rcvinfo {
    uint16_t rcv_sid;
    uint16_t rcv_ssn;
    uint16_t rcv_flags;
    uint32_t rcv_ppid;
    uint32_t rcv_tsn;
    uint32_t rcv_cumtsn;
    uint32_t rcv_context;
    sctp_assoc_t rcv_assoc_id;
};
```

Listing 3.5: The new structs: *struct sctp_sndinfo* and *struct sctp_rcvinfo*.

```
struct sctp_sndinfo snd;
memset(&snd, 0, sizeof(snd));

snd.snd_sid = 5;
snd.snd_flags |= SCTP_SENDALL;

/* struct sockaddr_in *from  - Address of the recipient.
sctp_sendv(sockfd, /* Socket identifier */
    msg.msg_iov,         /* The msghdr's data storage. */
    1,                   /* Number of iovec structs */
    &from,               /* Address of recipient */
    1,                   /* Number of addresses */
    &snd,                /* Ancillary data */
    sizeof(snd),         /* Ancillary data length */
    SCTP_SENDV_SNDINFO,  /* Type of ancillary data */
    0);                  /* Flags redirected to sendmsg() */
```

Listing 3.6: Example: Selecting a stream number with *struct sctp_sndinfo* and *sctp_-sendv()*.

```
    struct sctp_nxtinfo {
        uint16_t nxt_sid;
        uint16_t nxt_flags;
        uint32_t nxt_ppid;
        uint32_t nxt_length;
        sctp_assoc_t nxt_assoc_id;
    };
```

Listing 3.7: struct sctp_nxtinfo

3.2.1 struct sctp_nxtinfo

In addition to the split of *struct sctp_{snd/rcv}info*, an extended version of *sctp_rcvinfo* was present prior to RFC 6458. The structure *struct sctp_extrcvinfo* contained the same information that *struct sctp_sndrcvinfo* normally would, but also information about the next message to be received by *recvmsg()*. With RFC 6458, *struct sctp_extrcvinfo* has been replaced with *struct sctp_nxtinfo*, which is shown in Listing 3.7.

3.3 Notifications

Although the events themselves have not changed much, the way they are handled have changed a bit. The SCTP_EVENTS socket option which was described in Section 2.2 has been replaced with a new socket option SCTP_EVENT. The SCTP_EVENTS option that was used prior to RFC 6458 was not scalable enough. According to the RFC, the *struct sctp_event_subscribe*, which was introduced in Subsection 2.2.1 would have to be expanded as new events are added to SCTP. This can cause an application binary interface conflict, unless the implementation adds padding at the end of the structure. To avoid this, SCTP_EVENTS has been deprecated and the new socket option SCTP_-EVENT will take its place.

In addition, there are two minor changes to the actual events:

1. The event type SCTP_SEND_FAILED has been deprecated and replaced with SCTP_SEND_FAILED_EVENT. They seem to be pretty much identical, except that SCTP_SEND_FAILED_EVENT uses a *struct sctp_sndinfo* in its *ssfe_info* field. This event, if enabled, will be sent to the application layer if SCTP is unable to deliver a message. It includes a SCTP error code, which will reveal what type of failure has occurred (more details on SCTP error codes can be found in section 3.3.10. in RFC 4960 [14]).

2. The SCTP_NOTIFICATIONS_STOPPED_EVENT is missing. According to RFC 6458 this event might trigger if the implementation runs out of socket buffer space. When this occurs, it might wish to disable notifications and it will notify the application layer of this by sending this event.

3.4 Socket options

As was briefly mentioned in Subsection 1.1.1, socket options in SCTP are set with the function *setsockopt()*. Its prototype is shown in Listing 3.8.

```
    int setsockopt(
        int sd,                /* File descriptor */
        int level,             /* Protocol number */
        int optname,           /* Name of option to change */
        const void *optval,    /* Option input */
        socklen_t optlen);     /* Length of option input */
};
```

Listing 3.8: setsockopt()

Although this function is specified in the operating systems socket API, calls to this function with a *level* of *IPPROTO_SCTP* gets redirected to the SCTP function *sctp_-setsockopt()* which is defined in *socket.c* in the Linux kernel implementation.

This function can be used to ask the SCTP stack for a lot of things, and some of them will be listed here. The findings related to socket options will be presented in the following subsections.

3.4.1 Selecting which associations to affect

Socket options set on a one-to-one style sockets automatically apply to all future sockets, but a more fine-grained scheme is necessary for one-to-many sockets. Therefore, many socket options that can be used with one-to-many style sockets include an *sctp_-assoc_id* field in the structure that gets passed to the SCTP stack.

```
/* This structure is defined in linux/sctp.h */
struct sctp_assoc_value {
    sctp_assoc_t assoc_id;
    uint32_t assoc_value;
};

/* Prepare an instance of the struct */
struct sctp_assoc_value myopt;
memset(&myopt, 0, sizeof(struct sctp_assoc_value));

/* Set desired values */
myopt.assoc_id = SCTP\_FUTURE\_ASSOC;
myopt.assoc_value = 100;

/* Send the settings to the SCTP stack */
setsockopt(sock,
        IPPROTO_SCTP,
        SCTP_MAXSEG,
        &myopt,
        sizeof(struct sctp_assoc_value));
```

Listing 3.9: Setting the MAX_SEG socket option

Listing 3.9 shows an example of how to set a socket option. To set the maximum fragmentation size of a DATA chunk, a structure of type *sctp_assoc_value* specifies both which association the new setting should affect, and how many bytes of data the DATA chunks should be limited to.

For this particular socket option, SCTP_MAXSEG, the *assoc_id* field can be set to either a specific association ID, which will make the option only affect that association, or it can be set to SCTP_FUTURE_ASSOC. Different socket options allow different combinations of these flags. These are the available options:

- SCTP_CURRENT_ASSOC
  Only the currently existing associations on the socket will be affected by this call.

- SCTP_FUTURE_ASSOC
  Only future associations that are established on the socket will be affected.

- SCTP_ALL_ASSOC
  All associations, both current and future will be affected.

Since these three flags are not currently implemented on Linux, the example shown in Listing 3.9 would unfortunately not work at this point.

The socket options that would need to be modified to accomodate for this change are listed in Table 3.2. Note that this is not all socket options available in SCTP, merely the ones that these flags would affect.

Table 3.2: Affected socket options

Socket option	CURRENT	FUTURE	ALL	ID
SCTP_RTOINFO	✗	✓	✗	✓
SCTP_ASSOCINFO	✗	✓	✗	✓
SCTP_PRIMARY_ADDR	✗	✗	✗	✓
SCTP_PEER_ADDR_PARAMS	✗	✓	✗	✓
SCTP_DEFAULT_SEND_PARAMS	✓	✓	✓	✓★
SCTP_MAXSEG	✗	✓	✗	✓★★
SCTP_AUTH_ACTIVE_KEY	✓	✓	✓	✓
SCTP_DELAYED_SACK	✓	✓	✓	✓
SCTP_MAX_BURST	✓	✓	✓	✓
SCTP_CONTEXT	✓	✓	✓	✓
SCTP_EVENT	✓	✓	✓	✓
SCTP_DEFAULT_SNDINFO	✓	✓	✓	✓
SCTP_DEFAULT_PRINFO	✓	✓	✓	✓
SCTP_LOCAL_AUTH_CHUNKS	✗	✓	✗	✓
SCTP_AUTH_KEY	✓	✓	✓	✓
SCTP_AUTH_DEACTIVATE_KEY	✓	✓	✓	✓
SCTP_AUTH_DELETE_KEY	✓	✓	✓	✓

✓: Should have support for this flag
✗: Flag is not used here
★: This socket option has been deprecated with RFC 6458.
★★: The RFC is a bit unclear on this point. It says it is illegal to use SCTP_CURRENT | ALL_ASSOC,but does not specifically say that SCTP_FUTURE_ASSOC is allowed.

3.4.2 New socket options

All of the following socket options are not available in Linux, or their implementation has been found to be slightly different than what is proposed in RFC 6458. A short description on what has been found for each socket option will be given below.

SCTP_FRAGMENT_INTERLEAVE

The fragmented interleave socket option controls whether the SCTP stack should postpone sending other messages while an endpoint is in the process of receiving a partial delivery. According to RFC 6458, this option can be set to 3 different levels:

- Level 0: Block incoming messages on both other streams and associations.
- Level 1: Block incoming messages on other streams, but not on other associations.
- Level 2: Allow incoming messages on both streams and associations.

The suggested default is level 1, note that for one-to-one style sockets both level 0 and level 1 is the same, as they only have one association. The socket option is implemented in Linux, but only with two levels. It has not been fully mapped out during the writing of this thesis exactly which two levels are available, but it is safe to assume that either level 0 or 1 are missing. Thus, the ability to give one-to-many style sockets an optional fine-grained control over what to do is not implemented.

SCTP_GET_PEER_ADDR_INFO

Section 8.2.2. of RFC 6458 says that when the field *spinfo_assoc_id* in struct *sctp_paddrinfo* is set, it should be prioritized over the *spinfo_address*. Although the socket option is supported, this particular detail is currently not considered in the Linux implementation of SCTP.

SCTP_USE_EXT_RCVINFO

This socket option has been deprecated according to section 8.1.22. of RFC 6458 [3]. The "extended" part of the *struct sctp_sndrcvinfo* added data fields for the next message, which has been reworked to rather use the SCTP_NXTINFO (Described in Subsection 3.2.1). Nothing related to *struct sctp_extrcvinfo* has been implemented in Linux.

SCTP_REUSE_PORT

With this socket option a user can tell the SCTP stack to reuse the port of another socket that has been registered with the REUSE_PORT setting enabled. This does not mean that two sockets can be listening on the same port, but might enable multiple sockets to "take turns" listening. This option only applies to one-to-one style sockets.

SCTP_EVENT

Prior to RFC 6458 the events system was handled with the SCTP_EVENTS option (As described in Section 2.2). The scalability of this solution was not optimal, as it relied on a *struct sctp_event_subscribe* to describe which notifications the user was interested in. As new features are added to SCTP this structure will have to be expanded, and this can cause scalability issues. This notification interest mechanism has not been implemented in Linux.

SCTP_RECVRCVINFO & SCTP_RECVNXTINFO

These options are used to enable receival of the new *struct sctp_rcvinfo* and *struct sctp_-nxtinfo* as was described in Section 3.2. They replace the *data_io_event*-member in *struct sctp_event_subcribe*, but neither are currently implemented in SCTP for Linux.

SCTP_DEFAULT_SNDINFO

If a developer does not want to use the *sctp_sendv()* call to pass ancillary data, it is with this socket option possible to set default send parameters. It is as simple as filling a *struct sctp_sndinfo* and passing it along to the *setsockopt()* call. This option replaces the SCTP_DEFAULT_SEND_PARAM, but is not currently implemented in SCTP for Linux.

SCTP_DEFAULT_PRINFO

This socket option is used to set or get the the default parameters for PR-SCTP. [3] It has not been implemented in SCTP for Linux.

SCTP_AUTH_DEACTIVATE_KEY

A shared secret key can be used to built an association shared key. This can be set up with the SCTP_AUTH_KEY socket option. A mechanism to deactivate it has not been implemented in Linux, but deleting is possible with the SCTP_AUTH_KEY_DELETE socket option.

SCTP_EXPLICIT_EOR

This socket option is described in section 8.1.26. of RFC 6458 [3]. It is not currently available in SCTP for Linux. With this socket option enabled a developer should be able to make multiple send system calls and then make sure to have the MSG_EOR flag set on the last record. The MSG_EOR flag part of the sockets API is defined in *socket.h*. To implement this, some work would need to be done in the file *net/sctp/ulpqueue.c* in the Linux kernel.

3.5 Summary

Chapter 3 has presented a rundown of possible improvements in the current SCTP implementation for Linux. This includes new helper functions, changes to the handling of ancillary data and notifications, and a new scheme for dealing with socket options. In the next chapter we will have a look at how some of these improvements have been designed as part of the work with this thesis.

Chapter 4

Design and Implementation

This chapter will examine in detail the changes that were designed, implemented and evaluated on the Linux SCTP code base as part of this thesis. These implementations were done with Linux kernel version 3.12.9, downloaded from https://www.kernel.org in January 2014. Also the files *sctp_sendv.c* and *sctp_recvv.c* were added to *libsctp* version 1.0.15, downloaded from http://lksctp.sourceforge.net in January 2014. Section A in the Appendix contains hyperlinks to the git repositories where all code can be found.

In total, five files from the kernel tree were modified and one file in the userspace library in addition to the two new files that were added to the latter: *sctp_recvv.c* and *sctp_sendv.c*.

Modified files - Linux kernel:

- socket.c (src/linux-3.12/net/sctp/)
- ulpevent.c (src/linux-3.12/net/sctp/)
- structs.h (src/linux-3.12/include/net/sctp/)
- ulpevent.h (src/linux-3.12/include/net/sctp/)
- sctp.h (src/linux-3.12/include/uapi/linux/)

Modified files - Userspace (*libsctp*):

- sctp_sendv.c (src/lib/)
- sctp_recvv.c (src/lib/)
- sctp.h (src/include/netinet/)

Early on, it became apparent that the first thing that needed to be done, was to update the header file with all the new structures, flags and other related constants. As these flags, structures and constants are what everything else depends on, it was necessary to decide on a suitable location for them. In FreeBSD most of these seemed to have been defined in the files *sys/netinet/sctp_uio.h* and *sys/netinet/sctp/sctp.h*. Many, if not all, of the definitions found in these files were also found in the file *include/uapi/linux/sctp.h* on Linux, so it seemed reasonable to assume that adding the new flags, structures and constants would best be done here. As an example, the socket options were defined in *sys/netinet/sctp/sctp.h* on FreeBSD. As the socket options in FreeBSD were simply numbered constants, they were added in the same way to the already existing socket options defined in Linux. There were 31 options already defined, so the numbers were initially counted from 32 and onwards as shown in Listing 4.1.

```
#define SCTP_RECVRCVINFO        32
#define SCTP_RECVNXTINFO        33
#define SCTP_DEFAULT_SNDINFO    34
```

Listing 4.1: Socket options implemented in this thesis

The following sections will describe every accomplished implementation that has been performed as part of this thesis.

First, the next section will describe the process of getting the ancillary data *struct sctp_sndinfo* transferred properly with as little change to the existing structure as possible. Then, Section 4.2 will cover the implementation of the new socket options, and finally Section 4.3 and Section 4.4 will present the new functions that been implemented and an overview of the new flags for *struct sctp_sndinfo*.

4.1 Implementing support for sending ancillary data

Being able to send ancillary data in the form of the *struct sctp_sndinfo* proved to be a bit more complicated than receiving *struct sctp_rcvinfo*. This section will show a few of the changes that were done to accommodate for this. The function *sctp_sendmsg()* in the file *socket.c* calls a function called *sctp_msghdr_parse()* (shown in Listing 4.2). This function takes a *struct msghdr* (Listing 2.1) and a pointer to a location to store the *cmsghdrs* that is extracted from the *struct msghdr*. During this project, this storage structure has been changed slightly to be able to hold other kinds of ancillary data than just the old *struct sctp_sndrcvinfo*. The storage structure, *sctp_cmsgs_t*, is shown in Listing 4.3. Previously, it had only two fields: a pointer to a *struct sctp_initmsg*[1] and a pointer to a *struct sctp_sndrcvinfo*. The latter was replaced with the *void *info* seen in the listing. The purpose of this change is to make the *sctp_cmsgs_t* able to hold both a *struct sctp_sndinfo*, or the old *struct sndrcvinfo*. The *cmsg_type*-field was then added, to be able to distinguish between them. This separation of the *cmsg* types is done in the function *sctp_msghdr_parse()*. The next subsection will show what happens inside *sctp_msghdr_parse()*, and we will return to *sctp_sendmsg()* to see what happens further in Subsection 4.1.2.

```
    static int sctp_msghdr_parse(const struct msghdr *msg,
        sctp_cmsgs_t *cmsgs);
```

Listing 4.2: Prototype of sctp_msghdr_parse

[1]Nothing related to this structure has changed in this thesis. It is described in section 5.3.1. of RFC 6458 [3].

```
/* A convenience structure to parse out SCTP specific CMSGs
    */
typedef struct sctp_cmsgs {
    struct sctp_initmsg *init;
    void * info;
    sctp_cmsg_t cmsg_type;
} sctp_cmsgs_t;
```

Listing 4.3: The new *struct sctp_cmsgs_t* in *structs.h*.

4.1.1 sctp_msghdr_parse()

The *sctp_msghdr_parse()* function loops through all control messages (*cmsgs*) in the given *struct msghdr*. First it skips all *cmsgs* that is not of *cmsg_level* IPPROTO_SCTP. In other words, *cmsgs* that are not meant for SCTP. Then the function proceeds to check the type of the *cmsg*. Prior to this project, the supported *cmsg* types were SCTP_INIT or SCTP_SNDRCV. If it finds a *cmsg* type it recognizes, it checks that the *cmsg->cmsg_len* field is of correct size and then returns the CMSG_DATA [2] of the *cmsg* back to the given storage structure *sctp_cmsgs_t* mentioned in the previous section.

Just like the ones that were supported in that storage structure *sctp_cmsgs_t* in the previous section, the *sctp_msghdr_parse()*-function also needs support for the new *cmsg_type* SCTP_SNDINFO. The code shown in Listing 4.4 shows the code that was added to *sctp_msghdr_parse()* as part of this thesis.

At line 14-17 size-validation is done in the same manner as was already done for the existing SCTP_SNDRCV case. The function then sets the *cmsgs->info* and *cmsgs->cmsg_type* properly. Line 22-26 checks that only allowed flags are being given, as was done for the old SCTP_SNDRCV too. This time the SCTP_SENDALL flag was added, as this is to be supported by the new API.

4.1.2 Back in sctp_sendmsg()

After *sctp_msghdr_parse()* has found its control messages and made them available for the *sctp_sendmsg()* function, *sctp_sendmsg()* proceeds by validating a lot of things. It ensures that flags are set correctly in regards to what type of socket is being used, fetches the destination address for the message and performs various other checks. Next, *sctp_sendmsg()* looks up the association, and checks that it is open for communication, i.e. not in state CLOSED. According to comments in *socket.c* this can happen with certain one-to-one style sockets. If an association is not found, a new one will be created[3]. After the association has been found or created, handling of the ancillary data begins. Listing 4.5 shows another piece of code that has been contributed as part of this thesis; How the new control message is being detected and kept for use later in the function.

[2]It is recommended to handle control messages in Linux by utilizing a certain set of macros created for this purpose. CMSG_DATA is one of these [16].

[3]Implicit association setup is described in section 7.5. of RFC 6458 [3].

```
...
case SCTP_SNDINFO:
    /* SCTP Socket API Extension
     * 5.3.4 SCTP Send Information Structure (SCTP_SNDINFO)
     *
     * This cmsghdr structure specifies SCTP options for
     * sendmsg(). This structure and SCTP_RCVINFO replaces
     * SCTP_SNDRCV which has been depleted.
     *
     * cmsg_level    cmsg_type      cmsg_data[]
     * ------------  ------------   ---------------------
     * IPPROTO_SCTP  SCTP_SNDINFO    struct sctp_sndinfo
     * */
    if(cmsg->cmsg_len !=
            CMSG_LEN(sizeof(struct sctp_sndinfo))){
        return -EINVAL;
    }

    cmsgs->info = (struct sctp_sndinfo *)CMSG_DATA(cmsg);
    cmsgs->cmsg_type = SCTP_SNDINFO;

    if (((struct sctp_sndinfo *) cmsgs->info)->snd_flags &
            ~(SCTP_UNORDERED | SCTP_ADDR_OVER |
                SCTP_ABORT | SCTP_EOF | SCTP_SENDALL)){
        return -EINVAL;
    }

    break;
...
```

Listing 4.4: Excerpt from *sctp_msghdr_parse()*: Adding support for *SCTP_-SNDINFO*

```
    ...
    if(cmsgs.cmsg_type == SCTP_SNDINFO){
        /* Put the cmsg data into a temporary struct sctp_sndinfo and
           move it into the struct sctp_sndrcvinfo default_sinfo; */
        sndinfo = (struct sctp_sndinfo*) cmsgs.info;
        memset(&default_sinfo, 0, sizeof(default_sinfo));

        default_sinfo.sinfo_flags = sndinfo->snd_flags;
        default_sinfo.sinfo_stream = sndinfo->snd_sid;
        default_sinfo.sinfo_assoc_id = sndinfo->snd_assoc_id;
        default_sinfo.sinfo_ppid = sndinfo->snd_ppid;
        default_sinfo.sinfo_context = sndinfo->snd_context;

        sinfo = &default_sinfo;
    }else{
        /* cmsgs.info could be NULL, but will be replaced by a default sinfo later,
           since sndinfo has not been set. */
        sinfo = cmsgs.info;
    }
...
```

Listing 4.5: Excerpt from *sctp_sendmsg()*: Handling a sendmsg() with struct_-sndinfo

The *default_sinfo* structure is of type *sctp_sndrcvinfo*. It is intended to be used when no ancillary data has been given. It is in this case just filled up with defaults taken from the association. Earlier, the default approach was taken if no control message of type SCTP_SNDRCV was found, but it has been modified to include the condition that no control message of type SCTP_SNDINFO has been found either, leading to this default creation not happening when a SNDINFO control message has been found. At line 14 in Listing 4.5 the *sinfo* pointer (of type *struct sctp_sndrcvinfo **) is set to point to this new default structure. Hence, sending a control message of type SCTP_SNDINFO is handled just the same way as the SCTP_SNDRCV was handled previously.

4.2 Implementing new socket options

The socket options that were implemented in this project were all related to ancillary data. Two of them are simple boolean options, that can be set either on or off. The last one is the replacement of the deprecated option *SCTP_DEFAULT_SEND_PARAM*, now *SCTP_DEFAULT_SNDINFO*, which is basically a part of the move from the *struct sndrcvinfo* mentioned in Subsection 2.1.3 to *struct sndinfo* and *struct rcvinfo*. A little more details on the implementation of each one will be provided here.

SCTP_DEFAULT_SNDINFO

As described in Subsection 3.4.2, this option just sets the default *struct sctp_sndinfo* to be used, in cases where no *struct sctp_sndinfo* is provided. Apart from adding the constant *SCTP_DEFAULT_SNDINFO* to *include/uapi/linux/sctp.h*, all changes related to this option has been done in the file *net/sctp/socket.c*. In this latter file, three areas needed to be modified. The function *sctp_setsockopt()* is called first (thanks to the *struct proto*, which will be shown in Section 4.3). This function starts off by verifying that the option is SCTP-related, it then locks the socket structure while it calls a helper function that performs the necessary steps to do for the socket option in question. The invocation of this helper function is shown in Listing 4.6.

This socket option is read/write, which means that one can also do a *getsockopt()* call and get the current set of default parameters by passing in an empty *struct sctp_sndinfo* as parameter.

The function that takes care of the socket option from there, *sctp_setsockopt_default_-sndinfo()*, is very similar to the old function *sctp_setsockopt_default_send_param()*. The new function, *sctp_setsockopt_default_sndinfo()*, just validates the input by checking that the given structure has the size of a *struct sctp_sndinfo* and that it can be copied from user space successfully. It then tries to find the association identifier given in the *snd_-assoc_id* field of the *struct snd_sndinfo*. If this association identifier is not present on a one-to-many style socket, the function returns an error. The whole function *sctp_-setsockopt_default_sndinfo()* that has been implemented is shown in the Section A.5 in the Appendix.

If the association is found, or if the option is set on a one-to-one style socket, a default *struct sctp_sndinfo* is filled out with the given parameters and stored in the socket structure itself or at the association structure. This default structure is then used

```
...
case SCTP_DEFAULT_SEND_PARAM:
retval = sctp_setsockopt_default_send_param(sk, optval,
   optlen);
break;
case SCTP_DEFAULT_SNDINFO:
    retval = sctp_set_sockopt_default_sndinfo(sk,
                                              optval,
                                              optlen);
    break;
case SCTP_PRIMARY_ADDR:
retval = sctp_setsockopt_primary_addr(sk, optval, optlen);
break;
...
```

Listing 4.6: Excerpt from *sctp_set_sockopt()* in *socket.c*: Showing where SCTP_-DEFAULT_INFO was added to *sctp_setsockopt()*.

by future calls to *sendmsg()*, *sendto()* or *sctp_sendv()*.

An example showcasing the use of this new socket option can be found in Section A.4.

SCTP_RECVRCVINFO

Setting this socket option tells the SCTP stack to include a *struct rcvinfo* when *subsequent* calls to *recvmsg()* is made. This socket option is also read/write, thus it is possible to make a *getsockopt()* call and get information on whether or not this setting is enabled or not.

Figuring out where best to store this setting proved a bit difficult, as the FreeBSD implementation represents sockets quite differently. Settings like these in FreeBSD are stored as a 64 bit integer in the *struct in_pcb* which is compared with masks like SCTP_-PCB_FLAGS_RECVRCVNXTINFO. Since Linux does not have anything like this, it was necessary to see how similar options to SCTP_RECVRCVINFO had been stored before. The SCTP_NODELAY option, which turns on/off the Nagle-like algorithm (Described in section 8.1.5. of RFC 6458 [3]) works a lot like this. It expects a boolean integer to decide its status, and can be toggled on or off with a call to *setsockopt()*. Tracking down how the Linux implementation stored this setting led to the *struct sctp_-sock* in *include/net/sctp/structs.h* that keeps this information stored as an unsigned 8 bit field called *nodelay*. As a result of this discovery, the new options were added to the *struct sctp_sock* as shown in Listing 4.7.

The similarity with SCTP_NODELAY does however stop here, as the functionality of the two are completely different. The closest thing from here would be to look at how the Linux SCTP stack used to know when to send the *struct sctp_sndrcvinfo* as ancillary data. The way this was done prior to RFC 6458, was to set the *sctp_data_io_event* field and call the SCTP_EVENTS socket option as was described in Subsection 2.2.1. So the most logical followup was to check where *sctp_data_io_event* was being processed. In

```
...
__u8 nodelay;
__u8 disable_fragments;
__u8 v4mapped;
__u8 frag_interleave;
__u32 adaptation_ind;
__u32 pd_point;
__u8 recvrcvinfo;
__u8 recvnxtinfo;
...
```

Listing 4.7: Excerpt from *struct sctp_sock* in *structs.h*: The two new boolean options.

```
    ...
if (sp->subscribe.sctp_data_io_event)
    sctp_ulpevent_read_sndrcvinfo(event, msg);
    ...
```

Listing 4.8: Excerpt from *sctp_recvmsg()* in *socket.c*: Checking the deprecated *sctp_-data_io_event* field.

the Linux source this was not too hard to find, as the *sctp_data_io_event* was just read at one single location; In the function *sctp_recvmsg()* of *net/sctp/socket.c*.

As can be seen in Listing 4.8, the next function to handle a *struct sctp_sndrcvinfo* is the function *sctp_ulpevent_read_sndrcvinfo()*. It takes a *struct sctp_ulpevent*[4] (event) and a *struct msghdr* (msg) as parameters. The event is a pointer to the socket buffer's control block[5], which can be used by protocols for storing private per-packet information. In this case, this is the data needed to fill a struct *sctp_rcvinfo*. The function *sctp_ulpevent_-read_sndrcvinfo()* creates a *struct sctp_sndrcvinfo* and fills it with data taken from the event. The msg mentioned in Listing 4.8 is a pointer to the *struct msghdr* the application layer has provided, i.e. where the developer wants the data to go. Finally the filled up *struct sctp_sndrcvinfo* is handled by the kernel function *put_cmsg()* which creates a new *struct cmsghdr* (as was described in Subsection 2.1.2), copies the ancillary data into it and copies it all back to userspace as shown in Listing 4.9.

Since *sctp_ulpevent_read_sndrcvinfo()* was created to handle a *struct sctp_sndrcvinfo*,

[4]A *struct sctp_ulpevent* carries information to the Upper Layer Protocol (ULP). E.g. the sockets API.
[5]This gets a bit complicated, but it is described in more detail at kernel.org [22]

```
...
put_cmsg(msghdr, IPPROTO_SCTP, SCTP_SNDRCV, sizeof(struct
    sctp_sndrcvinfo), (void *)&sinfo);
...
```

Listing 4.9: Excerpt from *sctp_ulpevent_read_sndrcvinfo()* in *ulpevent.c*: The *put_-cmsg* call that copies a *struct sctp_sndrcvinfo* to userspace.

```
    ...
if(sp->recvrcvinfo)
    sctp\_ulpevent\_read\_rcvinfo(event, msg);
    ...
```

Listing 4.10: Excerpt from *sctp_recvmsg()* in *socket.c*: Verifying that the *recvrcvinfo* field is set before proceeding.

```
        ...
        put_cmsg(msghdr, IPPROTO_SCTP, SCTP_RCVINFO,
           sizeof(struct sctp_rcvinfo), (void *)&rinfo);
        ...
```

Listing 4.11: Excerpt from *sctp_ulpevent_read_rcvinfo()* in *ulpevent.c*: *The put_cmsg* call that copies a *struct sctp_rcvinfo* to userspace.

a new function was added to the file *ulpevent.c* called *sctp_ulpevent_read_rcvinfo()*. And to call this, the *__u8 recvrcvinfo* field shown in Listing 4.7 is examined instead of the *sp->subscribe.sctp_data_io_event* field shown in Listing 4.8. Listing 4.10 shows how this is currently being done.

Even though these two functions, *sctp_ulpevent_read_rcvinfo()* and *sctp_ulpevent_-read_sndrcvinfo()*, gets called by slightly different criterias, they are very similar. Instead of a *struct sctp_sndrcvinfo*, *sctp_ulpevent_read_rcvinfo()* creates a *struct sctp_rcvinfo*, and it sets the field *rcv_sid* instead of the deprecated *sinfo_stream*. Finally it puts the ancillary data back in a similar fashion, by calling *put_cmsg()* (Shown in Listing 4.11).

Note that the type parameter to *put_cmsg* has been set to SCTP_RCVINFO in this case, and the size parameter is now based on a *struct sctp_rcvinfo*.

SCTP_RECVNXTINFO

The RECVNXTINFO socket option is handled similarly to the RECVRCVINFO option, but with a slightly different approach to getting the actual message information. While the RECVRCVINFO information was taken from the socket buffer's control block, represented by the *struct sctp_ulpevent*, as was shown in the previous section, the information related to the next message took a bit more effort to extract. The socket buffers are kept in a queue structure[6] on Linux called the *sk_receive_queue*, a field of the *struct sock*[7]. Listing 4.12 shows how this queue is examined in more detail. The *skb_peek()* function takes a look at the next element in the queue without taking it out of the queue, thus leaving it to be returned as RECVRCVINFO if that socket option is also enabled. If the *skb_peek()* returns anything other than NULL, the queue has one element waiting to be received. This socket buffer is then converted to a *struct sctp_ulpevent* with the function *sctp_skb2event()* and then another new function gets called, the *sctp_ulpevent_-*

[6]Finding proper documentation on these queues seems difficult. The best found so far is the comments in the source code of *net/core/skbuff.c* and *include/linux/skbuff.h* [23,24].

[7]The *struct sock* is part of the Sockets API mentioned in Subsection 1.1.1.

```
...
if(sp->recvnxtinfo){
    spin_lock_bh(&sk->sk_receive_queue.lock);
    nxtskb = skb_peek(&sk->sk_receive_queue);
    if (nxtskb)
        atomic_inc(&nxtskb->users);
    spin_unlock_bh(&sk->sk_receive_queue.lock);

    if (nxtskb && nxtskb->len){
        nxt_event = sctp_skb2event(nxtskb);
        sctp_ulpevent_read_nxtinfo(nxt_event, msg, nxtskb);
    }
    /* If there is no nxtskb, just continue as if nothing
       happened. */
}
...
```

Listing 4.12: Excerpt from *sctp_recvmsg()* in socket.c: Checking if a next message is present on the queue.

read_nxtinfo(). Again, this function does the same as the previous two (*sctp_ulpevent_-read_{sndrcv|rcv}info*). The only difference here being that this one needs to provide the *nxt_length* field. The *nxt_length* is the length of the message currently within the socket buffer. According to section 5.3.6. of RFC 6458 [3], this does not necessarily mean the entire length of the message, as the next message might be part of a partial delivery. Only if the SCTP_COMPLETE (See Subsection 4.4.1) flag is set does this field represent the size of the entire next message.

4.3 Adding new functions

When the kernel module for SCTP is first loaded by the operating system, it will first look for the *module_init()* macro. This in turn, directs the operating system to the function *sctp_init()* defined in protocol.c. *Sctp_init()* initializes the whole SCTP "universe". Notably, this function sets up a storage area for future associations, endpoints and ports and then registers this universe with the operating system with a call to the function *proto_register()*. This function takes a structure of type *struct proto* as a parameter.

The *struct proto* for SCTP is shown in Listing 4.13. It tells the operating system where to find SCTP specific function for a given system call. It is the interface between the transport layer and the application layer. So if an application invokes any of the functions shown in the left column of Listing 4.13 the corresponding SCTP-specific function will be called in its place. Adding new functions to the userspace library does not require any modifications to the struct proto, as the struct proto is strictly for directing system calls to the proper SCTP-specific functions. For instance, the new *sctp_recvv()* and *sctp_sendv()* are implemented as userspace functions, as wrap-

```
struct proto sctp_prot = {
    .name         =  "SCTP",
    .owner        =  THIS_MODULE,
    .close        =  sctp_close,
    .connect      =  sctp_connect,
    .disconnect   =  sctp_disconnect,
    .accept       =  sctp_accept,
    .ioctl        =  sctp_ioctl,
    .init         =  sctp_init_sock,
    .destroy      =  sctp_destroy_sock,
    .shutdown     =  sctp_shutdown,
    .setsockopt   =  sctp_setsockopt,
    .getsockopt   =  sctp_getsockopt,
    .sendmsg      =  sctp_sendmsg,
    .recvmsg      =  sctp_recvmsg,
    .bind         =  sctp_bind,
    .backlog_rcv  =  sctp_backlog_rcv,
    .hash         =  sctp_hash,
    .unhash       =  sctp_unhash,
    .get_port     =  sctp_get_port,
    .obj_size     =  sizeof(struct sctp_sock),
    .sysctl_mem   =  sysctl_sctp_mem,
    .sysctl_rmem =  sysctl_sctp_rmem,
    .sysctl_wmem =  sysctl_sctp_wmem,
    .memory_pressure = &sctp_memory_pressure,
    .enter_memory_pressure = sctp_enter_memory_pressure,
    .memory_allocated = &sctp_memory_allocated,
    .sockets_allocated = &sctp_sockets_allocated,
};
```

Listing 4.13: The *struct proto* for SCTP (IPv4)

```
#define SCTP_SENDV_NOINFO    0
#define SCTP_SENDV_SNDINFO   1
#define SCTP_SENDV_PRINFO    2
#define SCTP_SENDV_AUTHINFO  3
#define SCTP_SENDV_SPA       4
```

Listing 4.14: Constants used by *sctp_sendv()*.

```
#define SCTP_RECVV_NOINFO  0
#define SCTP_RECVV_RCVINFO 1
#define SCTP_RECVV_NXTINFO 2
#define SCTP_RECVV_RN      3
```

Listing 4.15: Constants used by *sctp_recvv()*.

pers around *sendmsg()* and *recvmsg()*. And thanks to this setup, calls to *sendmsg()* or *recvmsg()* get redirected to their respective SCTP-specific counterparts *sctp_sendmsg()* and *sctp_recvmsg()* functions that are defined in *socket.c*.

4.3.1 Implementation of sctp_sendv() and sctp_recvv()

By studying the code of FreeBSD, it was apparent that some of the work on the user-space side was already done. Hence, the functions *sctp_sendv()* and *sctp_recvv()* were initially copied from the file */usr/src/lib/libc/sctp_sys_calls.c* in FreeBSD to their own, separate files in the *lib*-directory of the Linux userspace library. This is where all the other userspace functions were defined already, so it seemed appropriate to add the two new functions here as well. Both functions did, however, have many references to things that were not currently present in the Linux implementation. The definitions shown in Listing 4.14 and 4.15 were found in the file *sys/netinet/sctp_uio.h* in FreeBSD.

As none of these definitions existed in Linux yet, they were added to the file *include/netinet/sctp.h* in *libsctp*, and to *include/uapi/linux/sctp.h* for the kernel[8].

In addition, a few build-oriented changes had to be made to add the new files. After the new files had been put in the *lib*-directory, their names had to be added to the *lib/Versions.map* file, which already contained the names of all the old userspace functions. Note that this particular file, *lib/Versions.map*, is specific to *libsctp*, and that this is not meant to be a general recipe on how to add new files to other software projects. The filenames of the new files were then added to the *lib/Makefile.am*, which is part of the GNU Automake [26] software package.

Finally, the prototypes of the new functions were added to *include/netinet/sctp.h* as shown in Listing 4.16.

After the prototypes and the necessary constants had been added for both files, a quick cleanup of the #include-directives at the top of each file was all that was necessary. The FreeBSD versions of the new functions came from a file that contained numerous other functions, so a lot of the included files were unnecessary for the Linux version.

The only thing that had to be changed within the functions themselves, when including the FreeBSD code in *libsctp*, was a check to see if the given *struct sockaddr *addrs*-field contained addresses that were not of equal size as a *struct sockaddr_in*. The

[8]These two files are closely related, but since version 3.5 of the Linux kernel they have been split into kernelspace and userspace versions. A discussion of this subject has been written by Michael Kerrisk and can be read on Linux Weekly News [25].

```
...
/* RFC 6458 - Section 9.12*/
int sctp_sendv(int sd,
        const struct iovec *iov,
        int iovlen,
        struct sockaddr *addrs,
        int addrcnt,
        void *info,
        socklen_t infolen,
        unsigned int infotype,
        int flags);

/* RFC 6458 - Section 9.13*/
int sctp_recvv(int sd,
        const struct iovec *iov,
        int iovlen,
        struct sockaddr *from,
        socklen_t *fromlen,
        void *info,
        socklen_t *infolen,
        unsigned int *infotype,
        int *flags);
...
```

Listing 4.16: Excerpt from *include/netinet/sctp.h*: Prototypes for the new functions

struct sockaddr_in does not have the field *sin_len* in Linux, so this might have to be checked in a different way, but that has not been considered further in this thesis.

4.4 New struct sctp_sndinfo flags

The *sctp_sendv()*-function can include ancillary data in the form of a *struct sctp_sndinfo*. Listing 4.17 shows the three new flags added to *include/uapi/linux/sctp.h*. The new flags were added in the same order as the already existing ones. The SCTP_EOF is special, it is defined as MSG_FIN which in turn is set to 0x200 (or 512) in Linux. There should be enough room to put a few more flags here until this causes any trouble. Note that no new flags for the *struct sctp_rcvinfo* were introduced in RFC 6458.

```
enum sctp_sinfo_flags {
    SCTP_UNORDERED = 1,  /* Send/rcv message unordered. */
    SCTP_ADDR_OVER = 2,  /* Override primary dest. */
    SCTP_ABORT=4,        /* Send ABORT message to peer. */
    SCTP_SACK_IMMEDIATELY = 8,  /* SACK without delay */
    SCTP_EOF=MSG_FIN,    /* Initiate graceful shutdown. */
    SCTP_SENDALL = 16,
    SCTP_COMPLETE = 32,
};
```

Listing 4.17: New *sinfo_flags*.

Details regarding SCTP_COMPLETE and SCTP_SENDALL will be given in the following sections.

4.4.1 Implementation of the SCTP_COMPLETE flag

This flag helps determine whether a message is complete or not. It has been implemented in the function *sctp_ulpevent_make_rcvmsg()* in the file *ulpevent.c*. According to Stewart and Xie (Section 3.2.5 "The DATA chunk" [19]), a message is complete if both the B (beginning fragment bit) and E (ending fragment bit) bits are set in the chunk's header. The implementation of SCTP_COMPLETE is shown Listing 4.18. It makes sure that the chunk header has both the E and B bits set, and then sets the SCTP_COMPLETE flag on the given event. This is the same type of event that was used to fill the *struct sctp_rcvinfo* and *struct sctp_nxtinfo* earlier. Note that the U, B and E-bits were also mentioned in Section 2.6.2.

```
...
if (chunk->chunk_hdr->flags & SCTP_DATA_FIRST_FRAG &&
    chunk->chunk_hdr->flags & SCTP_DATA_LAST_FRAG) {
        event->flags |= SCTP_COMPLETE;
}
...
```

Listing 4.18: Excerpt from *sctp_ulpevent_make_rcvmsg()* in *ulpevent.c*: Implementation of the SCTP_COMPLETE flag.

4.4.2 Implementation of the SCTP_SENDALL flag

Calling *sctp_sendv()* with the field *snd_flags* set to SCTP_SENDALL indicates that the caller wants that particular message to be sent to all associations currently established on a socket. The SCTP_SENDALL flag is meant to be used by one-to-many style sockets, since a one-to-one style socket will only have at most one association, and will thus essentially just ignore this flag. In Section 1.7 it was listed that a solution to this problem had been implemented, and this section will describe that solution. Note that this implementation has its drawbacks, as will be explained in Section 5.2.

A few more changes to *sctp_sendmsg()* was made than those that were shown in Section 4.1. If the given *struct sctp_sndinfo* has the SCTP_SENDALL flag set, it will be turned off, and put back onto the message msg as shown in Listing 4.19. A check to see if the socket is of one-to-many style is performed first. Finally, the boolean *sendall* is set to 1, just as a reminder that this needs to be handled later on. At least one association has to be created or found prior to handling the rest.

```
if(sinfo_flags & SCTP_SENDALL && sctp_style(sk, UDP)){
    sinfo->sinfo_flags &= ~SCTP_SENDALL;
    put_cmsg(msg, IPPROTO_SCTP, SCTP_SNDRCV,
            sizeof(struct sctp_sndrcvinfo), (void *)sinfo);
    sendall = 1;
}
```

Listing 4.19: Preparing for a SCTP_SENDALL

A simpler version of *sctp_sendmsg()* was created which skips the whole association lookup/creation part and just sends the message to the given association. Its prototype looks like this:

```
static int sctp_sendmsg_to_association(
        struct kiocb *iocb,
        struct sock *sk,
        struct msghdr *msg,
        size_t msg_len,
        struct sctp_association* asoc);
```

Listing 4.20: Prototype of *sctp_sendmsg_to_association()*.

This special function gets called from *sendmsg()* after looping through each association established on the socket's endpoint. How this is done is shown in Listing 4.21.

```
if(sendall){
    tmp = NULL;

    list_for_each_entry(tmp, &sctp_sk(sk)->ep->asocs, asocs){
        /* The socket is locked in sctp_sendmsg_to_association(), so it must be released for
           each iteration. If something fails, sctp_sendmsg_to_association() will release it
           itself. */
        sctp_release_sock(sk);
        err = sctp_sendmsg_to_association(iocb, sk, msg, msg_len, tmp);
        if(err <= 0){
            goto out_free;
        }
    }
    goto out_unlock;
}
```

Listing 4.21: Calling *sctp_sendmsg_to_association()*.

An example of how the SCTP_SENDALL-flag might be used has been included in Section A.3 in the Appendix.

4.5 Summary

This chapter has described in detail the attempts at implementing some of the missing functionality mentioned in Chapter 3. Again, the solution to SCTP_SENDALL has its weakness, thus the following chapter will highlight some ways to improve upon the SCTP_SENDALL implementation, as well as provide a discussion and a retrospective view on other points that have been brought up in this chapter.

Chapter 5

Evaluation and Discussion

This chapter will first present a retrospective view of what was presented in the previous chapter, and also try to provide a short evaluation of the implementations that were shown. The challenges with the SCTP_SENDALL implementation is given in Section 5.2. Furthermore, Section 5.3 will review the given feedback from the developers on the LKSCTP-mailing list and show some of the suggested improvements. Finally, an overview of possible improvements for future work is shown in Section 5.4.

5.1 Retrospect

In retrospect, it would be more useful to look at bigger changes to SCTP than most of the subjects that were discussed in the previous chapter. The main new feature that has been contributed in this thesis is the support for *sctp_nxtinfo*. Since the deprecated *struct sctp_extrcvinfo* system was never implemented for Linux, it has not been possible for a developer to retrieve ancillary data about the next message from the SCTP stack before. Apart from that, the implemented changes are more of an evolutionary nature to bring the Linux implementation of SCTP a little closer to being compliant with RFC 6458. The old *struct sctp_sndrcvinfo* with its accompanying SCTP_SNDRCV control message type worked the same way as the *struct sctp_sndinfo* and *struct sctp_rcvinfo* does now. The new *sctp_recvv()* and *sctp_sendv()* functions are helper functions for doing something that was already possible prior to RFC 6458, but it required more effort.

The main part of the conducted research has without question been identifying missing pieces in the puzzle. RFC 6458 does, for the most part, not say explicitly what has changed or not. Thus, several aspects had to be examined very closely to figure out whether they had discrepancies or not. Small things like the SCTP_FRAGMENT_INTERLEAVE discrepancy described in Subsection 3.4.2 was particularly difficult to catch.

Getting the SCTP_SENDALL flag to work properly would have been a good achievement, but unfortunately time constraints and the general complexity of the problem as a whole was too much to handle in the available time frame. Also, although setting the SCTP_COMPLETE flag might be useful when checking whether the next incoming message is fragmented or not, it is not really useful on a user level. The SCTP stack will continue to deliver complete messages regardless of the fact that the flag is set or not.

It has been necessary to make some difficult decisions along the way, such as how to handle the new *struct sctp_sndinfo* in socket.c. Deciding to reuse the *default_sinfo* struct that was already being used for other things, is not necessarily the most elegant solution to the problem.

5.2 Problems with the SCTP_SENDALL solution

In Section 1.7, "Main contributions", the solution to SCTP_SENDALL was described as a simple draft solution. The reason for classifying it as such, is primarily that it does not factor in scalability at all. Michael Tüxen says (email exchange, 26. apr 2014) *"What if the socket has 100000 associations? ... Now you run a huge loop in the kernel. This might result in the system becoming unresponsive as long as the loop runs."* While testing the implementation done in this thesis, having such numbers of associations running simultaneously was not an option, so the solution outlined in the previous chapter did, however, work for a small number of concurrent associations.

According to Tüxen, the developers of SCTP for FreeBSD utilized a separate kernel thread called the iterator thread for this purpose. So that the main kernel thread would keep running even if a large number of simultaneous send operations were scheduled at the same time due to a SCTP_SENDALL flag.

Finally, the reuse of pretty much the entire *sctp_sendmsg()* function in *sctp_sendmsg_-to_association()* is not particularly elegant. It should be possible to implement this in a better way, making use of the Linux kernel thread system found in *linux/kthread.h*[1].

5.3 Submitting patches to LKSCTP

Four patches were submitted for review to the Linux kernel developers as part of the work with this thesis. The following features were submitted and are currently up for discussion:

1. New structures *sctp_sndinfo* and *sctp_rcvinfo*.
2. New structure *sctp_nxtinfo* and the socket option SCTP_RECVNXTINFO
3. New socket option SCTP_DEFAULT_SNDINFO to set a default *struct sndinfo*.
4. New functions *sctp_recvv()* and *sctp_sendv()*.

Note that the implementation of SCTP_COMPLETE has not been submitted. The specification is still a bit vague on what the criterias for setting it is, and it is still not certain that the implementation is correct. A developer working on SCTP for Linux

[1]Kernel threads are similar to user threads, but they only operate in kernel space. It was unfortunately difficult to find good documentation on this, but the source code for kthread.h can be found at https://github.com/torvalds/linux/blob/master/include/linux/kthread.h

```
...
if(sp->recvnxtinfo){
    nxtskb = sctp_skb_recv_datagram(sk, MSG_PEEK, noblock,
       &err2);

    if (nxtskb && nxtskb->len){
        nxt_event = sctp_skb2event(nxtskb);
        sctp_ulpevent_read_nxtinfo(nxt_event, msg, nxtskb);
    }
}
...
```

Listing 5.1: Excerpt from *sctp_recvmsg()* in socket.c: Checking if a next message is present on the queue.

```
...
out:
    if(nxtskb)
        kfree_skb(nxtskb);
    sctp_release_sock(sk);
...
```

Listing 5.2: Freeing up allocated space properly.

has said that SCTP_COMPLETE probably has to be done in relation to a reassembly routine elsewhere.

The initial review given by the people at the LKSCTP mailing list revealed a couple of mistakes regarding what was presented in the previous chapter. The following sections will quickly run what these mistakes were, and how they have been resolved.

5.3.1 Submitting the sctp_nxtinfo implementation

The way sctp_nxtinfo was handled had a few mistakes. The feedback given showed that using the function *sctp_skb_recv_datagram()* with the proper MSG_PEEK flags would be cleaner than what was being done. The new version can be seen in Listing 5.1. It can be compared to the previous version in Listing 4.12.

In addition to peeking at the next message in a less cluttered fashion, some code was added to properly free the *nxtskb* if space had been allocated for it. Listing 5.2 shows how this was done at the very end of the *sctp_recvmsg()*-function in *socket.c*.

5.3.2 Submitting the sctp_sndinfo implementation

A few changes has also been done to the way control messages is handled. The feedback indicated that using a C union was cleaner than using the *void* pointer approach as was shown in Listing 4.3. Thus, the new version of struct sctp_cmsgs_t looks like what is shown in Listing 5.3.

```
/* A convenience structure to parse out SCTP specific CMSGs
     */
typedef struct sctp_cmsgs {
    struct sctp_initmsg *init;
    union{
        struct sctp_sndrcvinfo *srinfo;
        struct sctp_sndinfo *sinfo;
    } info;
    sctp_cmsg_t cmsg_type;
} sctp_cmsgs_t;

#define sr_info info.srinfo
#define s_info info.sinfo
```

Listing 5.3: The new *struct sctp_cmsgs_t* in *structs.h*.

The changes to *struct sctp_cmsgs_t* induced a few changes to how *sctp_msghdr_-parse()* extracts the control messages off of a *struct msghdr* too, as shown in Listing 5.4. The original version is shown in Listing 4.4.

Using a union in *sctp_cmsgs_t* also allowed for a cleaner way of setting the *default_-sinfo* struct in the function *sctp_recvmsg()*. Listing 5.5 shows the new way, compared to the earlier version shown in Listing 4.5.

5.3.3 Submitting the new functions sctp_sendv()- and sctp_recvv()

According to a kernel developer, since the functions were not implemented as system calls in FreeBSD, but as userspace functions inclusion in the Linux implementation of SCTP should not be a problem. However, as mentioned in Section 4.3, Linux seems to have never implemented a *sin_len* member in the sockaddr structure. The fact that BSD is trying to validate the address that is passed in makes sense. This can currently not be done the same way in Linux. One could make assumptions based on the length of the *sa_family*[2] field, but as this is not completely the same the whole inclusion is on hold for now. A discussion on whether this could lead to a problem or not is in progress, and there is thus not much more to be said at this point.

As a result, the structures *struct sctp_sendv_spa* and *struct sctp_recvv_rn* has not been submitted at this point.

[2]A field specifying if the address is of the IPv4 or IPv6 family among others [27].

```
...
case SCTP_SNDINFO:
    /* SCTP Socket API Extension
     * 5.3.4 SCTP Send Information Structure (SCTP_SNDINFO)
     *
     * This cmsghdr structure specifies SCTP options for
     * sendmsg(). This structure and SCTP_RCVINFO replaces
     * SCTP_SNDRCV which has been depleted.
     *
     * cmsg_level    cmsg_type      cmsg_data[]
     * ------------  ------------   ---------------------
     * IPPROTO_SCTP  SCTP_SNDINFO    struct sctp_sndinfo
     * */
    if(cmsg->cmsg_len !=
            CMSG_LEN(sizeof(struct sctp_sndinfo))){
        return -EINVAL;
    }

    cmsgs->info.sinfo = (struct sctp_sndinfo
       *)CMSG_DATA(cmsg);
    cmsgs->cmsg_type = SCTP_SNDINFO;

    if (cmsgs->s_info->snd_flags &
            ~(SCTP_UNORDERED | SCTP_ADDR_OVER |
                SCTP_ABORT | SCTP_EOF | SCTP_SENDALL)){
        return -EINVAL;
    }
    break;
...
```

Listing 5.4: Excerpt from *sctp_msghdr_parse()*: Adding support for *SCTP_-SNDINFO*

```
    ...
    if(cmsgs.cmsg_type == SCTP_SNDINFO){
        /* Put the cmsg data into a temporary struct sctp_sndinfo and
           move it into the struct sctp_sndrcvinfo default_sinfo; */
        memset(&default_sinfo, 0, sizeof(default_sinfo));

        default_sinfo.sinfo_flags = cmsgs.s_info->snd_flags;
        default_sinfo.sinfo_stream = cmsgs.s_info->snd_sid;
        default_sinfo.sinfo_assoc_id = cmsgs.s_info->snd_assoc_id;
        default_sinfo.sinfo_ppid = cmsgs.s_info->snd_ppid;
        default_sinfo.sinfo_context = cmsgs.s_info->snd_context;
        sinfo = &default_sinfo;
    }else{
        /* cmsgs.info could be NULL, but will be replaced by a default sinfo later,
           since sndinfo has not been set. */
        sinfo = cmsgs.sr_info;
    }
...
```

Listing 5.5: Excerpt from *sctp_sendmsg()*: Handling a sendmsg() with struct_-sndinfo

5.4 Future work

The main topics that could be improved upon, but that have not been covered in this thesis are:

1. The SCTP_EVENTS mechanism is used by many SCTP applications, so it would be beneficial to get the new SCTP_EVENT system working to replace it. It is quite possible that implementing the new SCTP_EVENT mentioned in Section 3.3 might be as simple as just storing all options as their own fields on either the socket or association itself (or both), similarly to what was done with the SCTP_-RECVRCVINFO and SCTP_RECVNXTINFO shown in Section 4.2. The most straightforward approach would be for the SCTP stack to take in the notification interest messages with the new SCTP_EVENT socket option, but then store these the same way that has been done earlier, i.e. in a *struct sctp_event_subscribe*. But then no real progress would have been done. The same scalability issues would still be present. Figuring out the best way to store which notifications a peer is interested in receiving is a key issue with this mechanism.

2. The SCTP_SEND_FAILED_EVENT described briefly in Section 3.3 looks rather simple to do. But as SCTP_EVENT is not implemented it would not make too much sense to focus on a small detail like this.

3. Being able to set socket options for currently established-, future- or all sockets will be a big part of getting the Linux SCTP implementation closer to RFC 6458 compliancy. These options can be used in place of association identifiers in multiple places throughout the API defined in RFC 6458. Subsection 3.4.1 described these options. Unfortunately, implementing support for these options, would entail a rather complex system to store the settings on both the endpoint level and at the association level. There would also be necessary to come up with a system for inheritance values from endpoints or associations higher up in the hierarchy, to implement things like SCTP_FUTURE_ASSOC. Implementing this change in this thesis was not possible mainly due to its complexity.

4. Everything related to the Partially Reliability SCTP [21] extension and the AUTH [18] extension seem to be of pretty high focus in RFC 6458. These extensions were, however, not covered in great detail in this thesis since the whole concept of these seem to span over several RFCs' and be generally slightly too complex and out of scope of the problem. The AUTH extension's intended functionality seems to be partially implemented, but the Partially Reliability SCTP extension seems to be completely left out of the current Linux source code. Further proposals regarding future work on these extensions are therefore not covered here.

5.5 Summary

This chapter has made an attempt at providing a more critical look at the work performed in this thesis. The individual features that were developed have been discussed and an evaluation of both the submitted features and the reasoning behind leaving some of them out have been given.

Chapter 6

Conclusion

This thesis has outlined the necessary steps needed to get the SCTP API up to date with the new standard defined in RFC 6458. The outcome of the contributed patches is still not decided, but will hopefully be accepted after another round of review by the LKSCTP developers. However, even with the contributions presented here, getting the SCTP API up to date will continue to be a work in progress, but the work and research conducted in this thesis will take the implementation one step closer to finalization. Of particular significance in respect to getting the implementation done, is the identification of which features that need more work.

6.1 Summary

The first chapter has established the universe this thesis revolves around, SCTP, protocols and general capacities of SCTP compared to its closest sibling protocol, TCP. Then, chapter 2 explained the state of the SCTP implementation on Linux today. To address the problem statement in Section 1.5 a full review of the discovered discrepancies between the current Linux SCTP implementation and RFC 6458 was given in Chapter 3. Two out-of-date functions, and many other features were found to be missing or needed change to comply to the new API standard. A large portion of these discoveries were changes to the way ancillary data are to be handled. The process of implementing of some of the changes has been described in detail in Chapter 4.

A final summary of the changes that have been implemented as part of this thesis is given here:

- New socket option: SCTP_RECVV_NXTINFO.
- New socket option: SCTP_RECVV_RCVINFO.
- New socket option: SCTP_DEFAULT_SNDINFO.
- New struct: *struct sctp_nxtinfo*.
- New struct: *struct sctp_rcvinfo*.
- New struct: *struct sctp_sndinfo*.

- New snd_flag: SCTP_COMPLETE.
- Developed draft solution to SCTP_SENDALL flag.
- New function: *sctp_recvv()*.
- New function: *sctp_sendv()*.

Finally, Chapter 5 gave a more critical review of the accomplishments in Chapter 4, but also described the process of submitting patches to the LKSCTP mailing list, and the handling of the given feedback. The submitted code that await approval are as of now the socket options and the structs, but the functions are likely to be added in the near future.

Glossary

ABI Application Binary Interface.

API Application Programming Interface.

FreeBSD A free operating system descended from AT&T Unix.

Head-of-line blocking When a message M1 has to wait to be transmitted since a failed message M2 is blocking the transmission path of M1..

IEEE Institute of Electrical and Electronics Engineering.

IETF Internet Engineering Task Force.

Linux Linux is the worlds most popular non-proprietary operating system..

LKSCTP Linux Kernel SCTP.

Operating System Computer software that works as a layer between user applications and the computer hardware..

POSIX Portable Operating System Interface - A family of standards specified by IEEE for maintaining compatibility between operating systems..

RFC Request For Comments - A document describing a technical issue.

SCTP Stream Control Transmission Protocol.

Sockets API An API used for communication between the transport layer and the upper layer (the application layer)..

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

Bibliography

[1] R.R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, and M. Kalla. *Stream Control Transmission Protocol*. RFC 2960, Internet Engineering Task Force, October 2000.

[2] The Linux Foundation. *About Linux Kerne*l. June 2013. https://www.kernel.org/linux.html.

[3] R.R. Stewart, M. Tüxen, K. Poon, P. Lei, and V. Yasevich. *Socket API Extensions for the Stream Control Transmission Protocol (SCTP)*. Request for Comments 6458, Internet Engineering Task Force, December 2011.

[4] J. B. Postel. *Transmission Control Protocol*. RFC 793, Internet Engineering Task Force, September 1981.

[5] J. B. Postel. *Internet Protocol*. RFC 791, Internet Engineering Task Force, September 1981.

[6] J. B. Postel. *User Datagram Protocol*. RFC 768, Internet Engineering Task Force, August 1980.

[7] R. Mandeville and J. Perser. *Benchmarking Methodology for LAN Switching Devices*. RFC 2889, Internet Engineering Task Force, August 2000.

[8] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. Request for Comments 4987, Internet Engineering Task Force, August 2007.

[9] *Linux Kernel Stream Control Transmission Protocol Tools*. http://lksctp.sourceforge.net/.

[10] The FreeBSD Project. *FreeBSD - The Power To Serve*. https://www.freebsd.org/.

[11] *Signaling Transport Working Group*. http://datatracker.ietf.org/wg/sigtran/.

[12] *Transport Area Working Group*. http://datatracker.ietf.org/wg/tsvwg/charter/.

[13] La Monte H.P. Yarroll and Karl Knutson. *Linux Kernel SCTP: The third transport*. 2001. http://old.lwn.net/2001/features/OLS/pdf/pdf/sctp.pdf.

[14] R.R. Stewart. *Stream Control Transmission Protocol*. Request for Comments 4987, Internet Engineering Task Force, September 2007.

[15] GNU grep. http://www.gnu.org/software/grep/.

[16] Linux Programmer's Manual CMSG(3). http://man7.org/linux/man-pages/man3/cmsg.3.html.

[17] R.R. Stewart, Q. Xie, M. Tüxen, S. Maruyama, and M. Kozuka. *Stream Transmission Protocol (SCTP) Dynamic Address Reconfiguration*. Request for Comments 5061, Internet Engineering Task Force, September 2007.

[18] M. Tüxen, R.R. Stewart, P. Lei, and E. Rescorla. *Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)*. Request for Comments 4895, Internet Engineering Task Force, August 2007.

[19] R.R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley Professional, 2001.

[20] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104, Internet Engineering Task Force, February 1997.

[21] R.R. Stewart, M. Ramalho, Q. Xie, M. Tüxen, Univ. of Applied Sciences Muenster, and P. Conrad. *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*. RFC 3758, Internet Engineering Task Force, May 2004.

[22] *How SKBs work*. http://vger.kernel.org/~davem/skb.html.

[23] kernel/git/torvalds/linux.git - linux kernel source tree (skbuff.c). http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/net/core/skbuff.c.

[24] kernel/git/torvalds/linux.git - linux kernel source tree (skbuff.h). http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/linux/skbuff.h.

[25] The uapi header file split. http://lwn.net/Articles/507794/.

[26] Automake - gnu project - free software foundation (fsf). http://www.gnu.org/software/automake/.

[27] Address family numbers. http://www.iana.org/assignments/address-family-numbers/address-famil

Appendix A

Where to get the code

This appendix contains some code examples that was too big to fit within the actual document. All code that has been developed during this project can be viewed at the following git repositories:

- https://bitbucket.org/geirola/linux-sctp for the changes to the kernel module.
- https://bitbucket.org/geirola/lksctp for the changes to the userspace library.

A.1 Code example 1

```
        /* Initialize structs */
        struct msghdr outmsg;
        memset(&outmsg, 0, sizeof(outmsg));
        struct iovec iov;
        memset(&iov, 0, sizeof(iov));
        char outcmsg[CMSG_SPACE(sizeof(struct sctp_sndrcvinfo))];
        struct cmsghdr *cmsg;
        struct sctp_sndrcvinfo *sinfo;
        char buf[SIZE];
        memset(buf, 0, SIZE);

        strcpy(buf, "Hello, World!");

        outmsg.msg_iov = &iov;
        iov.iov_base = (void *)buf;
        iov.iov_len = SIZE;
        outmsg.msg_iovlen = 1;

        outmsg.msg_control = outcmsg;
        outmsg.msg_controllen = sizeof(outcmsg);
        outmsg.msg_flags = 0;

        cmsg = CMSG_FIRSTHDR(&outmsg);
        cmsg->cmsg_level = IPPROTO_SCTP;
        cmsg->cmsg_type = SCTP_SNDRCV;
        cmsg->cmsg_len = CMSG_LEN(sizeof(struct
            sctp_sndrcvinfo));

        outmsg.msg_controllen = cmsg->cmsg_len;
        sinfo = (struct sctp_sndrcvinfo *)CMSG_DATA(cmsg);
        memset(sinfo, 0, sizeof(struct sctp_sndrcvinfo));
        sinfo->sinfo_stream = 5;

        sendmsg(sockfd, &outmsg, 0);
```

Listing A.1: Example: Sending a message with ancillary data using *sendmsg()*.

A.2 Code example 2

```c
struct sctp_event_subscribe event;
memset(&event, 0, sizeof(struct sctp_event_subscribe));
event.sctp_data_io_event = 1;
setsockopt(client_sockfd,IPPROTO_SCTP, SCTP_EVENTS,
   &event, sizeof(event));

struct sctp_sndrcvinfo sndrcv;
struct msghdr msg;
struct cmsghdr cmsg;
struct iovec iov;
memset(&msg, 0, sizeof(msg));
memset(&cmsg, 0, sizeof(cmsg));
memset(&sndrcv, 0, sizeof(sndrcv));
memset(&iov, 0, sizeof(iov));

char cmsgbuf[CMSG_SPACE(sizeof(struct sctp_sndrcvinfo))];
char buf[SIZE];
memset(buf, 0, SIZE);

iov.iov_base = buf;
iov.iov_len = SIZE;

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
msg.msg_control = cmsgbuf;
msg.msg_controllen = sizeof(cmsgbuf);

recvmsg(client_sockfd, &msg, 0);

struct cmsghdr *cmsgptr;
for(cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
   cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)){
    if(IPPROTO_SCTP == cmsgptr->cmsg_level && SCTP_SNDRCV
        == cmsgptr->cmsg_type)
        break;
}
if(cmsgptr)
    memcpy(&sndrcv, CMSG_DATA(cmsgptr), sizeof(struct
        sctp_sndrcvinfo));
printf("Received sndrcv.sinfo_stream = %d\n",
   sndrcv.sinfo_stream);
```

Listing A.2: Example: Receiving a message with ancillary data using *recvmsg()*.

A.3 Code example 3

```c
struct iovec iov;
struct msghdr msg;
char buf[SIZE];
struct sctp_sndinfo snd;

memset(&snd, 0, sizeof(snd));
memset(&iov, 0, sizeof(iov));
memset(&buf, 0, sizeof(buf));
memset(&msg, 0, sizeof(msg));

snd.snd_flags = SCTP_SENDALL;

iov.iov_len = sizeof(buf);
iov.iov_base = buf;

strnpy(buf, "Message to all");
iov.iov_len = strlen(buf);
sctp_sendv(sockfd,
           msg.msg_iov,
           1, /* no. iovs */
           (struct sockaddr*) &to, /* Dest addr */
           1, /* no. addrs */
           &snd,
           sizeof(snd),
           SCTP_SENDV_SNDINFO,
           0);
```

Listing A.3: Example: Sending a message to all associations with SCTP_SENDALL and *sctp_sendv()*.

A.4 Code example 4

```
struct sctp_sndinfo snd;
memset(&snd, 0, sizeof(snd));
snd.snd_sid = 4;

setsockopt(sockfd, IPPROTO_SCTP, SCTP_DEFAULT_SNDINFO,
   &snd, sizeof(snd));

iov.iov_base = buf;
strcpy(buf, "Message");
iov.iov_len = strlen(buf);

msg.msg_iov = &iov;
msg.msg_iovlen = 1;

sendmsg(sockfd, &msg, 0);
```

Listing A.4: Example: Setting a default *struct sctp_sndinfo* with SCTP_DEFAULT_-SNDINFO.

A.5 Code example 5

```
static int sctp_setsockopt_default_sndinfo(struct sock *sk,
                                        char __user *optval,
                                        unsigned int optlen){
    struct sctp_sndinfo info;
    struct sctp_association *asoc;
    struct sctp_sock *sp = sctp_sk(sk);

     if (optlen != sizeof(struct sctp_sndinfo))
             return -EINVAL;
     if (copy_from_user(&info, optval, optlen))
             return -EFAULT;

     asoc = sctp_id2assoc(sk, info.snd_assoc_id);
     if (!asoc && info.snd_assoc_id && sctp_style(sk, UDP))
             return -EINVAL;

     if (asoc) {
             asoc->default_stream = info.snd_sid;
             asoc->default_flags = info.snd_flags;
             asoc->default_ppid = info.snd_ppid;
             asoc->default_context = info.snd_context;
             /* Note! asoc->default_timetolive is not set in this way anymore.
              * The PR-SCTP extension needs to be implemented.  */
     } else {
             sp->default_stream = info.snd_sid;
             sp->default_flags = info.snd_flags;
             sp->default_ppid = info.snd_ppid;
             sp->default_context = info.snd_context;
             /* Note! sp->default_timetolive is not set in this way anymore.
              * The PR-SCTP extension needs to be implemented.  */
     }
     return 0;
}
```

Listing A.5: The new *sctp_setsockopt_default_sndinfo()* function.