# Socket API for the SCTP User-land Implementation (usrsctp)

I. Rüngeler[*] M. Tüxen[†]

December 5, 2017

## Contents

## 1 Introduction

In this manual the socket API for the SCTP User-land implementation will be described. It is based on RFC 6458 [?]. The main focus of this document is on pointing out the differences to the SCTP Sockets API. For all aspects of the sockets API that are not mentioned in this document, please refer to RFC 6458. Questions about SCTP can hopefully be answered by RFC 4960 [?].

## 2 Getting Started

The User-land stack has been tested on FreeBSD 10.0, Ubuntu 11.10, Windows 7, Mac OS X 10.6, and MAC OS X 10.7. The current version of the User-land stack is provided on *http://sctp.fh-muenster.de/sctp-user-land-stack.html*. Download the tarball and untar it in a folder of your choice. The tarball contains all the sources to build the libusrsctp, which has to be linked to the object file of an example program. In addition there are two applications in the folder *programs* that can be built and run.

### 2.1 Building the Library and the Applications

#### 2.1.1 Unix-like Operating Systems

In the folder *usrsctp* type

```
./bootstrap
./configure
make
```

[*]Münster University of Applied Sciences, Department of Electrical Engineering and Computer Science, Stegerwaldstr. 39, D-48565 Steinfurt, Germany, `i.ruengeler@fh-muenster.de`.

[†]Münster University of Applied Sciences, Department of Electrical Engineering and Computer Science, Stegerwaldstr. 39, D-48565 Steinfurt, Germany, `tuexen@fh-muenster.de`.

Now, the library *libusrsctp.la* has been built in the subdirectory *usrsctplib*, and the example programs are ready to run from the subdirectory *programs*.

If you have root privileges or are in the sudoer group, you can install the library in */usr/local/lib* and copy the header file to */usr/include* with the command

```
sudo make install
```

### 2.1.2 Windows

On Windows you need a compiler like Microsoft Visual Studio. You can build the library and the example programs with the command line tool of the compiler by typing

```
nmake -f Makefile.nmake
```

in the directory *usrsctp*.

## 2.2 Running the Test Programs

There are two test programs included, a discard server and a client. You can run both to send data from the client to the server. The client reads data from stdin and sends them to the server, which prints the message in the terminal and discards it. The sources of the server are also provided in Section **??** and those of the client in Section **??**.

### 2.2.1 Using UDP Encapsulation

Both programs can either send data over SCTP directly or use UDP encapsulation, thus encapsulating the SCTP packet in a UDP datagram. The first mode works on loopback or in a protected setup without any NAT boxes involved. In all other cases it is better to use UDP encapsulation.

The usage of the discard_server is

```
discard_server [local_encaps_port remote_encaps_port]
```

For UDP encapsulation the ports have to be specified. The local and remote encapsulation ports can be arbitrarily set. For example, you can call

```
./discard_server 11111 22222
```

on a Unix-like OS and

```
discard_server.exe 11111 22222
```

on Windows.

The client needs two additional parameters, the server's address and its port. Its usage is

```
client remote_addr remote_port [local_port local_encaps_port remote_encaps_port]
```

The remote address is the server's address. If client and server are started on the same machine, the loopback address 127.0.0.1 can be used for Unix-like OSs and the local address on Windows. The discard port is 9, thus 9 has to be taken as remote port. The encapsulation ports have to match those of the server, i.e. the server's local_encaps_port is the client's remote_encaps_port and vice versa. Thus, the client can be started with

```
./client 127.0.0.1 9 0 22222 11111
```

on a Unix-like OS and

```
client.exe 192.168.0.1 9 0 22222 11111
```

on Windows provided your local IP address is 192.168.0.1.

#### 2.2.2 Sending over SCTP

To send data over SCTP directly you might need root privileges because raw sockets are used. Thus instead of specifying the encapsulation ports you have to start the programs prepending `sudo` or in case of Windows start the program from an administrator console.

#### 2.2.3 Using the Callback API

Instead of asking constantly for new data, a callback API can be used that is triggered by SCTP. A callback function has to be registered that will be called whenever data is ready to be delivered to the application.

The discard_server has a flag to switch between the two modi. If use_cb is set to 1, the callback API will be used. To change the setting, just set the flag and compile the program again.

# 3 Basic Operations

All system calls start with the prefix *usrsctp_* to distinguish them from the kernel variants. Some of them are changed to account for the different demands in the userland environment.

## 3.1 Differences to RFC 6458

### 3.1.1 usrsctp_init()

Every application has to start with *usrsctp_init()*. This function calls *sctp_init()* and reserves the memory necessary to administer the data transfer. The function prototype is

```
void
usrsctp_init(uint16_t udp_port)
```

As it is not always possible to send data directly over SCTP because not all NAT boxes can process SCTP packets, the data can be sent over UDP. To encapsulate SCTP into UDP a UDP port has to be specified, to which the datagrams can be sent. This local UDP port is set with the parameter `udp_port`. The default value is 9899, the standard UDP encapsulation port. If UDP encapsulation is not necessary, the UDP port has to be set to 0.

### 3.1.2 usrsctp_finish()

At the end of the program *usrsctp_finish()* should be called to free all the memory that has been allocated before. The function prototype is

```
int
usrsctp_finish(void).
```

The return code is 0 on success and -1 in case of an error.

### 3.1.3 usrsctp_socket()

A representation of an SCTP endpoint is a socket. Is it created with *usrsctp_socket()*. The function prototype is

```
struct socket *
usrsctp_socket(int domain,
               int type,
               int protocol,
               int (*receive_cb)(struct socket *sock,
                                 union sctp_sockstore addr,
                                 void *data,
                                 size_t datalen,
                                 struct sctp_rcvinfo,
                                 int flags),
               int (*send_cb)(struct socket *sock,
                              uint32_t sb_free),
               uint32_t sb_threshold).
```

and the arguments taken from RFC 6458 are

- domain: PF_INET or PF_INET6 can be used.
- type: In case of a one-to-many style socket it is SOCK_SEQPACKET, in case of a one-to-one style socket it is SOCK_STREAM. For an explanation of the differences between the socket types please refer to RFC 6458.
- protocol: Set IPPROTO_SCTP.

In usrsctp a callback API can be used. The function pointers of the receive and send callbacks are new arguments to the socket call. They are NULL, if no callback API is used. The sb_threshold specifies the amount of free space in the send socket buffer before the send function in the application is called. If a send callback function is specified and sb_threshold is 0, the function is called whenever there is room in the send socket buffer.

On success *usrsctp_socket()* returns the pointer to the new socket in the `struct socket` data type. It will be needed in all other system calls. In case of a failure NULL is returned and errno is set to the appropriate error code.

### 3.1.4 usrsctp_close()

The function prototype of *usrsctp_close()* is

```
void
usrsctp_close(struct socket *so).
```

Thus the only difference is the absence of a return code.

## 3.2 Same Functionality as RFC 6458

The following functions have the same functionality as their kernel pendants. There prototypes are described in the following subsections. For a detailed description please refer to RFC 6458.

### 3.2.1 usrsctp_bind()

The function prototype of *usrsctp_bind()* is

```
int
usrsctp_bind(struct socket *so,
             struct sockaddr *addr,
             socklen_t addrlen).
```

The arguments are

- so: Pointer to the socket as returned by *usrsctp_socket()*.
- addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address).
- addrlen: The size of the address structure.

*usrsctp_bind()* returns 0 on success and -1 in case of an error.

### 3.2.2 usrsctp_listen()

The function prototype of *usrsctp_listen()* is

```
int
usrsctp_listen(struct socket *so,
               int backlog).
```

The arguments are

- so: Pointer to the socket as returned by *usrsctp_socket()*.
- backlog: If backlog is non-zero, enable listening, else disable listening.

*usrsctp_listen()* returns 0 on success and -1 in case of an error.

### 3.2.3 usrsctp_accept()

The function prototype of *usrsctp_accept()* is

```
struct socket *
usrsctp_accept(struct socket *so,
               struct sockaddr * addr,
               socklen_t * addrlen).
```

The arguments are

- so: Pointer to the socket as returned by *usrsctp_socket()*.
- addr: On return, the primary address of the peer (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address).
- addrlen: Size of the returned address structure.

*usrsctp_accept()* returns the accepted socket on success and NULL in case of an error.

### 3.2.4 usrsctp_connect()

The function prototype of *usrsctp_connect()* is

```
int
usrsctp_connect(struct socket *so,
                struct sockaddr *name,
                socklen_t addrlen)
```

The arguments are

- so: Pointer to the socket as returned by *usrsctp_socket()*.
- name: Address of the peer to connect to (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address).
- addrlen: Size of the peer's address.

*usrsctp_connect()* returns 0 on success and -1 in case of an error.

### 3.2.5 usrsctp_shutdown()

The function prototype of *usrsctp_shutdown()* is

```
int
usrsctp_shutdown(struct socket *so, int how).
```

The arguments are

- so: Pointer to the socket of the association to be closed.
- how: Specifies the type of shutdown. The values are as follows:

- SHUT_RD: Disables further receive operations. No SCTP protocol action is taken.
- SHUT_WR: Disables further send operations, and initiates the SCTP shutdown sequence.
- SHUT_RDWR: Disables further send and receive operations, and initiates the SCTP shutdown sequence.

*usrsctp_listen()* returns 0 on success and -1 in case of an error.

## 4 Sending and Receiving Data

Since the publication of RFC 6458 there is only one function for sending and one for receiving that is not deprecated. Therefore, only these two are described here.

### 4.1 usrsctp_sendv()

The function prototype is

```
ssize_t
usrsctp_sendv(struct socket *so,
              const void *data,
              size_t len,
              struct sockaddr *addrs,
              int addrcnt,
              void *info,
              socklen_t infolen,
              unsigned int infotype,
              int flags).
```

The arguments are

- so: The socket to send data on.
- data: As it is more convenient to send data in a buffer and not a `struct iovec` data structure, we chose to pass the data as a void pointer.
- len: Length of the data.
- addrs: In this version of usrsctp at most one destination address is supported. In the case of a connected socket, the parameter `addrs` can be set to NULL.
- addrcnt: Number of addresses. As at most one address is supported, addrcnt is 0 if addrs is NULL and 1 otherwise.
- info: Additional information for a message is stored in `void *info`. The data types `struct sctp_sndinfo`, `struct sctp_prinfo`, and `struct sctp_sendv_spa` are supported as defined in RFC 6458. Support for `structsctp_authinfo` is not implemented yet, therefore, errno is set EINVAL and -1 will be returned, if it is used.

- infolen: Length of info in bytes.
- infotype: Identifies the type of the information provided in info. Possible values are SCTP_SENDV_NOINFO, SCTP_SENDV_SNDINFO, SCTP_SENDV_PRINFO, SCTP_SENDV_SPA. For additional information please refer to RFC 6458.
- flags: Flags as described in RFC 6458.

*usrsctp_sendv()* returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

## 4.2 usrsctp_recvv()

The function prototype is

```
ssize_t
usrsctp_recvv(struct socket *so,
             void *dbuf,
             size_t len,
             struct sockaddr *from,
             socklen_t * fromlen,
             void *info,
             socklen_t *infolen,
             unsigned int *infotype,
             int *msg_flags).
```

The arguments are

- so: The socket to receive data on.
- dbuf: Analog to *usrsctp_sendv()* the data is returned in a buffer.
- len: Length of the buffer in bytes.
- from: A pointer to an address to be filled with the sender of the received message's address.
- fromlen: An in/out parameter describing the from length.
- info: A pointer to the buffer to hold the attributes of the received message. The structure type of info is determined by the infotype parameter. The attributes returned in `info` have to be handled in the same way as specified in RFC 6458.
- infolen: An in/out parameter describing the size of the info buffer.
- infotype: On return, *infotype is set to the type of the info buffer. The current defined values are SCTP_RECVV_NOINFO, SCTP_RECVV_RCVINFO, SCTP_RECVV_NXTINFO, and SCTP_RECVV_RN. A detailed description is given in RFC 6458.
- flags: A pointer to an integer to be filled with any message flags (e.g., MSG_NOTIFICATION). Note that this field is an in/out parameter. Options for the receive may also be passed into the value (e.g., MSG_EOR). Returning from the call, the flags' value will differ from its original value.

*usrsctp_recvv()* returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

## 5 Socket Options

Socket options are used to change the default behavior of socket calls. Their behavior is specified in RFC 6458. The functions to get or set them are

```
int
usrsctp_getsockopt(struct socket *so,
                   int level,
                   int optname,
                   void *optval,
                   socklen_t *optlen)
```

and

```
int
usrsctp_setsockopt(struct socket *so,
                   int level,
                   int optname,
                   const void *optval,
                   socklen_t optlen).
```

and the arguments are

- so: The socket of type struct socket.
- level: Set to IPPROTO_SCTP for all SCTP options.
- optname: The option name as specified in Table ??.
- optval: The buffer to store the value of the option as specified in the second column of Table ??.
- optlen: The size of the buffer (or the length of the option returned in case of *usrsctp_getsockopt*).

These functions return 0 on success and -1 in case of an error.

| **Option** | **Datatype** | **r/w** |
|---|---|---|
| SCTP_RTOINFO | struct sctp_rtoinfo | r/w |
| SCTP_ASSOCINFO | struct sctp_assocparams | r/w |
| SCTP_INITMSG | struct sctp_initmsg | r/w |
| SCTP_NODELAY | int | r/w |
| SCTP_AUTOCLOSE | int | r/w |
| SCTP_PRIMARY_ADDR | struct sctp_setprim | r/w |
| SCTP_ADAPTATION_LAYER | struct sctp_setadaptation | r/w |
| SCTP_DISABLE_FRAGMENTS | int | r/w |
| SCTP_PEER_ADDR_PARAMS | struct sctp_paddrparams | r/w |
| SCTP_I_WANT_MAPPED_V4_ADDR | int | r/w |
| SCTP_MAXSEG | struct sctp_assoc_value | r/w |

| Option | Datatype | r/w |
|---|---|---|
| SCTP_DELAYED_SACK | struct sctp_sack_info | r/w |
| SCTP_FRAGMENT_INTERLEAVE | int | r/w |
| SCTP_PARTIAL_DELIVERY_POINT | int | r/w |
| SCTP_HMAC_IDENT | struct sctp_hmacalgo | r/w |
| SCTP_AUTH_ACTIVE_KEY | struct sctp_authkeyid | r/w |
| SCTP_AUTO_ASCONF | int | r/w |
| SCTP_MAX_BURST | struct sctp_assoc_value | r/w |
| SCTP_CONTEXT | struct sctp_assoc_value | r/w |
| SCTP_EXPLICIT_EOR | int | r/w |
| SCTP_REUSE_PORT | int | r/w |
| SCTP_EVENT | struct sctp_event | r/w |
| SCTP_RECVRCVINFO | int | r/w |
| SCTP_RECVNXTINFO | int | r/w |
| SCTP_DEFAULT_SNDINFO | struct sctp_sndinfo | r/w |
| SCTP_DEFAULT_PRINFO | struct sctp_default_prinfo | r/w |
| SCTP_REMOTE_UDP_ENCAPS_PORT | int | r/w |
| SCTP_ENABLE_STREAM_RESET | struct sctp_assoc_value | r/w |
| SCTP_STATUS | struct sctp_status | r |
| SCTP_GET_PEER_ADDR_INFO | struct sctp_paddrinfo | r |
| SCTP_PEER_AUTH_CHUNKS | struct sctp_authchunks | r |
| SCTP_LOCAL_AUTH_CHUNKS | struct sctp_authchunks | r |
| SCTP_GET_ASSOC_NUMBER | uint32_t | r |
| SCTP_GET_ASSOC_ID_LIST | struct sctp_assoc_ids | r |
| SCTP_RESET_STREAMS | struct sctp_reset_streams | w |
| SCTP_RESET_ASSOC | struct sctp_assoc_t | w |
| SCTP_ADD_STREAMS | struct sctp_add_streams | w |

Table 1: Socket Options supported by usrsctp

An overview of the supported options is given in Table ??. Their use is described in RFC 6458 [?], RFC 6525 [?], and [?].

## 6 Sysctl variables

In kernel implementations like for instance FreeBSD, it is possible to change parameters in the operating system. These parameters are called sysctl variables.

In usrsctp applications can set or retrieve these variables with the functions

```
void
usrsctp_sysctl_set_ ## (uint32_t value)
```

and

```
uint32_t
usrsctp_sysctl_get_ ## (void)
```

respectively, where ## stands for the name of the variable.

In the following paragraphs a short description of the parameters will be given.

## 6.1 Manipulate Memory

### 6.1.1 usrsctp_sysctl_set_sctp_sendspace()

The space of the available send buffer can be changed from its default value of 262,144 bytes to a value between 0 and $2^{32} - 1$ bytes.

### 6.1.2 usrsctp_sysctl_set_sctp_recvspace()

The space of the available receive buffer can be changed from its default value of 262,144 bytes to a value between 0 and $2^{32} - 1$ bytes.

### 6.1.3 usrsctp_sysctl_set_sctp_hashtblsize()

The TCB (Thread Control Block) hash table sizes, i.e. the size of one TCB in the hash table, can be tuned between 1 and $2^{32} - 1$ bytes. The default value is 1,024 bytes. A TCB contains for instance pointers to the socket, the endpoint, information about the association and some statistic data.

### 6.1.4 usrsctp_sysctl_set_sctp_pcbtblsize()

The PCB (Protocol Control Block) hash table sizes, i.e. the size of one PCB in the hash table, can be tuned between 1 and $2^{32} - 1$ bytes. The default value is 256 bytes. The PCB contains all variables that characterize an endpoint.

### 6.1.5 usrsctp_sysctl_set_sctp_system_free_resc_limit()

This parameters tunes the maximum number of cached resources in the system. It can be set between 0 and $2^{32} - 1$. The default value is 1000.

### 6.1.6 usrsctp_sysctl_set_sctp_asoc_free_resc_limit()

This parameters tunes the maximum number of cached resources in an association. It can be set between 0 and $2^{32} - 1$. The default value is 10.

### 6.1.7 usrsctp_sysctl_set_sctp_mbuf_threshold_count()

Data is stored in mbufs. Several mbufs can be chained together. The maximum number of small mbufs in a chain can be set with this parameter, before an mbuf cluset is used. The default is 5.

### 6.1.8 usrsctp_sysctl_set_sctp_add_more_threshold()

TBD This parameter configures the threshold below which more space should be added to a socket send buffer. The default value is 1452 bytes.

## 6.2 Configure RTO

The retransmission timeout (RTO), i.e. the time that controls the retransmission of messages, has several parameters, that can be changed, for example to shorten the time, before a message is retransmitted. The range of these parameters is between 0 and $2^{32} - 1$ ms.

#### 6.2.1 usrsctp_sysctl_set_sctp_rto_max_default()

The default value for the maximum retransmission timeout in ms is 60,000 (60 secs).

#### 6.2.2 usrsctp_sysctl_set_sctp_rto_min_default()

The default value for the minimum retransmission timeout in ms is 1,000 (1 sec).

#### 6.2.3 usrsctp_sysctl_set_sctp_rto_initial_default()

The default value for the initial retransmission timeout in ms is 3,000 (3 sec). This value is only needed before the first calculation of a round trip time took place.

#### 6.2.4 usrsctp_sysctl_set_sctp_init_rto_max_default()

The default value for the maximum retransmission timeout for an INIT chunk in ms is 60,000 (60 secs).

### 6.3 Set Timers

#### 6.3.1 usrsctp_sysctl_set_sctp_valid_cookie_life_default()

A cookie has a specified life time. If it expires the cookie is not valid any more and an ABORT is sent. The default value in ms is 60,000 (60 secs).

#### 6.3.2 usrsctp_sysctl_set_sctp_heartbeat_interval_default()

Set the default time between two heartbeats. The default is 30,000 ms.

#### 6.3.3 usrsctp_sysctl_set_sctp_shutdown_guard_time_default()

If a SHUTDOWN is not answered with a SHUTDOWN-ACK while the shutdown guard timer is still running, the association will be aborted after the default of 180 secs.

#### 6.3.4 usrsctp_sysctl_set_sctp_pmtu_raise_time_default()

TBD To set the size of the packets to the highest value possible, the maximum transfer unit (MTU) of the complete path has to be known. The default time interval for the path mtu discovery is 600 secs.

#### 6.3.5 usrsctp_sysctl_set_sctp_secret_lifetime_default()

TBD The default secret lifetime of a server is 3600 secs.

#### 6.3.6 usrsctp_sysctl_set_sctp_vtag_time_wait()

TBD Vtag time wait time, 0 disables it. Default: 60 secs

## 6.4 Set Failure Limits

Transmissions and retransmissions of messages might fail. To protect the system against too many retransmissions, limits have to be defined.

### 6.4.1 usrsctp_sysctl_set_sctp_init_rtx_max_default()

The default maximum number of retransmissions of an INIT chunks is 8, before an ABORT is sent.

### 6.4.2 usrsctp_sysctl_set_sctp_assoc_rtx_max_default()

This parameter sets the maximum number of failed retransmissions before the association is aborted. The default vaule is 10.

### 6.4.3 usrsctp_sysctl_set_sctp_path_rtx_max_default()

This parameter sets the maximum number of path failures before the association is aborted. The default value is 5. Notice that the number of paths multiplied by this value should be equal to sctp_assoc_rtx_max_default. That means that the default configuration is good for two paths.

### 6.4.4 usrsctp_sysctl_set_sctp_max_retran_chunk()

The parameter configures how many times an unlucky chunk can be retransmitted before the association aborts. The default is set to 30.

### 6.4.5 usrsctp_sysctl_set_sctp_path_pf_threshold()

TBD Default potentially failed threshold. Default: 65535

### 6.4.6 usrsctp_sysctl_set_sctp_abort_if_one_2_one_hits_limit()

TBD When one-2-one hits qlimit abort. Default: 0

## 6.5 Control the Sending of SACKs

### 6.5.1 usrsctp_sysctl_set_sctp_sack_freq_default()

The SACK frequency defines the number of packets that are awaited, before a SACK is sent. The default value is 2.

### 6.5.2 usrsctp_sysctl_set_sctp_delayed_sack_time_default()

As a SACK (Selective Acknowlegment) is sent after every other packet, a timer is set to send a SACK in case another packet does not arrive in due time. The default value for this timer is 200 ms.

### 6.5.3 usrsctp_sysctl_set_sctp_strict_sacks()

TBD This is a flag to turn the controlling of the coherence of SACKs on or off. The default value is 1 (on).

#### 6.5.4 usrsctp_sysctl_set_sctp_nr_sack_on_off()

If a slow hosts receives data on a lossy link it is possible that its receiver window is full and new data can only be accepted if one chunk with a higher TSN (Transmission Sequence Number) that has previously been acknowledged is dropped. As a consequence the sender has to store data, even if they have been acknowledged in case they have to be retransmitted. If this behavior is not necessary, non-renegable SACKs can be turned on. By default the use of non-renegable SACKs is turned off.

#### 6.5.5 usrsctp_sysctl_set_sctp_enable_sack_immediately()

In some cases it is not desirable to wait for the SACK timer to expire before a SACK is sent. In these cases a bit called SACK-IMMEDIATELY [?] can be set to provoke the instant sending of a SACK. The default is to turn it off.

#### 6.5.6 usrsctp_sysctl_set_sctp_L2_abc_variable()

TBD SCTP ABC max increase per SACK (L). Default: 1

## 6.6 Change Max Burst

Max burst defines the maximum number of packets that may be sent in one flight.

#### 6.6.1 usrsctp_sysctl_set_sctp_max_burst_default()

The default value for max burst is 0, which means that the number of packets sent as a flight is not limited by this parameter, but may be by another one, see the next paragraph.

#### 6.6.2 usrsctp_sysctl_set_sctp_use_cwnd_based_maxburst()

The use of max burst is based on the size of the congestion window (cwnd). This parameter is set by default.

#### 6.6.3 usrsctp_sysctl_set_sctp_hb_maxburst()

Heartbeats are mostly used to verify a path. Their number can be limited. The default is 4.

#### 6.6.4 usrsctp_sysctl_set_sctp_fr_max_burst_default()

In the state of fast retransmission the number of packet bursts can be limited. The default value is 4.

## 6.7 Handle Chunks

#### 6.7.1 usrsctp_sysctl_set_sctp_peer_chunk_oh()

In order to keep track of the peer's advertised receiver window, the sender calculates the window by subtracting the amount of data sent. Yet, some OSs reduce the receiver window by the real space needed to store the data. This parameter sets the additional amount to debit the peer's receiver window per chunk sent. The default value is 256, which is the value needed by FreeBSD.

#### 6.7.2 usrsctp_sysctl_set_sctp_max_chunks_on_queue()

This parameter sets the maximum number of chunks that can be queued per association. The default value is 512.

#### 6.7.3 usrsctp_sysctl_set_sctp_min_split_point()

TBD The minimum size when splitting a chunk is 2904 bytes by default.

#### 6.7.4 usrsctp_sysctl_set_sctp_chunkscale()

TBD This parameter can be tuned for scaling of number of chunks and messages. The default is10.

#### 6.7.5 usrsctp_sysctl_set_sctp_min_residual()

TBD This parameter configures the minimum size of the residual data chunk in the second part of the split. The default is 1452.

### 6.8 Calculate RTT

The calculation of the round trip time (RTT) depends on several parameters.

#### 6.8.1 usrsctp_sysctl_set_sctp_rttvar_bw()

TBD Shift amount for bw smoothing on rtt calc. Default: 4

#### 6.8.2 usrsctp_sysctl_set_sctp_rttvar_rtt()

TBD Shift amount for rtt smoothing on rtt calc. Default: 5

#### 6.8.3 usrsctp_sysctl_set_sctp_rttvar_eqret()

TBD What to return when rtt and bw are unchanged. Default: 0

### 6.9 Influence the Congestion Control

The congestion control should protect the network against fast senders.

#### 6.9.1 usrsctp_sysctl_set_sctp_ecn_enable

Explicit congestion notifications are turned on by default.

#### 6.9.2 usrsctp_sysctl_set_sctp_default_cc_module()

This parameter sets the default algorithm for the congestion control. Default is 0, i.e. the one specified in RFC 4960.

#### 6.9.3 usrsctp_sysctl_set_sctp_initial_cwnd()

Set the initial congestion window in MTUs. The default is 3.

#### 6.9.4 usrsctp_sysctl_set_sctp_use_dccc_ecn()

TBD Enable for RTCC CC datacenter ECN. Default: 1

#### 6.9.5 usrsctp_sysctl_set_sctp_steady_step()

TBD How many the sames it takes to try step down of cwnd. Default: 20

## 6.10 Configure AUTH and ADD-IP

An important extension of SCTP is the dynamic address reconfiguration [?], also known as ADD-IP, which allows the changing of addresses during the lifetime of an association. For this feature the AUTH extension [?] is necessary.

#### 6.10.1 usrsctp_sysctl_set_sctp_auto_asconf()

If SCTP Auto-ASCONF is enabled, the peer is informed automatically when a new address is added or removed. This feature is enabled by default.

#### 6.10.2 usrsctp_sysctl_set_sctp_multiple_asconfs()

By default the sending of multiple ASCONFs is disabled.

#### 6.10.3 usrsctp_sysctl_set_sctp_auth_disable()

The use of AUTH, which is normally turned on, can be disabled by setting this parameter to 1.

#### 6.10.4 usrsctp_sysctl_set_sctp_asconf_auth_nochk()

It is also possible to disable the requirement to use AUTH in conjunction with ADD-IP by setting this parameter to 1.

## 6.11 Concurrent Multipath Transfer (CMT)

A prominent feature of SCTP is the possibility to use several addresses for the same association. One is the primary path, and the others are needed in case of a path failure. Using CMT the data is sent on several paths to enhance the throughput.

#### 6.11.1 usrsctp_sysctl_set_sctp_cmt_on_off()

To turn CMT on, this parameter has to be set to 1.

#### 6.11.2 usrsctp_sysctl_set_sctp_cmt_use_dac()

To use delayed acknowledgments with CMT this parameter has to be set to 1.

### 6.11.3 usrsctp_sysctl_set_sctp_buffer_splitting()

For CMT it makes sense to split the send and receive buffer to have shares for each path. By default buffer splitting is turned off.

## 6.12 Network Address Translation (NAT)

To be able to pass NAT boxes, the boxes have to handle SCTP packets in a specific way.

### 6.12.1 usrsctp_sysctl_set_sctp_nat_friendly()

SCTP NAT friendly operation. Default:1

### 6.12.2 usrsctp_sysctl_set_sctp_inits_include_nat_friendly()

Enable sending of the nat-friendly SCTP option on INITs. Default: 0

### 6.12.3 usrsctp_sysctl_set_sctp_udp_tunneling_port()

Set the SCTP/UDP tunneling port. Default: 9899

## 6.13 SCTP Mobility

### 6.13.1 usrsctp_sysctl_set_sctp_mobility_base()

TBD Enable SCTP base mobility. Default: 0

### 6.13.2 usrsctp_sysctl_set_sctp_mobility_fasthandoff()

TBD Enable SCTP fast handoff. default: 0

## 6.14 Miscellaneous

### 6.14.1 usrsctp_sysctl_set_sctp_no_csum_on_loopback()

Calculating the checksum for packets sent on loopback is turned off by default. To turn it on, set this parameter to 0.

### 6.14.2 usrsctp_sysctl_set_sctp_nr_outgoing_streams_default()

The peer is notified about the number of outgoing streams in the INIT or INIT-ACK chunk. The default is 10.

### 6.14.3 usrsctp_sysctl_set_sctp_do_drain()

Determines whether SCTP should respond to the drain calls. Default: 1

### 6.14.4 usrsctp_sysctl_set_sctp_strict_data_order()

TBD Enforce strict data ordering, abort if control inside data. Default: 0

#### 6.14.5 usrsctp_sysctl_set_sctp_default_ss_module()

Set the default stream scheduling module. Implemented modules are: SCTP_SS_DEFAULT, SCTP_SS_ROUND_ROBIN, SCTP_SS_ROUND_ROBIN_PACKET, SCTP_SS_PRIORITY, SCTP_SS_FAIR_BANDWITH, and SCTP_SS_FIRST_COME.

#### 6.14.6 usrsctp_sysctl_set_sctp_default_frag_interleave()

TBD Default fragment interleave level. Default: 1

#### 6.14.7 usrsctp_sysctl_set_sctp_blackhole()

TBD Enable SCTP blackholing. Default: 0

#### 6.14.8 usrsctp_sysctl_set_sctp_logging_level()

Set the logging level. The default is 0.

#### 6.14.9 usrsctp_sysctl_set_sctp_debug_on()

Turn debug output on or off. It is disabled by default. To obtain debug output, SCTP_DEBUG has to be set as a compile flag.

# 7 Examples

## 7.1 Discard Server

```
/*
 * Copyright (C) 2011-2012 Michael Tuexen
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the project nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/*
 * Usage: discard_server [local_encaps_port] [remote_encaps_port]
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#if !defined(__Userspace_os_Windows)
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#endif
#include <usrsctp.h>

#define BUFFER_SIZE 10240

const int use_cb = 0;

static int
receive_cb(struct socket *sock, union sctp_sockstore addr, void *data,
           size_t datalen, struct sctp_rcvinfo rcv, int flags)
{
    char name[INET6_ADDRSTRLEN];

    if (data) {
        if (flags & MSG_NOTIFICATION) {
            printf("Notification of length %d received.\n", (int)datalen);
        } else {
            printf("Msg of length %d received from %s:%u on stream %d with "
                   "SSN %u and TSN %u, PPID %d, context %u.\n",
                   (int)datalen,
                   addr.sa.sa_family == AF_INET ?
                       inet_ntop(AF_INET, &addr.sin.sin_addr, name,
                                 INET6_ADDRSTRLEN):
                       inet_ntop(AF_INET6, &addr.sin6.sin6_addr, name,
                                 INET6_ADDRSTRLEN),
                   ntohs(addr.sin.sin_port),
                   rcv.rcv_sid,
                   rcv.rcv_ssn,
                   rcv.rcv_tsn,
                   ntohl(rcv.rcv_ppid),
                   rcv.rcv_context);
        }
        free(data);
    }
    return 1;
}

int
main(int argc, char *argv[])
{
    struct socket *sock;
    struct sockaddr_in6 addr;
    struct sctp_udpencaps encaps;
    struct sctp_event event;
    uint16_t event_types[] = {SCTP_ASSOC_CHANGE,
                              SCTP_PEER_ADDR_CHANGE,
                              SCTP_REMOTE_ERROR,
                              SCTP_SHUTDOWN_EVENT,
                              SCTP_ADAPTATION_INDICATION,
                              SCTP_PARTIAL_DELIVERY_EVENT};
    unsigned int i;
    struct sctp_assoc_value av;
    const int on = 1;
    int n, flags;
    socklen_t from_len;
    char buffer[BUFFER_SIZE];
    char name[INET6_ADDRSTRLEN];
    struct sctp_recvv_rn rn;
    socklen_t infolen = sizeof(struct sctp_recvv_rn);
    struct sctp_rcvinfo rcv;
    struct sctp_nxtinfo nxt;
    unsigned int infotype = 0;

    if (argc > 1) {
        usrsctp_init(atoi(argv[1]));
    } else {
        usrsctp_init(9899);
    }
    usrsctp_sysctl_set_sctp_debug_on(0);
    usrsctp_sysctl_set_sctp_blackhole(2);

    if ((sock = usrsctp_socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP,
                               use_cb?receive_cb:NULL, NULL, 0)) == NULL) {
        perror("userspace_socket");
    }
    if (usrsctp_setsockopt(sock, IPPROTO_SCTP, SCTP_I_WANT_MAPPED_V4_ADDR,
                           (const void*)&on, (socklen_t)sizeof(int)) < 0) {
        perror("setsockopt");
    }
    memset(&av, 0, sizeof(struct sctp_assoc_value));
    av.assoc_id = SCTP_ALL_ASSOC;
    av.assoc_value = 47;

    if (usrsctp_setsockopt(sock, IPPROTO_SCTP, SCTP_CONTEXT, (const void*)&av,
                           (socklen_t)sizeof(struct sctp_assoc_value)) < 0) {
        perror("setsockopt");
    }
    if (argc > 2) {
        memset(&encaps, 0, sizeof(struct sctp_udpencaps));
        encaps.sue_address.ss_family = AF_INET6;
        encaps.sue_port = htons(atoi(argv[2]));
        if (usrsctp_setsockopt(sock, IPPROTO_SCTP, SCTP_REMOTE_UDP_ENCAPS_PORT,
                               (const void*)&encaps,
                               (socklen_t)sizeof(struct sctp_udpencaps)) < 0) {
            perror("setsockopt");
        }
    }
    memset(&event, 0, sizeof(event));
    event.se_assoc_id = SCTP_FUTURE_ASSOC;
    event.se_on = 1;
    for (i = 0; i < (unsigned int)(sizeof(event_types)/sizeof(uint16_t)); i++) {
        event.se_type = event_types[i];
        if (usrsctp_setsockopt(sock, IPPROTO_SCTP, SCTP_EVENT, &event,
                               sizeof(struct sctp_event)) < 0) {
            perror("userspace_setsockopt");
        }
    }
    memset((void *)&addr, 0, sizeof(struct sockaddr_in6));
#ifdef HAVE_SIN_LEN
    addr.sin6_len = sizeof(struct sockaddr_in6);
#endif
    addr.sin6_family = AF_INET6;
    addr.sin6_port = htons(9);
    addr.sin6_addr = in6addr_any;
    if (usrsctp_bind(sock, (struct sockaddr *)&addr,
                     sizeof(struct sockaddr_in6)) < 0) {
        perror("userspace_bind");
    }
    if (usrsctp_listen(sock, 1) < 0) {
        perror("userspace_listen");
    }
    while (1) {
        if (use_cb) {
#if defined (__Userspace_os_Windows)
            Sleep(1*1000);
#else
            sleep(1);
#endif
        } else {
            from_len = (socklen_t)sizeof(struct sockaddr_in6);
            flags = 0;
            rn.recvv_rcvinfo = rcv;
            rn.recvv_nxtinfo = nxt;
            n = usrsctp_recvv(sock, (void*)buffer, BUFFER_SIZE,
                              (struct sockaddr *) &addr, &from_len, (void *)&rn,
                     &infolen, &infotype, &flags);
            if (n > 0) {
                if (flags & MSG_NOTIFICATION) {
                    printf("Notification of length %d received.\n", n);
                } else {
                    printf("Msg of length %d received from %s:%u on stream "
                           "%d with SSN %u and TSN %u, PPID %d, context %u, "
                           "complete %d.\n",
                           n,
                           inet_ntop(AF_INET6, &addr.sin6_addr, name,
                                     INET6_ADDRSTRLEN), ntohs(addr.sin6_port),
                           rn.recvv_rcvinfo.rcv_sid,
                           rn.recvv_rcvinfo.rcv_ssn,
                           rn.recvv_rcvinfo.rcv_tsn,
                           ntohl(rn.recvv_rcvinfo.rcv_ppid),
                           rn.recvv_rcvinfo.rcv_context,
                           (flags & MSG_EOR) ? 1 : 0);
                }
            }
        }
    }
    usrsctp_close(sock);
    while (usrsctp_finish() != 0) {
#if defined (__Userspace_os_Windows)
        Sleep(1000);
#else
        sleep(1);
#endif
    }
    return (0);
}
```

## 7.2 Client

```
/*
 * Copyright (C) 2011-2012 Michael Tuexen
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the project nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/*
 * Usage: client remote_addr remote_port [local_encaps_port remote_encaps_port]
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#if !defined(__Userspace_os_Windows)
#include <unistd.h>
#endif
#include <sys/types.h>
#if !defined(__Userspace_os_Windows)
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#endif
#include <usrsctp.h>

int done = 0;

static int
receive_cb(struct socket *sock, union sctp_sockstore addr, void *data,
           size_t datalen, struct sctp_rcvinfo rcv, int flags)
{
    if (data == NULL) {
        done = 1;
        usrsctp_close(sock);
    } else {
        write(fileno(stdout), data, datalen);
        free(data);
    }
    return 1;
}

int
main(int argc, char *argv[])
{
    struct socket *sock;
    struct sockaddr_in addr4;
    struct sockaddr_in6 addr6;
    struct sctp_udpencaps encaps;
    char buffer[80];

    if (argc > 3) {
        usrsctp_init(atoi(argv[3]));
    } else {
        usrsctp_init(9899);
    }
    usrsctp_sysctl_set_sctp_debug_on(0);
    usrsctp_sysctl_set_sctp_blackhole(2);
    if ((sock = usrsctp_socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP,
                               receive_cb, NULL, 0)) == NULL) {
        perror("userspace_socket ipv6");
    }
    if (argc > 4) {
        memset(&encaps, 0, sizeof(struct sctp_udpencaps));
        encaps.sue_address.ss_family = AF_INET6;
        encaps.sue_port = htons(atoi(argv[4]));
        if (usrsctp_setsockopt(sock, IPPROTO_SCTP, SCTP_REMOTE_UDP_ENCAPS_PORT,
                               (const void*)&encaps,
                               (socklen_t)sizeof(struct sctp_udpencaps)) < 0) {
            perror("setsockopt");
        }
    }
    memset((void *)&addr4, 0, sizeof(struct sockaddr_in));
    memset((void *)&addr6, 0, sizeof(struct sockaddr_in6));
#if !defined(__Userspace_os_Linux) && !defined(__Userspace_os_Windows)
    addr4.sin_len = sizeof(struct sockaddr_in);
    addr6.sin6_len = sizeof(struct sockaddr_in6);
#endif
    addr4.sin_family = AF_INET;
    addr6.sin6_family = AF_INET6;
    addr4.sin_port = htons(atoi(argv[2]));
    addr6.sin6_port = htons(atoi(argv[2]));
    if (inet_pton(AF_INET6, argv[1], &addr6.sin6_addr) == 1) {
        if (usrsctp_connect(sock, (struct sockaddr *)&addr6,
                            sizeof(struct sockaddr_in6)) < 0) {
            perror("userspace_connect");
        }
    } else if (inet_pton(AF_INET, argv[1], &addr4.sin_addr) == 1) {
        if (usrsctp_connect(sock, (struct sockaddr *)&addr4,
                            sizeof(struct sockaddr_in)) < 0) {
            perror("userspace_connect");
        }
    } else {
        printf("Illegal destination address.\n");
    }
    while ((fgets(buffer, sizeof(buffer), stdin) != NULL) && !done) {
        usrsctp_sendv(sock, buffer, strlen(buffer), NULL, 0,
                  NULL, 0, SCTP_SENDV_NOINFO, 0);
    }
    if (!done) {
        usrsctp_shutdown(sock, SHUT_WR);
    }
    while (!done) {
#if defined (__Userspace_os_Windows)
        Sleep(1*1000);
#else
        sleep(1);
#endif
    }
    while (usrsctp_finish() != 0) {
#if defined (__Userspace_os_Windows)
        Sleep(1000);
#else
        sleep(1);
#endif
    }
    return(0);
}
```

## References

[1] R. Stewart, M. Tüxen, K. Poon, and V. Yasevich: *Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)*. RFC 6458, Dezember 2011.

[2] R. Stewart: *Stream Control Transmission Protocol*. RFC 4960, September 2007.

[3] M. Tüxen, R. Stewart, P. Lei, and E. Rescorla: *Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)*. RFC 4895, August 2007.

[4] R. Stewart, M. Tüxen, and P. Lei: *Stream Control Transmission Protocol (SCTP) Stream Reconfiguration*. RFC 6525, February 2012.

[5] R. Stewart, Q. Xie, M. Tüxen, S. Maruyama, and M. Kozuka: *Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration*. RFC 5061, September 2007.

[6] M. Tüxen, I. Rüngeler, and R. Stewart: *SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol*. draft-tuexen-tsvwg-sctp-sack-immediately-09 (work in progress), April 2012.

[7] M. Tüxen and R. Stewart *UDP Encapsulation of SCTP Packetsl*. draft-ietf-tsvwg-sctp-udp-encaps-03 (work in progress), March 2012.