Documentation of the SCTP-Implementation
Release: sctplib-1.0

Andreas Jungmaier
andreas.jungmaier@web.de
Michael Tüxen
tuexen@fh-muenster.de
Thomas Dreibholz
thomas.dreibholz@gmail.com

July 9, 2025

1 Nomenclature

Throughout this document,

- function names are written as **func_name**(),
- function parameters as `parameter`,
- data types as **typename**, and
- file names as *name_of_file*.
- constants as `CONSTANT`

2 General Concepts

The sctplib-1.0 library release is the product of a cooperation between Siemens AG (ICN), Munich, Germany and the Computer Networking Technology Group at the IEM of the University of Essen, Germany. It has been developed since late 1999 and has been planned to become a fairly complete prototype implementation of the SCTP protocol as described in [2].

The API of the function library was modeled after section 10 of the RFC 4960 [2], and most parameters an functions should be self-explanatory to the user familiar with this document. In addition to the interface functions between an Upper Layer Protocol (ULP) and an SCTP instance, the library also provides a number of helper functions that can be used to manage callback function routines to make them execute at a certain point of time (i.e. timer-based) or open and bind UDP sockets on a configurable port, which may then be used for an asynchronous interprocess communication.

All of these functions may be made use of by simply linking the static `libsctp`-library to an application, and including the file *sctp.h*. As SCTP operates on top of IP, we chose to open a raw socket to catch all incoming packets with the IP protocol id byte set to 132 (=SCTP). For this, it is necessary that an application using the library functions have privileges to do so. On most Unix machines that means this application has to be run by root (i.e. be made setuid 0).

An application making use of the `libsctp`-library can expect to be able to manage a large number of associations (e.g. as a server, all bound to one port) as well as handle several so-called *SCTP instances*, which may work on several different ports. Ports of incoming packets are checked whether they belong to an already existing association or may start a new one.

The concept of the library is the following: After registration of a first *SCTP instance*, sockets are opened and an application may register timer events, or file descriptors, that asynchronously trigger execution of callback functions. The function callbacks for these routines are passed in the registration functions, and callbacks for SCTP events are passed in a **SCTP_ulpCallbacks** (described more closely in sections 6.2.1 and 6.5).

Then the application either calls the possibly blocking function **sctp_eventLoop**() or the nonblocking function **sctp_getEvents**(). While calling the former, it will react to a previously scheduled timer or any file descriptor event (by executing the registered callback functions). In case a timer is scheduled at a very late point in time, and no events happen on registered file descriptors (e.g. sockets), the program will sleep (because the system call **poll**() is used. In this case, the control flow is handled by the library, and the user **must** register appropriate callbacks for events and timers before handing control over to the **sctp_eventLoop**() function. The proper use of the **sctp_eventLoop**() is explained in some simple example programs in section 7.

On the other hand, there is also the function **sctp_getEvents**() which may be used to check for timer or file descriptor events. If there are none, it returns immediately. While scheduling CPU intensive functions to run, an application should call this function in between in order to treat SCTP events while performing other tasks.

2.1 Notes

Note that the `libsctp`-library depends on a software packet (glib-2.0) for portable definition of types, list functions etc. Since we started using this late during the project's development cycle, its application is sometimes not very consequent. We continue working on that, though, so things should clean up a bit during later releases. For one, removal of the *dll_main.c/.h* files is planned in favor of the glib-list functions, so that linked list handling will become less prone to inefficient use of **malloc**() and **free**().

3 Convenient Use of the Library

If you compile the entire project, and install the files *sctp.h* in the directory `/usr/local/include` and *libsctp.a* in `/usr/local/lib` (by simply copying these from the `libsctp/sctp/` subdirectory to their repective destinations, it is trivial to compile an application.

Let's assume, your application is named `app` and consists of the file *app.c*. If you use `gcc` as compiler, it is sufficient to issue the command

```
host:~> gcc -o app app.c -lsctp -lglib2.0
```

in order to compile your application.

4 Features of the Implementation

The implementation is able to establish associations according to RFC 4960 and its interoperablility has been tested with implementations of about 25 companies. It features a complete API according to section 10 of RFC 4960 [2]. Naturally it supports the SCTP multihoming features, and is able to handle all-IPv4 as well as all-IPv6 or mixed IPv4/IPv6 associations. An association may be established using a Cookie-structure that is secured by a secret key and a secure MD5 hash function message authentication code.

The data path implements the flow control features as prescribed in the RFC, and supports the delayed acknowledgement feature, the delay of which may be configured, optionally. Using the **sctp_setPathStatus**(), **sctp_setAssocStatus**() and **sctp_setAssocDefaults**() functions, a wide range of protocol parameters may optionally be configured for each association.

There can be several SCTP instances per host, provided they have a set of disjoint local transport addresses. The implementation also supports *reassembly* of fragmented chunks, and delivery of unordered chunks, as well as ordered chunks.

As of version `sctplib-1.0.0-pre14` we also support fragmented data transmission as well as reception of fragmented chunks.

5 What is Missing

- Reception and creation of Error Chunks is not fully implemented, a number of hooks are provisioned that do not actually do anything, yet.
- Handling of ICMP messages is not currently being done. So we do NOT do MTU path discovery, as prescribed by [2].
- A few API functions are in the API, but currently do not do anything (namely **sctp_receiveUnsent**(), **sctp_receiveUnacked**() and **sctp_setPathStatus**()).
- The `CommunicationError`-notification callback is never called.
- Probably there is some more. Additions, comments, bug-reports and bug-fixes as welll as patches, patches, and patches are always welcome !

6 The API

6.1 Constants

The maximum size of an IPv4/IPv6 address string is limited to `SCTP_MAX_IP_LEN`. An endpoint may (for now) have a maximum number of addresses, which is limited to `SCTP_MAX_NUM_ADDRESSES`. The maximum size of a datagram that may be passed to the **sctp_send**() function is `SCTP_MAXIMUM_DATA_LENGTH`. Since we do NOT do proper MTU path discovery in this release, we cannot guarantee that IP fragmentation occurs, but for Ethernet hardware type and most packets, these settings will ensure that there is no fragmentation.

```
#define SCTP_MAX_IP_LEN           46
#define SCTP_MAX_NUM_ADDRESSES    20
#define SCTP_MAXIMUM_DATA_LENGTH  1400
```

The following constants have been defined for the state of an association:

```
#define SCTP_CLOSED               0
#define SCTP_COOKIE_WAIT          1
#define SCTP_COOKIE_ECHOED        2
#define SCTP_ESTABLISHED          3
#define SCTP_SHUTDOWN_PENDING     4
#define SCTP_SHUTDOWN_RECEIVED    5
#define SCTP_SHUTDOWN_SENT        6
#define SCTP_SHUTDOWNACK_SENT     7
```

When the status of a *path* changes, the following values may be assumed:

```
#define SCTP_PATH_OK              0
#define SCTP_PATH_UNREACHABLE     1
```

For setting the heartbeat (see section 6.4.19, **sctp_changeHeartBeat**()), these constants may be used:

```
#define SCTP_HEARTBEAT_ON         1
#define SCTP_HEARTBEAT_OFF        0
```

When calling **sctp_send**() (see section 6.4.7), the following constants may be used to select, how chunks are sent. For delivery with or without stream reordering, one may set:

```
#define SCTP_UNORDERED_DELIVERY   1
#define SCTP_ORDERED_DELIVERY     0
```

When calling **sctp_receive**() (see section below), the following flags can be passed along:

```
#define SCTP_MSG_DEFAULT          0x00
#define SCTP_MSG_PEEK             0x02
```

The former can be used to achieve the default behavior, i.e. data is taken from the streamengine, and copied into a user-supplied buffer. After that the data is deleted within the stream-engine. The use of the SCTP_MSG_PEEK flag allows for copying the data into a user-supplied buffer without deleting it from the stream-engine. A second call to **sctp_receive**() will then yield the same data.

To prevent messages from being bundled, the following constants may be passed as parameters to **sctp_send**():

```
#define SCTP_BUNDLING_ENABLED     0
#define SCTP_BUNDLING_DISABLED    1
```

Three more constants may be needed frequently for **sctp_send**(),

1. for path selection, if the primary path is to be taken (which is the default):

   ```
   #define SCTP_USE_PRIMARY          -1
   ```

2. this is used to have chunks have an unlimited lifetime:

   ```
   #define SCTP_INFINITE_LIFETIME    0
   ```

3. if there is no context passed along (which otherwise might be used for data retrieval - but that has not yet been implemented):

   ```
   #define SCTP_NO_CONTEXT           NULL
   ```

In order to encourage future proper and consequent use of appropriate error return codes for functions (which unfortunately up to now has only partially been implemented), a number of return codes have been defined. They are:

```
#define SCTP_SUCCESS                    0
#define SCTP_UNSPECIFIED_ERROR          -1
#define SCTP_SPECIFIC_FUNCTION_ERROR    1
```

6.2 Type Definitions

The main include file of *sctp.h* defines a number of types that are used throughout the library as well as in some of the interface functions. They will be explained in the subsequent sections.

6.2.1 SCTP_ulpCallbacks

This is a structure containing pointers to functions (used as callbacks for SCTP events that may occur and that the ULP needs to be notified of), which are all explained in detail in section 6.5. This structure is usually initialized early in the program, and then passed to the function **sctp_registerInstance**() (see section 6.4.2), which in turn registers the appropriate functions from this structure for the corresponding events.

Definition:

```
struct SCTP_ulp_Callbacks
{
   void  (*dataArriveNotif)  (unsigned int, unsigned int, unsigned int,
                              unsigned int, unsigned int,   void*);
   void  (*sendFailureNotif) (unsigned int, unsigned char *, unsigned int,
                              unsigned int *, void*);
   void  (*networkStatusChangeNotif) (unsigned int, short, unsigned short, void*);
   void* (*communicationUpNotif) (unsigned int, unsigned short,
                                  int, unsigned short, unsigned short, void*);
   void  (*communicationLostNotif)  (unsigned int, unsigned short, void*);
   void  (*communicationErrorNotif) (unsigned int, unsigned short, void*);
   void  (*restartNotif) (unsigned int, void*);
   void  (*shutdownCompleteNotif) (unsigned int, void*);
}SCTP_ulpCallbacks;
```

6.2.2 SCTP_InstanceParameters

This struct contains variables that may be set per SCTP instance (as defaults), and can subsequently set or retrieved after an instance has been registered. All associations that will be created from this instance will then inherit these parameters.
Definition:

```
struct SCTP_Instance_Parameters {
   /* the initial round trip timeout */
   unsigned int rtoInitial;
   /** the minimum timeout value */
   unsigned int rtoMin;
   /** the lifetime of a cookie */
   unsigned int validCookieLife;
   /** maximum retransmissions per association */
   unsigned int assocMaxRetransmits;
   /** maximum retransmissions per path */
   unsigned int pathMaxRetransmits;
   /** maximum initial retransmissions */
   unsigned int maxInitRetransmits;
   /** local receiver window */
   unsigned int myRwnd;
   /** delay for delayed ACK in msecs */
   unsigned int delay;
   /** per instance: for the IP type of service field. */
   unsigned char ipTos;
   /** currently unused, to limit the number of chunks
       queued in the send queue */
   unsigned int maxSendQueue;
   /** currently unused, may limit the number of chunks
       queued in the receive queue later. */
   unsigned int maxRecvQueue;
} SCTP_InstanceParameters;
```

6.2.3 SCTP_AssociationStatus

This struct contains variables that may be retrieved for each SCTP association. A number of these parameters may even be set, and are marked accordingly in the declaration.

These parameters can subsequently be set or retrieved after a `CommunicationUp` has been received. The values contained in this structure give most of the important data about the current association:

rtoInitial initial round trip timeout.

validCookieLife cookie life time.

assocMaxRetransmits maximum number of retransmission before the peer is considered unreachable.

pathMaxRetransmits maximum number of retransmission before a destination address is considered unreachable.

maxInitRetransmits maximum number of retransmissions of init and cookie messages.

state returns an unsigned integer representing the state of the association. The states have been defined as:

- `SCTP_CLOSED`
- `SCTP_COOKIE_WAIT`
- `SCTP_COOKIE_ECHOED`
- `SCTP_ESTABLISHED`
- `SCTP_SHUTDOWN_PENDING`
- `SCTP_SHUTDOWN_RECEIVED`
- `SCTP_SHUTDOWN_SENT`
- `SCTP_SHUTDOWNACK_SENT`

numberOfAddresses returns the number of destination addresses this association has, i.e. the number of possible paths.

primaryDestinationAddress returns a string containing the primary IP destination address

inStreams number of inbound streams

outStreams number of outbound streams

Definition:

```
struct SCTP_Association_Status
{
  /** (get)  */
  unsigned short state;
  /**  (get) */
  unsigned short numberOfAddresses;
  /**  (get) */
  unsigned char  primaryDestinationAddress[SCTP_MAX_IP_LEN];
  /**  (get) */
  unsigned short outStreams;
  /**  (get) */
  unsigned short inStreams;
  /**  (get/set) */
  unsigned short primaryAddressIndex;
  /**  (get) */
  unsigned int   currentReceiverWindowSize;
  /**  (get) */
  unsigned int   outstandingBytes;
  /**  (get) */
  unsigned int   noOfChunksInSendQueue;
  /**  (get) */
  unsigned int   noOfChunksInRetransmissionQueue;
  /**  (get) */
  unsigned int   noOfChunksInReceptionQueue;
  /** (get/set) the initial round trip timeout */
  unsigned int   rtoInitial;
  /** (get/set) the minimum RTO timeout */
  unsigned int   rtoMin;
  /** (get/set) the maximum RTO timeout */
  unsigned int   rtoMax;
  /** (get/set) the lifetime of a cookie */
  unsigned int   validCookieLife;
  /** (get/set) maximum retransmissions per association */
  unsigned int   assocMaxRetransmits;
  /** (get/set) maximum retransmissions per path */
  unsigned int   pathMaxRetransmits;
  /** (get/set) maximum initial retransmissions */
  unsigned int   maxInitRetransmits;
  /** (get/set) local receiver window */
  unsigned int myRwnd;
  /** (get/set) delay for delayed ACK in msecs */
  unsigned int delay;
  /** (get/set) per instance: for the IP type of service field. */
  unsigned char ipTos;
  /** currently unused, to limit the number of chunks
       queued in the send queue */
  unsigned int maxSendQueue;
   /** currently unused, may limit the number of chunks
       queued in the receive queue later. */
  unsigned int maxRecvQueue;
} SCTP_AssociationStatus;
```

6.2.4 SCTP_PathStatus

This struct contains path specific parameters. These values can only be retrieved using the function **sctp_getPathStatus**(), when the association already exists. Setting of these parameters is currently NOT imple-

mented, although the API already contains the function **sctp_setPathStatus**(), that may be used for this purpose in a later release.

This struct also contains values from the flow control module, and may thus be used to check the status of the congestion control mechanisms.

Definition:

```
struct SCTP_Path_Status
{
  unsigned char destinationAddress[SCTP_MAX_IP_LEN];
  /**  SCTP_PATH_ACTIVE  0, SCTP_PATH_INACTIVE   1    */
  short state;
  /** smoothed round trip time in msecs */
  unsigned int srtt;
  /** current rto value in msecs */
  unsigned int rto;
  /** round trip time variation, in msecs */
  unsigned int rttvar;
  /** defines the rate at which heartbeats are sent */
  unsigned int heartbeatIntervall;
  /**  congestion window size */
  unsigned int cwnd;
  /**  congestion window size 2  */
  unsigned int cwnd2;
  /**  Partial Bytes Acked  */
  unsigned int partialBytesAcked;
  /**  Slow Start Threshold  */
  unsigned int ssthresh;
  unsigned int outstandingBytesPerAddress;
  /**  Current MTU (flowcontrol) */
  unsigned int mtu;
  /** per path ? per instance ? for the IP type of service field. */
  unsigned char ipTos;
}SCTP_PathStatus;
```

6.2.5 sctp_socketCallback

Definition:

```
typedef void (*sctp_socketCallback) (int, unsigned char *, int,
                                     unsigned char[] , unsigned short);
```

A callback function type used for registering callbacks for socket events (for events POLLIN or POLLPRI). The parameters that are passed by the sctp library, when a socket has been active are (in this order):

int socket file descriptor of the socket where a datagram was received

unsigned char* pointer to the data

int length of datagram

unsigned char[] source address string (from which originated the data)

unsigned short source port (for UDP data, not SCTP data !)

6.2.6 sctp_timerCallback

Definition:

```
typedef void (*sctp_timerCallback) (unsigned int, void *, void *);
```

Defines the callback function that is called when a timer expires. Parameters:

unsigned int ID of timer

void* pointer to param1

void* pointer to param2

param1 and param2 are pointers to data that are returned to the callback function, when the timer expires. These must still exist at that time and point to valid data !

6.3 Helper Functions

The `libsctp`-library contains a few functions that influence the general control flow of a program using them, and are needed to create an application that works as expected. The important functions needed in *any* program using the `libsctp`-library are the functions **sctp_eventLoop**() and **sctp_getEvents**()

6.3.1 sctp_eventLoop()

Basically this is a wrapper to a **poll**() or **select**()-system call. The function waits until either of one events occurrs:

1. the time for which a timer event was scheduled has passed by, so the callback function belonging to that (previously registered) event is executed.
2. one of the sockets that had previously been registered with the function **sctp_registerUdpCallback**() or the raw socket that waits for incoming SCTP packets has encountered a read event, so there is data available. The appropriate function callback is then called to treat that event.

The control flow of the program is given to the callback function which may in turn register new timer events. The function returns -1 if an error ocurrs, 0 if a timeout has ocurred, or else the number of file descriptor events that have been treated.
Definition:

```
int sctp_eventLoop();
```

6.3.2 sctp_getEvents()

Basically this is a wrapper to a **poll**() or **select**()-system call (if such a syscall exists on the used OS). The function checks whether a timer event needs to be handled and does so, if the time has arisen. Then it checks via the **poll**()-system call whether events on the socket need to be handled, and does so, if there are any.

After these events have been handled, it returns the number of events handled, 0 if none had to be handled (i.e. a timer had been handled) or -1 if an error has occurred.
Definition:

```
int sctp_getEvents();
```

6.3.3 sctp_registerUdpCallback()

This function is supposed to open and bind a UDP socket listening on a port to incoming UDP pakets from a local IP address. After that it registers a callback function that is called, to dispacth UDP data which arrives for that local address. The function takes the following parameters:

`me` local address string (zero-terminated)

`my_port` the UDP port to bind (locally)

`scf` callback funtion that is called when data has arrived

It returns a new UDP socket file descriptor, or -1 if error ocurred.
Definition:

```
int sctp_registerUdpCallback(unsigned char me[],
                             unsigned short my_port,
                             sctp_socketCallback scf);
```

6.3.4 sctp_registerStdinCallback()

This function is supposed to register a callback function for catching input from the Unix STDIN file descriptor. We expect this to be useful in test programs mainly, so it is provided here for convenience. Events on this file descriptor are dispatched differently, as applications may read directly from STDIN (i.e. the callback function buffer does not contain the data from STDIN, and the registered callback function may retrieve this data with functions as **fgets**()).

`scf` callback funtion that is called when data has arrived

It returns 0, or -1 if an error ocurred.
Definition:

```
int sctp_registerStdinCallback(sctp_socketCallback scf);
```

6.3.5 sctp_startTimer()

This function adds a callback that is to be called some time from now. It realizes the timer (in an ordered list). The function takes the following parameters:

`milliseconds` action is to be started in milliseconds ms from now

`timer_cb` pointer to a function to be executed, when timer expires

`param1` pointer to be returned to the caller when timer expires

`param2` pointer to be returned to the caller when timer expires

The function returns a timer ID value, that can be used to cancel or restart this timer. **NOTE:** the pointers `param1` and `param2` exist to point to data useful for the function callback. They need to point to data that is still valid at the time the callback is activated. Do not pass pointers to temporary objects !
Definition:

```
unsigned int sctp_startTimer(unsigned int milliseconds,
                        sctp_timerCallback timer_cb, void *param1, void *param2);
```

6.3.6 sctp_stopTimer()

This function stops a previously started timer. The function takes the following parameter:

`tid` timer-id of timer to be removed

The function returns 0 on success, 1 if tid not in the list, -1 on error.
Definition:

```
int sctp_stopTimer(unsigned int tid);
```

6.3.7 sctp_restartTimer()

Restarts a timer that is currently running. The function takes the following parameters:

`timer_id` the timer id returned by start_timer

`milliseconds` action is to be taken in milliseconds ms from now

The function returns a new timer ID, zero when there is an error (i.e. the timer was not running). The function basically stops the old timer, and sets a new timer. So it is there for convenience. The timer ID will be different after calling this function !
Definition:

```
unsigned int sctp_restartTimer(unsigned int timer_id, unsigned int milliseconds);
```

6.3.8 sctp_getTime()

This helper function returns a 32 bit value representing the current time in milliseconds. Beware, this counter wraps about once per 20 days. Keep that in mind when calculating time differences ! This function may be useful, or may not be useful.
Definition:

```
unsigned int sctp_getTime(void);
```

6.4 ULP-to-SCTP

6.4.1 sctp_initLibrary()

This function will open raw sockets for capturing SCTP packets (IPv4 and if possible, IPv6, too) from the network and initialize the timer list. *It needs to be called before any other functions of the* `libsctp`*-library are called !*

Definition:

```
int sctp_initLibrary(void);
```

6.4.2 sctp_registerInstance()

sctp_registerInstance() is called to initialize one SCTP instance. An application may register several instances with different sets of callback functions, but there should not be several instances with the same port. If the port is set to zero, the instance will not be able to receive any datagrams, unless the **sctp_associate**() function is called. So it does not make sense to specify 0 as port number, unless the application is a client. A server needs a port number greater than 0 here. A client may have the `libsctp`-library choose a free source port by specifying zero.
The function takes the following parameters:

`localPort` port of this SCTP instance

`noOfInStreams` default maximum inbound stream number

`noOfOutStreams` default maximum outbound stream number

`noOfLocalAddresses` number of local addresses

`localAddressList` pointer to an array of local address-strings (zero-terminated)

`ULPcallbackFunctions` callback functions for primitives passed to ULP

The function returns the instance name of this new SCTP instance.
Definition:

```
unsigned short sctp_registerInstance(unsigned short localPort,
                                     unsigned short noOfInStreams,
                                     unsigned short noOfOutStreams,
                                     unsigned int   noOfLocalAddresses,
                                     unsigned char  localAddressList[][SCTP_MAX_IP_LEN],
                                     SCTP_ulpCallbacks ULPcallbackFunctions);
```

6.4.3 sctp_unregisterInstance()

sctp_unregisterInstance() is called to release resources used by a previously registered SCTP instance. If no instances are registered for either IPv4 or IPv6 raw sockets any more, callbacks for these sockets are also removed.
To be checked here:

- Check that all resources belonging to an instance are correctly released. What about chunks that are still in the queues ?

The function takes the following parameters:

`instance_name` name of the SCTP instance to un-register

The function returns an error code, 0 on success, -1 on error.
Definition:

```
int sctp_unregisterInstance(unsigned short instance_name);
```

6.4.4 sctp_associate()

This function is called to set up an association. It triggers sending of an INIT chunk to a server, and when the association gets established, the `CommunicationUp`-notification is called. The ULP must specify the SCTP instance which this association belongs to. The function takes the following parameters:

`SCTP_InstanceName` the SCTP instance this association belongs to. If the local port of this SCTP instance is zero, we will get a free port number assigned, else we will use the one specified in the call to **sctp_registerInstance**().

`noOfOutStreams` number of output streams the ULP would like to have.

`destinationAddress` destination address string (zero-terminated) containing the address which the INIT will be sent to.

`destinationPort` destination port

`ulp_data` void pointer, that will be returned with each callback. May be used by the ULP to find data faster.

The function returns the association ID of this association (identical with local tag), or 0 in case of failures.

Definition:

```
unsigned int sctp_associate(unsigned short SCTP_InstanceName,
                            unsigned short noOfOutStreams,
                            unsigned char  destinationAddress[],
                            unsigned short destinationPort,
                            void* ulp_data);
```

6.4.5 sctp_shutdown()

sctp_shutdown() initiates the shutdown of the specified association. Basically triggers sending of a SHUTDOWN chunk. After the shutdown procedure is completed, the `ShutdownComplete`-Notification is called. The function takes the following parameter:

`associationID` the ID of the addressed association.

The function returns 0 for success, 1 for error (assoc. does not exist).
Definition:

```
int sctp_shutdown(unsigned int associationID)
```

6.4.6 sctp_abort()

sctp_abort() initiates the abort of the specified association. The function takes the following parameter:

`associationID` the ID of the addressed association.

The function returns 0 for success, 1 for error (assoc. does not exist).
Definition:

```
int sctp_abort(unsigned int associationID)
```

6.4.7 sctp_send()

sctp_send() is used by the ULP to send data as data chunks. There are quite a few parameters that can be or must be passed along:

`associationID` the ID of the addressed association.

`streamID` identifies the stream on which the chunk is sent.

`buffer` chunk data.

`length` length of chunk data.

`protocolId` the payload protocol identifier

`path_id` index of destination address, if different from primary path. Primary Path is taken, if the constant `SCTP_USE_PRIMARY` is used here.

`context` ULP context for this data chunk, to be returned in case of errors. The constant `SCTP_NO_CONTEXT` may be used here.

`lifetime` maximum time of chunk in send queue. Use the constant `SCTP_INFINITE_LIFETIME` for infinite lifetime (the lifetime feature is not supported yet).

`unorderedDeliver` the chunk is delivered to peer ULP without resequencing. Use constants `SCTP_ORDERED_DELIVERY` or `SCTP_UNORDERED_DELIVERY`

`dontBundle` if true, chunk must not be bundled with other data chunks. Use constants `SCTP_BUNDLING_ENABLED` or `SCTP_BUNDLING_DISABLED`.

The function returns an error code: -1 for send error, 1 for association error, 0 if successful.
Definition:

```
int sctp_send(unsigned int associationID, unsigned short streamID,
              unsigned char *buffer, unsigned int length,
              unsigned int protocolId, short path_id,
              void * context, unsigned int lifetime,
              int unorderedDelivery, int dontBundle);
```

6.4.8 sctp_setPrimary()

sctp_setPrimary() changes the primary path of an association. The function takes the following parameters:

`associationID` ID of assocation.

`path_id` index to the new primary path

The function returns an error code: 0 on success, 1 on error (i.e. no assoc, path with this index does not exist etc.).
Definition:

```
short sctp_setPrimary(unsigned int associationID, short path_id)
```

6.4.9 sctp_receive()

sctp_receive() is called in response to the `DataArrive`-Notification to get the received data. You can use a flag of `SCTP_MSG_PEEK` to just peek at one part of the chunk that has arrived. A subsequent call to **sctp_receive**() with the `SCTP_MSG_DEFAULT` flag set will give you the same data. You may specify a certain length, and will get data up the maximum length of the datagram. The function takes the following parameters:

`associationID` ID of association.

`streamID` the stream on which the data chunk is received.

`buffer` pointer to where payload data of arrived chunk will be copied

`length` length of chunk data.

`flags` `SCTP_MSG_PEEK` or `SCTP_MSG_DEFAULT`

It returns 1 if association does not exist, 0 if okay.
Definition:

```
unsigned short sctp_receive(unsigned int associationID, unsigned short streamID,
                            unsigned char *buffer, unsigned int *length, unsigned int flags)
```

6.4.10 sctp_getAssocDefaults()

This function returns all the default values of an SCTP instance, i.e. it fills the **SCTP_InstanceParameters** structure. Values that are not supported yet, but already integrated in this API are set 0 by default (here: `maxSendQueue`, `maxRecvQueue`).
The function takes the following parameters:

`SCTP_InstanceName` instance name

`params` pointer to a structure, that will be filled

It returns -1 if the SCTP instance does not exist, 0 if okay.
Definition:

```
int sctp_getAssocDefaults(unsigned short SCTP_InstanceName, SCTP_InstanceParameters* params);
```

6.4.11 sctp_setAssocDefaults()

This function sets all the default values for new SCTP instances. Some values are not supported yet, but already integrated in this API (i.e. `maxSendQueue`, `maxRecvQueue`).
The function takes the following parameters:

`SCTP_InstanceName` instance name

`params` pointer to a structure, that contains new values

It returns -1 if the SCTP instance does not exist, 0 if okay.
Definition:

```
int sctp_setAssocDefaults(unsigned short SCTP_InstanceName, SCTP_InstanceParameters* params);
```

6.4.12 sctp_getAssocStatus()

This function may be used to retrieve a number of values or parameters that belong to a certain (and already existing) association. When it is called, it fills the parameters of a **SCTP_AssociationStatus** structure. Some values are not supported yet, but already integrated in this API. These will be set to 0 (i.e. `maxSendQueue`, `maxRecvQueue`).
The function takes the following parameters:

`associationID` ID of association.

`status` pointer to the structure to be filled

It returns -1 if the association does not exist, 0 if okay.
Definition:

```
int sctp_getAssocStatus(unsigned int associationID, SCTP_AssociationStatus* status);
```

6.4.13 sctp_setAssocStatus()

This function may be used to set a number of values or parameters that belong to a certain (and already existing) association. Some values are not supported yet, but already integrated in this API (i.e. `maxSendQueue`, `maxRecvQueue`).
The function takes the following parameters:

`associationID` ID of association.

`status` pointer to the structure to be filled

It returns -1 if the association does not exist, 0 if okay.
Definition:

```
int sctp_setAssocStatus(unsigned int associationID, SCTP_AssociationStatus* new_status);
```

6.4.14 sctp_getPathStatus()

This function may be used to retrieve a number of path specific values or parameters within an existing association. When it is called, it fills the parameters of a **SCTP_PathStatus** structure.
The function takes the following parameters:

`associationID` ID of association.

`path_id` index of the path for which to get the data

`status` pointer to the structure to be filled

It returns -1 if the association or the path does not exist, 0 if okay.
Definition:

```
int sctp_getPathStatus(unsigned int associationID, short path_id,
                       SCTP_PathStatus* status);
```

6.4.15 sctp_setPathStatus()

This function is currently not yet implemented, but may be implemented in a later release !
Definition:

```
int sctp_setPathStatus(unsigned int associationID, short path_id,
                       SCTP_PathStatus* new_status);
```

6.4.16 sctp_getPrimary()

This function may be conveniently used to get the path index of the current primary address. It does not give the IP address string ! The function takes the association ID as parameter, and returns 0 the primary path as result, or -1 if an error occurred.
Definition:

```
short sctp_getPrimary(unsigned int associationID);
```

6.4.17 sctp_setPrimary()

This function sets a new primary address. It is given the index of the new primary path and the associaton ID as parameter, and returns 0 on success, or 1 on error.
Definition:

```
short sctp_setPrimary(unsigned int associationID, short path_id);
```

6.4.18 sctp_getSrttReport()

Returns the smoothed round trip time in milliseconds for a certain destination path within an association, or zero on error.
Definition:

```
unsigned int sctp_getSrttReport(unsigned int associationID, short path_id);
```

6.4.19 sctp_changeHeartBeat()

sctp_changeHeartBeat() turns the heartbeat of an association on or off, and sets the interval, if it is turned on. Our `libsctp`-library implementation is currently missing the jitter, that is required by the RFC [2]. The function takes the following parameters:

`associationID` ID of assocation.

`path_id` index of the address for which HB is turned on/off.

`heartbeatON` turn heartbeat on or off. Use constants `SCTP_HEARTBEAT_ON` or `SCTP_HEARTBEAT_OFF`.

`timeIntervall` heartbeat time intervall in milliseconds

The function returns an error code, 0 for success, 1 if association or destination address do not exist.
Definition:

```
int sctp_changeHeartBeat(unsigned int associationID, short path_id,
                         int heartbeatON, unsigned int timeIntervall)
```

6.4.20 sctp_requestHeartbeat()

sctp_requestHeartbeat() sends a heartbeat to the given address of an association. This is an explicit request from the ULP to the SCTP instance to perform an on-demand heartbeat. The function takes the following parameters:

`associationID` ID of assocation.

`path_id` destination address to which the heartbeat shall be sent.

Function returns error code (0 == success, 1 == error).
Definition:

```
int sctp_requestHeartbeat(unsigned int associationID, short path_id);
```

6.4.21 sctp_setFailureThreshold()

Is used to set the threshold for retransmissions of the given association. If the threshold is exceeded, the the destination address is considered unreachable. The function takes the following parameters:

`associationID` ID of assocation.

`pathMaxRetransmissions` new threshold for retransmissions.

Function returns 0 on success, 1 otherwise.
Definition:

```
int sctp_setFailureThreshold(unsigned int associationID,
                             unsigned short pathMaxRetransmissions)
```

6.4.22 sctp_receiveUnsent()

sctp_receiveUnsent() is currently NOT implemented ! It will return messages that could not be sent for some reason, e.g. because association was aborted before they could be transmitted for the first time. The function takes the following parameters:

`associationID` ID of assocation.

`buffer` pointer to a buffer that the application needs to pass. Data is copied there.

`length` pointer to size of the buffer passed by application. Contains the actual length of the data chunk after the call returns.

`streamID` pointer to the stream id, where data should have been sent.

`streamSN` pointer to stream sequence number of the data chunk that was not sent.

`protocolId` pointer to the protocol ID of the unsent chunk

Function currently not implemented, so it returns -1, always.
Definition:

```
int sctp_receiveUnsent(unsigned int associationID, unsigned char *buffer,
                       unsigned int *length, unsigned short *streamID,
                       unsigned short *streamSN,unsigned int* protocolId);
```

6.4.23 sctp_receiveUnacked()

sctp_receiveUnacked() is currently NOT implemented ! Will return messages that were sent, but have not been acknowlegded by the peer before the association was terminated (or aborted). The function takes the following parameters:

`associationID` ID of assocation.

`buffer` pointer to a buffer that the application needs to pass. The data retrieved is copied there.

`length` pointer to the size of the buffer passed by application. After the function returns, the value is changed to the actual length of the chunk.

`streamID` pointer to the stream id, where data should have been sent.

`streamSN` pointer to stream sequence number of the data chunk that was not acked

`protocolId` pointer to the protocol ID of the unacked chunk

Function currently not implemented, so it returns -1, always.
Definition:

```
int sctp_receiveUnacked(unsigned int associationID, unsigned char *buffer,
                        unsigned int *length, unsigned short *streamID,
                        unsigned short *streamSN,unsigned int* protocolId);
```

6.4.24 sctp_deleteAssociation()

sctp_deleteAssociation() is called to remove all the data structures belonging to an association. It should be called some time after a `ShutdownComplete` or a `CommunicationLost`-notification has been received, and the ULP has retrieved all the data that may still be in the queues using the (currently unimplemented) functions **sctp_receiveUnacked**() and **sctp_receiveUnsent**(). A call to this function finally frees all association resources.
The function returns 0 on success, 1 if association does not exist, and -1 if association is still active, and not marked *deleted*.
Definition:

```
int sctp_deleteAssociation(unsigned int associationID);
```

6.5 SCTP-to-ULP

All notifcations are realized via callbacks, i.e. upon registering a new SCTP instance, the ULP has to pass a struct containing pointers to a set of functions, that will be called when the respective event occurs, with appropriate parameters.

The `CommunicationUp`-notification may return a pointer to a data structure that the upper layer is interested in. This pointer is subsequently registered within the new association structure, and passed along transparently with all callbacks.

6.5.1 DataArrive Notification

Indicates that new data has arrived from peer (chapter 10.2.A). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned int stream ID

unsigned int length of data

unsigned int protocol ID

unsigned int unordered flag (uses constants SCTP_UNORDERED_DELIVERY==1,or SCTP_ORDERED_DELIVERY==0 for normal numbered chunk)

void* pointer to upper layer data

Definition:

```
void (*dataArriveNotif) (unsigned int, unsigned int, unsigned int,
                         unsigned int, unsigned int, void*);
```

6.5.2 Send Failure Indication

Indicates a send failure, i.e. the data could not be sent for some reason (chapter 10.2.B). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned char * pointer to data that was not sent

unsigned int length of data

unsigned int * pointer to context from sctp_send

void* pointer to upper layer data

Definition:

```
void (*sendFailureNotif) (unsigned int, unsigned char *,
                          unsigned int, unsigned int *, void*);
```

6.5.3 NetworkStatusChange Indication

Indicates a change of network status, i.e. a path has changed from state SCTP_PATH_OK to SCTP_PATH_UNREACHABLE, or vice versa. (chapter 10.2.C). If there is only *one* active path left, which turns INACTIVE, a `CommunicationLost`-notification is given instead of a `NetworkStatusChange`. The parameters passed with this callback are (in this order):

unsigned int association ID

short index of the destination address concerned

unsigned short new path state (uses constants `SCTP_PATH_OK`, `SCTP_PATH_UNREACHABLE`)

void* pointer to upper layer data

Definition:

```
void (*networkStatusChangeNotif) (unsigned int, short, unsigned short, void*);
```

6.5.4 CommunicationUp Notification

Indicates that an association has been successfully established (chapter 10.2.D). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned short status, type of event; the following events are defined:

- `SCTP_COMM_UP_RECEIVED_VALID_COOKIE`
- `SCTP_COMM_UP_RECEIVED_COOKIE_ACK`
- `SCTP_COMM_UP_RECEIVED_COOKIE_RESTART`

int number of destination addresses

unsigned short number input streamns

unsigned short number output streams

void* pointer to upper layer data

The function (i.e. the upper layer !) may return a pointer that the upper layer is intersted in. This pointer will then be passed along with all subsequent notification calls.
Definition:

```
void* (*communicationUpNotif) (unsigned int, unsigned short, int,
                               unsigned short, unsigned short, void*);
```

6.5.5 CommunicationLost Notification

Indicates that the association with the peer was closed for some reason (chapter 10.2.E). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned short status, type of event; the following events are defined:

- `SCTP_COMM_LOST_ABORTED`
  ULP has called **sctp_abort**(), and an ABORT chunk has been sent out, or we have received an ABORT chunk from the peer. The association has been terminated.
- `SCTP_COMM_LOST_ENDPOINT_UNREACHABLE`
  All paths have turned unreachable.
- `SCTP_COMM_LOST_EXCEEDED_RETRANSMISSIONS`
  The number of retransmissions has been exceeded in normal transmission of data or heartbeat chunks, or the number of retransmissions of INIT, COOKIE_ECHO, SHUTDOWN or SHUTDOWN_COMPLETE chunks have been exceeded.
- `SCTP_COMM_LOST_NO_TCB`
  SCTP instance has received SHUTDOWN chunk from a peer in state `SCTP_CLOSED`, or a SHUTDOWN_ACK in any of the states `SCTP_CLOSED`, `SCTP_COOKIE_WAIT` or `SCTP_COOKIE_ECHOED`.
- `SCTP_COMM_LOST_ZERO_STREAMS`
  SCTP instance has received an otherwise valid INIT chunk requesting either 0 inbound or 0 outbound streams. We send an ABORT, and terminate the association, if it exists.
- `SCTP_COMM_LOST_FAILURE`
  Currently unsused.

void* pointer to upper layer data

Definition:

```
void (*communicationLostNotif) (unsigned int, unsigned short, void*);
```

6.5.6 CommunicationError Notification

Indicates that communication had an error (chapter 10.2.F). Currently not implemented !

unsigned int association ID

unsigned short status, type of error

void* pointer to upper layer data

Definition:

```
void (*communicationErrorNotif) (unsigned int, unsigned short, void*);
```

6.5.7 Restart Indication

Indicates that a restart has occurred (chapter 10.2.G). The notification has been added to our implementation lately, but the restart scenario is still somewhat untested. So we need feedback here !

unsigned int association ID

void* pointer to upper layer data

Definition:

```
void (*restartNotif) (unsigned int, void*);
```

6.5.8 ShutdownComplete Indication

Indicates that a Shutdown has been successfully completed (chapter 10.2.H).

unsigned int association ID

void* pointer to upper layer data

Definition:

```
void (*shutdownCompleteNotif) (unsigned int, void*);
```

7 Example Programs

7.1 A Discard Server

The discard-server is a very simple example for an application and the appropriate use of the `libsctp`-library . The **main**() function calls **getArgs**(), which parses some command line options, that allow for changing default behavior of the discard server to verbose or very verbose (respective `-v` or `-V` options) and for specifying a number of source addresses.

Then, **main**() assigns all callback functions to a **SCTP_ulpCallbacks** variable, and calls upon **sctp_registerInstance**(), passing a number of default parameters (e.g. port number 9). Finally, the function **sctp_eventLoop**() is executed in a loop that only terminates if a severe error is encountered.

The specified callback functions do nothing except logging the callback events in verbose mode, and put received data into a local buffer that is overwritten each time data arrives. So any data that arrives is discarded.

7.2 An Echo Server

The echo server is structured similarly to the discard server (see section 7.1). The main difference of their functionalities is, of course, in the callback functions. The general control flow is as follows:

the **main**() function calls **getArgs**(), which parses some command line options, that allow for changing default behavior of the echo server to verbose or very verbose (respective `-v` or `-V` options) and for specifying a number of source addresses.

Then, **main**() assigns all callback functions to a **SCTP_ulpCallbacks** variable, and calls upon **sctp_registerInstance**(), passing a number of default parameters (e.g. port number 7). Finally, the function **sctp_eventLoop**() is executed in a loop that only terminates if a severe error is encountered. The maximum number of associations the echo server can handle is limited by the constant `MAXIMUM_NUMBER_OF_ASSOCIATIONS`.

The `CommunicationUp` callback function limits this number of existing associations in that it calls **sctp_abort**() when the maximum number of existing associations is exceeded.

The interesting callback function is actually that for the `DataArrive` notification. It receives the arrived data into a local buffer using the function **sctp_receive**(), and calls **sctp_send**() to echo the data back to the peer.

The callback for the `NetworkStatusChange` makes sure that as long as there is an active path left in the association, the primary path is an active path.

7.3 An Echo Tool

The echo tool works very similar to the echo server, but has several modes of operation. In the first, executed after using the `-b` switch, it echoes incoming data back to all existing associations. Thus, it may be thought of as a multicast server.

In the second, executed after using the `-d` switch, it sends a configurable number of packets to a target endpoint. If this target endpoint runs an echo server, the two endpoints will start echoing the initial packets to each other. This supposedly generates high network load.

Finally, both modes may be combined. The echo client takes the following command line options:

`-s source address` an IP source address belonging to this host

`-d destination` IP address of a destination (which may be running an echo server)

`-b` send back incoming data on all existing associations

`-l length` length of payload in bytes (must be less than SCTP_MAXIMUM_DATA_LENGTH==1400 bytes)

`-n packets` number of initial packets

`-t IP_TOS_BYTE` sets the IP_TOS_BYTE in the IP header. These values should be 2, 4, 8 or 16

`-m` print number of received bytes an chunks per period of time

`-p period` set period for the measurements in milliseconds

`-v` verbose execution

`-V` very verbose execution

References

[1] Stewart, R.R. and Xie, Q. and others: **RFC 2960 - Stream Control Transmission Protocol**, IETF, Network Working Group, October 2000

[2] Stewart, R.R.: **RFC 4960 - Stream Control Transmission Protocol**, IETF, Network Working Group, September 2007