9.1 Introduction

SCTP is a newer transport protocol, standardized in the IETF in 2000 (compared with TCP, which was standardized in 1981). It was first designed to meet the needs of the growing IP telephony market; in particular, transporting telephony signaling across the Internet. The requirements it was designed to fulfill are described in RFC 2719 [Ong et al. 1999]. SCTP is a reliable, message-oriented protocol, providing multiple streams between endpoints and transport-level support for multihoming. Since it is a newer transport protocol, it does not have the same ubiquity as TCP or UDP; however, it provides some new features that may simplify certain application designs. We will discuss the reasons to consider using SCTP instead of TCP in Section 23.12.

Although there are some fundamental differences between SCTP and TCP, the *one-to-one* interface for SCTP provides very nearly the same application interface as TCP. This allows for trivial porting of applications, but does not permit use of some of SCTP's advanced features. The *one-to-many* interface provides full support for these features, but may require significant retooling of existing applications. The one-to-many interface is recommended for most new applications developed for SCTP.

This chapter describes additional elementary socket functions that can be used with SCTP. We first describe the two different interface models that are available to the application developer. We will develop a version of our echo server using the one-to-many model in Chapter 10. We also describe the new functions available for and used exclusively with SCTP. We look at the `shutdown` function and how its use with SCTP differs from TCP. We then briefly cover the use of *notifications* in SCTP. Notifications allow an application to be informed of significant protocol events other than the arrival of user data. We will see an example of how to use notifications in Section 23.4.

Since SCTP is a newer protocol, the interface for all its features has not yet completely stabilized. As of this writing, the interfaces described are believed to be stable, but are not yet as ubiquitous as the rest of the sockets API. Users of applications designed to use SCTP exclusively may need to be prepared to install kernel patches or otherwise upgrade their operating system, and applications which need to be ubiquitous need to be able to use TCP if SCTP is not available on the system they are running on.

9.2 Interface Models

There are two types of SCTP sockets: a *one-to-one* socket and a *one-to-many* socket. A one-to-one socket corresponds to exactly one SCTP association. (Recall from Section 2.5 that an SCTP association is a connection between two systems, but may involve more than two IP addresses due to multihoming.) This mapping is similar to the relationship between a TCP socket and a TCP connection. With a one-to-many socket, several SCTP associations can be active on a given socket simultaneously. This mapping is similar to the manner in which a UDP socket bound to a particular port can receive interleaved datagrams from several remote UDP endpoints that are all simultaneously sending data.

When deciding which style of interface to use, the application needs to consider several factors, including:

- What type of server is being written, *iterative* or *concurrent*?
- How many socket descriptors does the server wish to manage?
- Is it important to optimize the association setup to enable data on the third (and possibly fourth) packet of the four-way handshake?
- How much connection state does the application wish to maintain?

> When the sockets API for SCTP was under development, different terminology was used for the two styles of sockets, and readers may sometimes encounter these older terms in documentation or source code. The original term for the one-to-one socket was a "TCP-style" socket, and the original term for a one-to-many socket was a "UDP-style" socket.
>
> These style terms were later dropped because they tended to cause confusion by creating expectations that SCTP would behave more like TCP or UDP, depending on which style of socket was used. In fact, these terms referred to only one aspect of the differences between TCP and UDP sockets (i.e., whether a socket supports multiple concurrent transport-layer associations). The current terminology ("one-to-one" versus "one-to-many") focuses our attention on the key difference between the two socket styles. Finally, note that some writers use the term "many-to-one" instead of "one-to-many"; the terms are interchangeable.

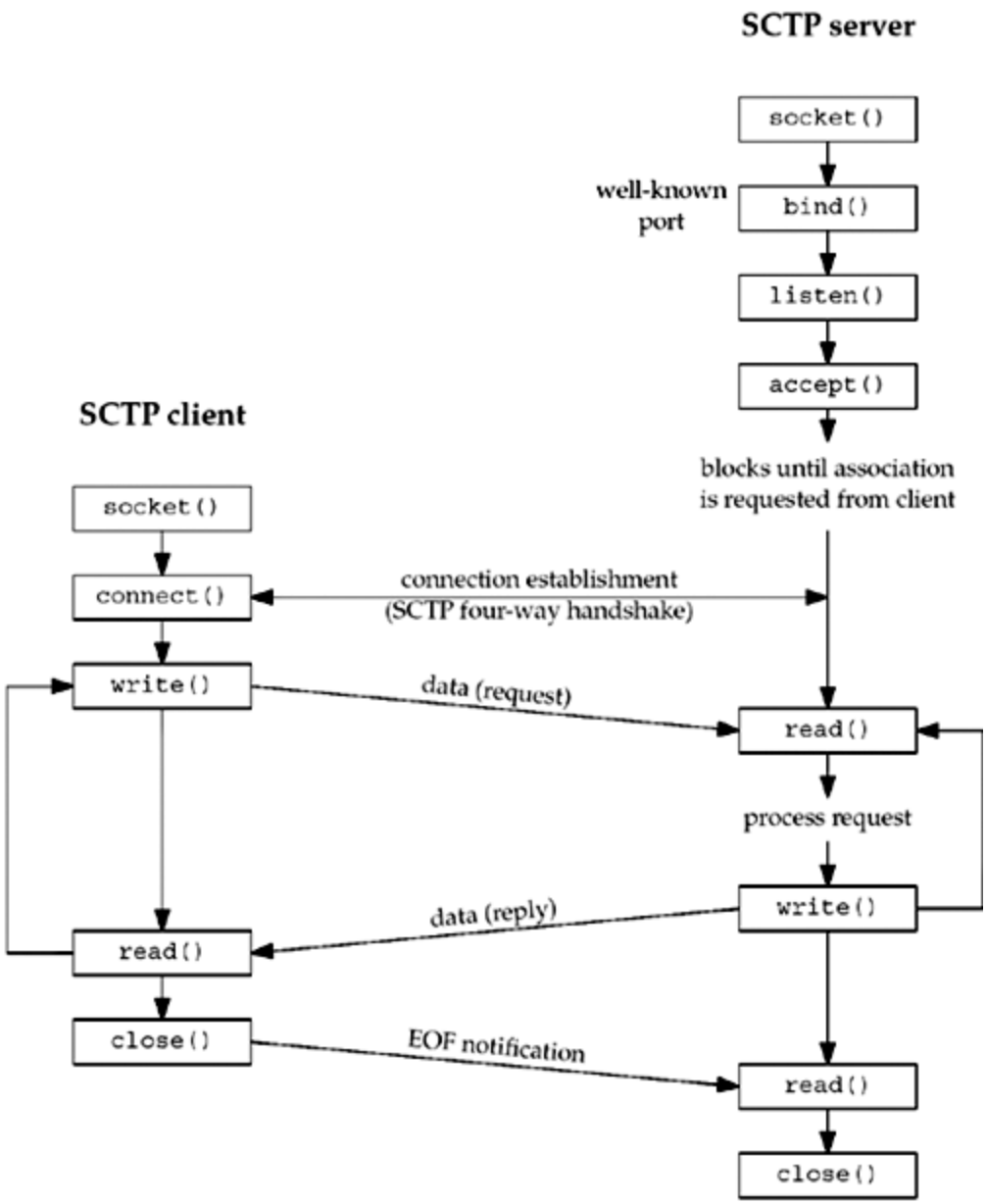
The One-to-One Style

The one-to-one style was developed to ease the porting of existing TCP applications to SCTP. It provides nearly an identical model to that described in Chapter 4. There are some differences one should be aware of, especially when porting existing TCP applications to SCTP using this style.

1. Any socket options must be converted to the SCTP equivalent. Two commonly found options are `TCP_NODELAY` and `TCP_MAXSEG`. These can be easily mapped to `SCTP_NODELAY` and `SCTP_MAXSEG`.
2. SCTP preserves message boundaries; thus, application-layer message boundaries are not required. For example, an application protocol based on TCP might do a `write()` system call to write a two-byte message length field, *x*, followed by a `write()` system call that writes *x* bytes of data. However, if this is done with SCTP, the receiving SCTP will receive two separate messages (i.e., the read call will return twice: once with a two-byte message, and then again with an *x* byte message).
3. Some TCP applications use a half-close to signal the end of input to the other side. To port such applications to SCTP, the application-layer protocol will need to be rewritten so that the application signals the end of input in the application data stream.
4. The `send` function can be used in the normal fashion. For the `sendto` and `sendmsg` functions, any address information included is treated as an override of the primary destination address (see Section 2.8).

A typical user of the one-to-one style will follow the timeline shown in Figure 9.1. When the server is started, it opens a socket, binds to an address, and waits for a client connection with the `accept` system call. Sometime later, the client is started, it opens a socket, and initiates an association with the server. We assume the client sends a request to the server, the server processes the request, and the server sends back a reply to the client. This cycle continues until the client initiates a shutdown of the association. This action closes the association, whereupon the server either exits or waits for a new association. As can be seen by comparison to a typical TCP exchange, an SCTP one-to-one socket exchange proceeds in a fashion similar to that shown in Figure 4.1.

Figure 9.1. Socket functions for SCTP one-to-one style.



A one-to-one-style SCTP socket is an IP socket (family `AF_INET` or `AF_INET6`), with type `SOCK_STREAM` and protocol `IPPROTO_SCTP`.

The One-to-Many Style

The one-to-many style provides an application writer the ability to write a server without managing a large number of socket descriptors. A single socket descriptor will represent multiple associations, much the same way that a UDP socket can receive messages from multiple clients. An association identifier is used to identify a single association on a one-to-many-style socket. This association identifier is a value of type `sctp_assoc_t`; it is normally an integer. It is an opaque value; an application should not use an association identifier that it has not previously been given by the kernel. Users of the one-to-many style should keep the following issues in mind:

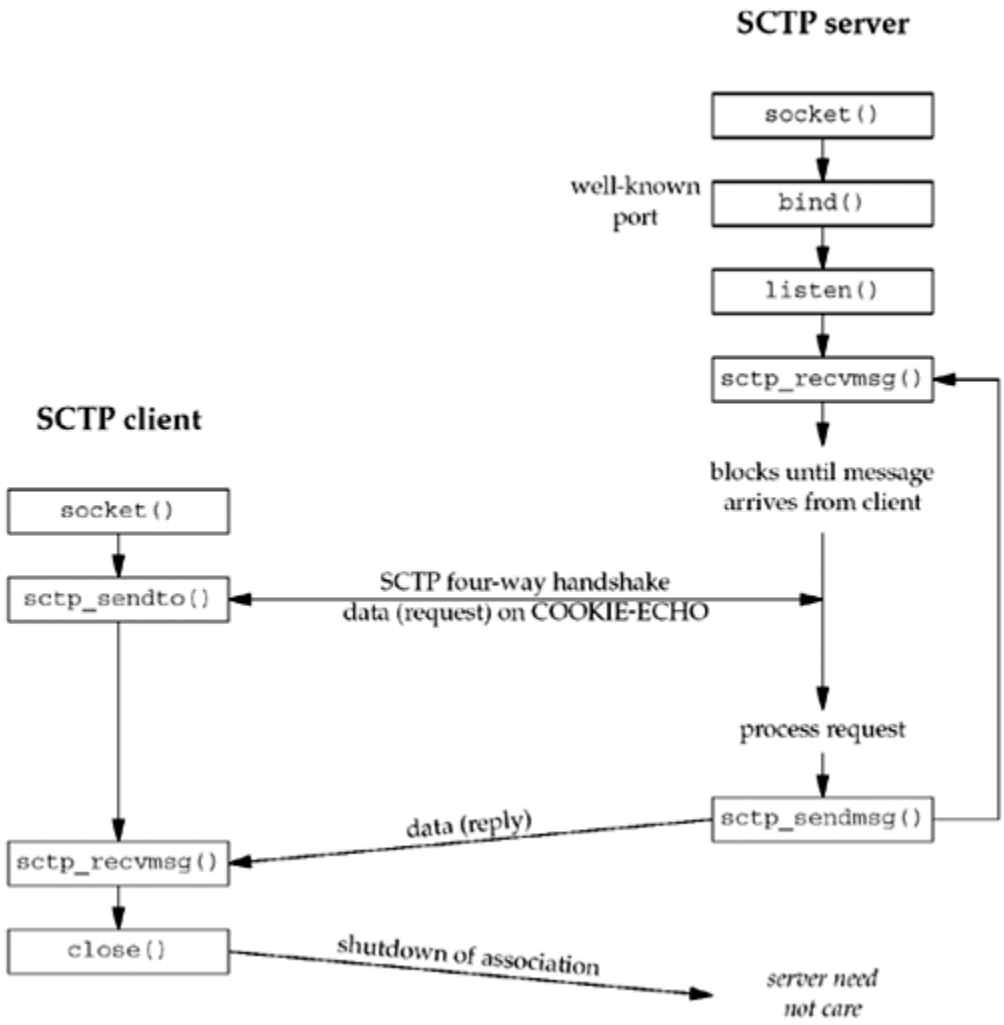
1. When the client closes the association, the server side will automatically close as well, thus removing any state for the association inside the kernel.
2. Using the one-to-many style is the only method that can be used to cause data to be piggybacked on the third or fourth packet of the four-way handshake (see Exercise 9.3).
3. Any `sendto`, `sendmsg`, or `sctp_sendmsg` to an address for which an association does not yet exist will cause an active open to be attempted, thus creating (if successful) a new association with that address. This behavior occurs even if the application doing the send has called the `listen` function to request a passive open.
4. The user must use the `sendto`, `sendmsg`, or `sctp_sendmsg` functions, and may not use the `send` or `write` function. (If the `sctp_peeloff` function is used to create a one-to-one-style socket, `send` or `write` may be used on it.)
5. Anytime one of the send functions is called, the primary destination address that was chosen by the system at association initiation time (Section 2.8) will be used unless the `MSG_ADDR_OVER` flag is set by the caller in a supplied `sctp_sndrcvinfo` structure. To supply this, the caller needs to use the `sendmsg` function with ancillary data, or the `sctp_sendmsg` function.

6. Association events (one of a number of SCTP notifications discussed in Section 9.14) may be enabled, so if an application does not wish to receive these events, it should disable them explicitly using the `SCTP_EVENTS` socket option. By default, the only event that is enabled is the `sctp_data_io_event`, which provides ancillary data to the `recvmsg` and `sctp_recvmsg` call. This default setting applies to both the one-to-one and one-to-many style.

   When the SCTP sockets API was first developed, the one-to-many-style interface was defined to have the association notification turned on by default as well. Later versions of the API document have since disabled all notifications except the `sctp_data_io_event` for both the one-to-one- and one-to-many-style interface. However not all implementations may have this behavior. It is always good practice for an application writer to explicitly disable (or enable) the notifications that are unwanted (or desired). This explicit approach assures the developer that the expected behavior will result no matter which OS the code is ported to.

A typical one-to-many style timeline is depicted in Figure 9.2. First, the server is started, creates a socket, binds to an address, calls `listen` to enable client associations, and calls `sctp_recvmsg`, which blocks waiting for the first message to arrive. A client opens a socket and calls `sctp_sendto`, which implicitly sets up the association and piggybacks the data request to the server on the third packet of the four-way handshake. The server receives the request, and processes and sends back a reply. The client receives the reply and closes the socket, thus closing the association. The server loops back to receive the next message.

Figure 9.2. Socket functions for SCTP one-to-many style.



This example shows an iterative server, where (possibly interleaved) messages from many associations (i.e., many clients) can be processed by a single thread of control. With SCTP, a one-to-many socket can also be used in conjunction with the `sctp_peeloff` function (see Section 9.12) to allow the iterative and concurrent server models to be combined as follows:

1. The `sctp_peeloff` function can be used to peel off a particular association (for example, a long-running session) from a one-to-many socket into its own one-to-one socket.
2. The one-to-one socket of the extracted association can then be dispatched to its own thread or forked process (as in the concurrent model).
3. Meanwhile, the main thread continues to handle messages from any remaining associations in an iterative fashion on the original socket.

A one-to-many-style SCTP socket is an IP socket (family `AF_INET` or `AF_INET6`) with type `SOCK_SEQPACKET` and protocol `IPPROTO_SCTP`.

9.3 sctp_bindx Function

An SCTP server may wish to bind a subset of IP addresses associated with the host system. Traditionally, a TCP or UDP server can bind one or all addresses on a host, but they cannot bind a subset of addresses. The sctp_bindx function provides more flexibility by allowing an SCTP socket to bind a particular subset of addresses.

```
#include <netinet/sctp.h>

int sctp_bindx(int sockfd, const struct sockaddr *addrs, int addrcnt, int flags);
```

Returns: 0 if OK, −1 on error

The *sockfd* is a socket descriptor returned by the socket function. The second argument, *addrs*, is a pointer to a packed list of addresses. Each socket address structure is placed in the buffer immediately following the preceding socket address structure, with no intervening padding. See Figure 9.4 for an example.

The number of addresses being passed to sctp_bindx is specified by the *addrcnt* parameter. The *flags* parameter directs the sctp_bindx call to perform one of the two actions shown in Figure 9.3.

Figure 9.3. *flags* used with sctp_bindx function.

flags	Description
SCTP_BINDX_ADD_ADDR	Add the address(es) to the socket
SCTP_BINDX_REM_ADDR	Remove the address(es) from the socket

The sctp_bindx call can be used on a bound or unbound socket. For an unbound socket, a call to sctp_bindx will bind the given set of addresses to the socket descriptor. If sctp_bindx is used on a bound socket, the call can be used with SCTP_BINDX_ADD_ADDR to associate additional addresses with the socket descriptor or with SCTP_BINDX_REM_ADDR to remove a list of addresses associated with the socket descriptor. If sctp_bindx is performed on a listening socket, future associations will use the new address configuration; the change does not affect any existing associations. The two flags passed to sctp_bindx are mutually exclusive; if both are given, sctp_bindx will fail, returning the error code EINVAL. The port number in all the socket address structures must be the same and must match any port number that is already bound; if it doesn't, then sctp_bindx will fail, returning the error code EINVAL.

If an endpoint supports the dynamic address feature, a call to sctp_bindx with the SCTP_BINDX_REM_ADDR or SCTP_BINDX_ADD_ADDR flag will cause the endpoint to send an appropriate message to the peer to change the peer's address lists. Since adding and removing addresses from a connected association is optional functionality, implementations that do not support this functionality will return EOPNOTSUPP. Note that both ends of an association must support this feature for proper operation. This feature can be useful if the system supports dynamic provisioning of interfaces; for example, if a new Ethernet interface is brought up, the application can use SCTP_BINDX_ADD_ADDR to start using the additional interface on an existing connection.

9.4 sctp_connectx Function

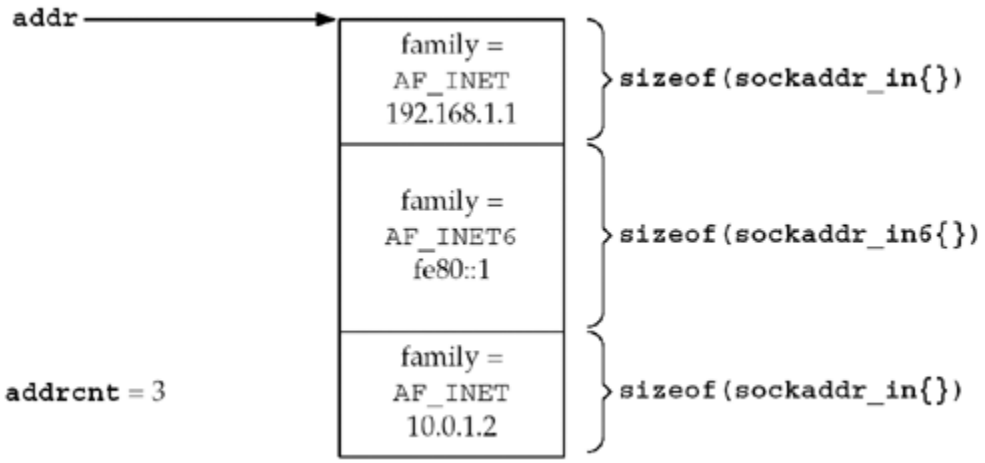
```
#include <netinet/sctp.h>

int sctp_connectx(int sockfd, const struct sockaddr *addrs, int addrcnt);
```

Returns: 0 for success, −1 on error

The sctp_connectx function is used to connect to a multihomed peer. We specify *addrcnt* addresses, all belonging to the same peer, in the *addrs* parameter. The *addrs* parameter is a packed list of addresses, as in Figure 9.4. The SCTP stack uses one or more of the given addresses for establishing the association. All the addresses listed in *addrs* are considered to be valid, confirmed addresses.

Figure 9.4. Packed address list format for SCTP calls.



9.5 `sctp_getpaddrs` Function

The `getpeername` function was not designed with the concept of a multihoming-aware transport protocol; when using SCTP, it only returns the primary address. When all the addresses are required, the `sctp_getpaddrs` function provides a mechanism for an application to retrieve all the addresses of a peer.

```
#include <netinet/sctp.h>

int sctp_getpaddrs(int sockfd, sctp_assoc_t id, struct sockaddr **addrs);
```

Returns: the number of peer addresses stored in *addrs*, −1 on error

The *sockfd* parameter is the socket descriptor returned by the `socket` function. The *id* is the association identification for a one-to-many-style socket. If the socket is using the one-to-one style, the *id* field is ignored. *addrs* is the address of a pointer that `sctp_getpaddrs` will fill in with a locally allocated, packed list of addresses. See Figures 9.4 and 23.12 for details on the structure of this return value. The caller should use `sctp_freepaddrs` to free resources allocated by `sctp_getpaddrs` when finished with them.

9.6 `sctp_freepaddrs` Function

The `sctp_freepaddrs` function frees resources allocated by the `sctp_getpaddrs` function. It is called as follows:

```
#include <netinet/sctp.h>

void sctp_freepaddrs(struct sockaddr *addrs);
```

addrs is the pointer to the array of addresses returned by `sctp_getpaddrs`.

9.7 `sctp_getladdrs` Function

The `sctp_getladdrs` function can be used to retrieve the local addresses that are part of an association. This function is often necessary when a local endpoint wishes to know exactly which local addresses are in use (which may be a proper subset of the system's addresses).

```
#include <netinet/sctp.h>

int sctp_getladdrs(int sockfd, sctp_assoc_t id, struct sockaddr **addrs);
```

Returns: the number of local addresses stored in *addrs*, −1 on error

The *sockfd* is the socket descriptor returned by the `socket` function. *id* is the association identification for a one-to-many-style socket. If the socket is using the one-to-one style, the *id* field is ignored. The *addrs* parameter is an address of a pointer that `sctp_getladdrs` will fill in with a locally allocated, packed list of addresses. See Figures 9.4 and 23.12 for details on the structure of this return value. The caller should use `sctp_freeladdrs` to free resources allocated by `sctp_getladdrs` when finished with them.

9.8 `sctp_freeladdrs` Function

The `sctp_freeladdrs` function frees resources allocated by the `sctp_getladdrs` function. It is called as follows:

```
#include <netinet/sctp.h>

void sctp_freeladdrs(struct sockaddr *addrs);
```

addrs is the pointer to the array of addresses returned by `sctp_getladdrs`.

9.9 sctp_sendmsg Function

An application can control various features of SCTP by using the sendmsg function along with ancillary data (described in Chapter 14). However, because the use of ancillary data may be inconvenient, many SCTP implementations provide an auxiliary library call (possibly implemented as a system call) that eases an application's use of SCTP's advanced features. The call takes the following form:

```
ssize_t sctp_sendmsg(int sockfd, const void *msg, size_t msgsz, const struct sockaddr *to, socklen_t tolen, uint32_t ppid,
uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t context);
```

Returns: the number of bytes written, −1 on error

The user of sctp_sendmsg has a greatly simplified sending method at the cost of more arguments. The *sockfd* field holds the socket descriptor returned from a socket system call. The *msg* field points to a buffer of *msgsz* bytes to be sent to the peer endpoint *to*. The *tolen* field holds the length of the address stored in *to*. The *ppid* field holds the pay-load protocol identifier that will be passed with the data chunk. The *flags* field will be passed to the SCTP stack to identify any SCTP options; valid values for this field may be found in Figure 7.16.

A caller specifies an SCTP stream number by filling in the *stream*. The caller may specify the lifetime of the message in milliseconds in the *lifetime* field, where 0 represents an infinite lifetime. A user context, if any, may be specified in *context*. A user context associates a failed message transmission, received via a message notification, with some local application-specific context. For example, to send a message to stream number 1, with the send flags set to MSG_PR_SCTP_TTL, the lifetime set to 1000 milliseconds, a payload protocol identifier of 24, and a context of 52, a user would formulate the following call:

```
ret = sctp_sendmsg(sockfd,
                   data, datasz, &dest, sizeof(dest),
                   24, MSG_PR_SCTP_TTL, 1, 1000, 52);
```

This approach is much easier than allocating the necessary ancillary data and setting up the appropriate structures in the msghdr structure. Note that if an implementation maps the sctp_sendmsg to a sendmsg function call, the *flags* field of the sendmsg call is set to 0.

9.10 sctp_recvmsg Function

Just like sctp_sendmsg, the sctp_recvmsg function provides a more user-friendly interface to the advanced SCTP features. Using this function allows a user to retrieve not only its peer's address, but also the msg_flags field that would normally accompany the recvmsg function call (e.g., MSG_NOTIFICATION, MSG_EOR, etc.). The function also allows the user to retrieve the sctp_sndrcvinfo structure that accompanies the message that was read into the message buffer. Note that if an application wishes to receive sctp_sndrcvinfo information, the sctp_data_io_event must be subscribed to with the SCTP_EVENTS socket option (ON by default). The sctp_recvmsg function takes the following form:

```
ssize_t sctp_recvmsg(int sockfd, void *msg, size_t msgsz, struct sockaddr *from, socklen_t *fromlen, struct
sctp_sndrcvinfo *sinfo, int *msg_flags);
```

Returns: the number of bytes read, −1 on error

On return from this call, *msg* is filled with up to *msgsz* bytes of data. The message sender's address is contained in *from*, with the address size filled in the *fromlen* argument. Any message flags will be contained in the *msg_flags* argument. If the notification sctp_data_io_event has been enabled (the default), the sctp_sndrcvinfo structure will be filled in with detailed information about the message as well. Note that if an implementation maps the sctp_recvmsg to a recvmsg function call, the *flags* field of the call will be set to 0.

9.11 `sctp_opt_info` Function

The `sctp_opt_info` function is provided for implementations that cannot use the `getsockopt` functions for SCTP. This inability to use the `getsockopt` function is because some of the SCTP socket options, for example, `SCTP_STATUS`, need an in-out variable to pass the association identification. For systems that cannot provide an in-out variable to the `getsockopt` function, the user will need to use `sctp_opt_info`. For systems like FreeBSD that do allow in-out variables in the socket option call, the `sctp_opt_info` call is a library call that repackages the arguments into the appropriate `getsockopt` call. For portability's sake, applications should use `sctp_opt_info` for all the options that require in-out variables (Section 7.10).

The call has the following format:

```
int sctp_opt_info(int sockfd, sctp_assoc_t assoc_id, int opt void *arg, socklen_t *siz);
```

Returns: 0 for success, −1 on error

sockfd is the socket descriptor that the user would like the socket option to affect. *assoc_id* is the identification of the association (if any) on which the user is performing the option. *opt* is the socket option (as defined in Section 7.10) for SCTP. *arg* is the socket option argument, and *siz* is a pointer to a *socklen_t* which holds the size of the argument.

9.12 `sctp_peeloff` Function

As previously mentioned, it is possible to extract an association contained by a one-to-many socket into an individual one-to-one-style socket. The semantics are much like the `accept` function call with an additional argument. The caller passes the *sockfd* of the one-to-many socket and the association identification *id* that is being extracted. At the completion of the call, a new socket descriptor is returned. This new descriptor will be a one-to-one-style socket descriptor with the requested association. The function takes the following form:

```
int sctp_peeloff(int sockfd, sctp_assoc_t id);
```

Returns: a new socket descriptor on success, −1 on error

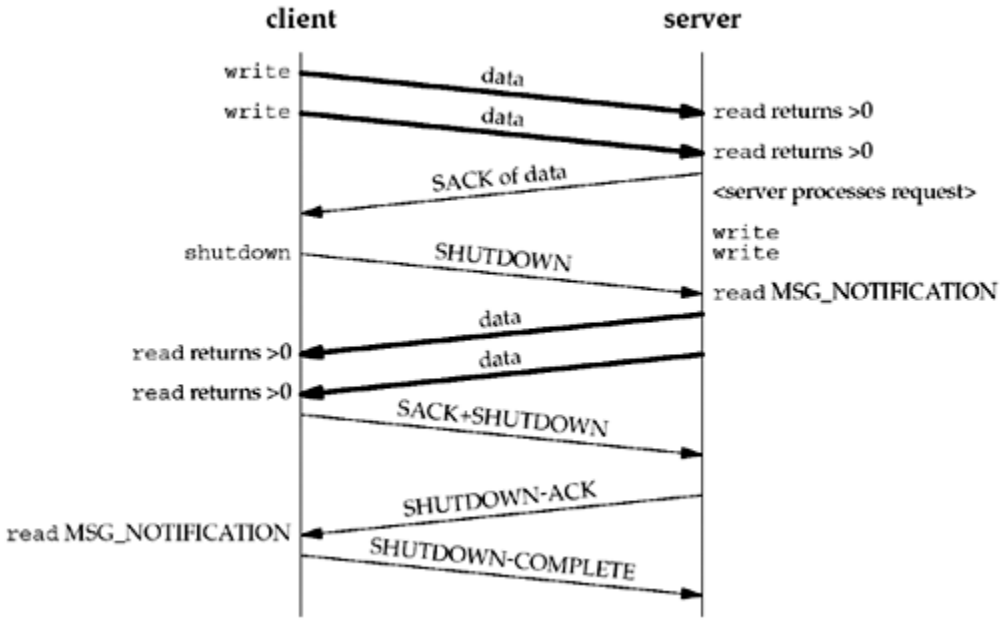
[Team LiB]

PREVIOUS NEXT

9.13 shutdown Function

The shutdown function that we discussed in Section 6.6 can be used with an SCTP endpoint using the one-to-one-style interface. Because SCTP's design does not provide a half-closed state, an SCTP endpoint reacts to a shutdown call differently than a TCP endpoint. When an SCTP endpoint initiates a shutdown sequence, both endpoints must complete transmission of any data currently in the queue and close the association. The endpoint that initiated the active open may wish to invoke shutdown instead of close so that the endpoint can be used to connect to a new peer. Unlike TCP, a close followed by the opening of a new socket is not required. SCTP allows the endpoint to issue a shutdown, and after the shutdown completes, the endpoint can reuse the socket to connect to a new peer. Note that the new connection will fail if the endpoint does not wait until the SCTP shutdown sequence completes. Figure 9.5 shows the typical function calls in this scenario.

Figure 9.5. Calling shutdown to close an SCTP association.



Note that in Figure 9.5, we depict the user receiving the MSG_NOTIFICATION events. If the user had not subscribed to receive these events, then a read of length 0 would have been returned. The effects of the shutdown function for TCP were described in Section 6.6. The shutdown function *howto* holds the following semantics for SCTP:

SHUT_RD	The same semantics as for TCP discussed in Section 6.6; no SCTP protocol action is taken.
SHUT_WR	Disables further send operations and initiates the SCTP shutdown procedures, which will terminate the association. Note that this option does not provide a half-closed state, but does allow the local endpoint to read any queued data that the peer may have sent prior to receiving the SCTP SHUTDOWN message.
SHUT_RDWR	Disables all read and write operations, and initiates the SCTP shutdown procedure. Any queued data that was in transit to the local endpoint will be acknowledged and then silently discarded.

9.14 Notifications

SCTP makes a variety of notifications available to the application programmer. The SCTP user can track the state of its association(s) via these notifications. Notifications communicate transport-level events, including network status change, association startups, remote operational errors, and undeliverable messages. For both the one-to-one and the one-to-many styles, all events are disabled by default with the exception of `sctp_data_io_event`. We will see an example of using notifications in Section 23.7.

Eight events can be subscribed to using the `SCTP_EVENTS` socket option. Seven of these events generate additional data—termed a notification—that a user will receive via the normal socket descriptor. The notifications are added to the socket descriptor inline with data as the events that generate them occur. When reading from a socket with notification subscriptions, user data and notifications will be interleaved on the socket buffer. To differentiate between peer data and a notification, the user uses either the `recvmsg` function or the `sctp_recvmsg` function. When the data returned is an event notification, the `msg_flags` field of these two functions will contain the `MSG_NOTIFICATION` flag. This flag tells the application that the message just read is not data from the peer, but a notification from the local SCTP stack.

Each type of notification is in tag-length-value form, where the first eight bytes of the message identify what type of notification has arrived and its total length. Enabling the `sctp_data_io_event` event causes the receipt of `sctp_sndrcvinfo` structures on every read of user data (this option is enabled by default for both interface styles). This information is normally received in ancillary data using the `recvmsg` call. An application can also use the `sctp_recvmsg` call, which will fill a pointer to the `sctp_sndrcvinfo` structure with this information.

Two notifications contain an SCTP error cause code field. The values for this field are listed in Section 3.3.10 of RFC 2960 [Stewart et al. 2000] and in the "CAUSE CODES" section of http://www.iana.org/assignments/sctp-parameters.

Notifications have the following form:

```
struct sctp_tlv {
  u_int16_t sn_type;
  u_int16_t sn_flags;
  u_int32_t sn_length;
};

/* notification event */
union sctp_notification {
  struct sctp_tlv sn_header;
  struct sctp_assoc_change sn_assoc_change;
  struct sctp_paddr_change sn_paddr_change;
  struct sctp_remote_error sn_remote_error;
  struct sctp_send_failed sn_send_failed;
  struct sctp_shutdown_event sn_shutdown_event;
  struct sctp_adaption_event sn_adaption_event;
  struct sctp_pdapi_event sn_pdapi_event;
};
```

Note that the `sn_header` field is used to interpret the type value, to decode the actual message being sent. Figure 9.6 illustrates the value found in the `sn_header. sn_type` field and the corresponding subscription field used with the `SCTP_EVENTS` socket option.

Figure 9.6. *sn_type* and event subscription field.

sn_type	Subscription field
SCTP_ASSOC_CHANGE	sctp_association_event
SCTP_PEER_ADDR_CHANGE	sctp_address_event
SCTP_REMOTE_ERROR	sctp_peer_error_event
SCTP_SEND_FAILED	sctp_send_failure_event
SCTP_SHUTDOWN_EVENT	sctp_shutdown_event
SCTP_ADAPTION_INDICATION	sctp_adaption_layer_event
SCTP_PARTIAL_DELIVERY_EVENT	sctp_partial_delivery_event

Each notification has its own structure that gives further information about the event that has occurred on the transport.

SCTP_ASSOC_CHANGE

This notification informs an application that a change has occurred to an association; either a new association has begun or an existing association has ended. The information provided with this event is defined as follows:

```
struct sctp_assoc_change {
```

```
  u_int16_t sac_type;
  u_int16_t sac_flags;
  u_int32_t sac_length;
  u_int16_t sac_state;
  u_int16_t sac_error;
  u_int16_t sac_outbound_streams;
  u_int16_t sac_inbound_streams;
  sctp_assoc_t sac_assoc_id;
  uint8_t sac_info[];
};
```

The *sac_state* describes the type of event that has occurred on the association, and will take one of the following values:

SCTP_COMM_UP	This state indicates that a new association has just been started. The inbound and outbound streams fields indicate how many streams are available in each direction. The association identification is filled with a unique value that can be used to communicate with the local SCTP stack regarding this association.
SCTP_COMM_LOST	This state indicates that the association specified by the association identification has closed due to either an unreachability threshold being triggered (i.e., the SCTP endpoint timed out multiple times and hit its threshold, which indicates the peer is no longer reachable), or the peer performed an abortive close (usually with the SO_LINGER option or by using sendmsg with a MSG_ABORT flag) of the association. Any user-specific information will be found in the sac_info field of the notification.
SCTP_RESTART	This state indicates that the peer has restarted. The most likely cause of this notification is a peer crash and restart. The application should verify the number of streams in each direction, since these values may change during a restart.
SCTP_SHUTDOWN_COMP	This state indicates that a shutdown initiated by the local endpoint (via either a shutdown call or a sendmsg with a MSG_EOF flag) has completed. For the one-to-one style, after receiving this notification, the socket descriptor can be used again to connect to a different peer.
SCTP_CANT_STR_ASSOC	This state indicates that a peer did not respond to an association setup attempt (i.e., the INIT message).

The *sac_error* field holds any SCTP protocol error cause code that may have caused an association change. The *sac_outbound_streams* and *sac_inbound_streams* fields inform the application how many streams in each direction have been negotiated on the association. *sac_assoc_id* holds a unique handle for an association that can be used to identify the association in both socket options and future notifications. *sac_info* holds any other information available to the user. For example, if an association was aborted by the peer with a user-defined error, that error would be found in this field.

SCTP_PEER_ADDR_CHANGE

This notification indicates that one of the peer's addresses has experienced a change of state. This change may either be a failure, such as the destination is not responding when sent to, or a recovery, such as a destination that was in a failed state has recovered. The structure that accompanies an address change is as follows:

```
struct sctp_paddr_change {
  u_int16_t spc_type;
  u_int16_t spc_flags;
  u_int32_t spc_length;
  struct sockaddr_storage spc_aaddr;
  u_int32_t spc_state;
  u_int32_t spc_error;
  sctp_assoc_t spc_assoc_id;
};
```

The *spc_aaddr* field holds the address of the peer affected by this event. The spc_state field holds one of the values described in Figure 9.7.

Figure 9.7. SCTP peer address state notifications.

spc_state	Description
SCTP_ADDR_ADDED	Address is now added to the association
SCTP_ADDR_AVAILABLE	Address is now reachable
SCTP_ADDR_CONFIRMED	Address has now been confirmed and is valid
SCTP_ADDR_MADE_PRIM	Address has now been made the primary destination
SCTP_ADDR_REMOVED	Address is no longer part of the association
SCTP_ADDR_UNREACHABLE	Address can no longer be reached

When an address is declared SCTP_ADDR_UNREACHABLE, any data sent to that address will be rerouted to an alternate address. Note also that some of the states will only be available on SCTP implementations that support the dynamic address option (e.g., SCTP_ADDR_ADDED and SCTP_ADDR_REMOVED).

The *spc_error* field contains any notification error code to provide more information about the event, and *spc_assoc_id* holds the association identification.

SCTP_REMOTE_ERROR

A remote peer may send an operational error message to the local endpoint. These messages can indicate a variety of error conditions for the association. The entire error chunk will be passed to the application in wire format when this notification is enabled. The format of the message will be as follows:

```
struct sctp_remote_error {
  u_int16_t sre_type;
  u_int16_t sre_flags;
  u_int32_t sre_length;
  u_int16_t sre_error;
  sctp_assoc_t sre_assoc_id;
  u_int8_t sre_data[];
};
```

The *sre_error* will hold one of the SCTP protocol error cause codes, *sre_assoc_id* will contain the association identification, and *sre_data* will hold the complete error in wire format.

SCTP_SEND_FAILED

When a message cannot be delivered to a peer, the message is sent back to the user through this notification. This notification is usually soon followed by an association failure notification. In most cases, the only way a message will not be delivered is if the association has failed. The only time a message failure will occur without an association failure is when the partial reliability extension of SCTP is being used.

When an error notification is sent, the following format will be read by the application:

```
struct sctp_send_failed {
 u_int16_t ssf_type;
 u_int16_t ssf_flags;
 u_int32_t ssf_length;
 u_int32_t ssf_error;
 struct sctp_sndrcvinfo ssf_info;
 sctp_assoc_t ssf_assoc_id;
 u_int8_t ssf_data[];
};
```

ssf_flags will be set to one of two values:

- SCTP_DATA_UNSENT, which indicates that the message could never be transmitted to the peer (e.g., flow control prevented the message from being sent before its lifetime expired), so the peer never received it
- SCTP_DATA_SENT, which indicates that the data was transmitted to the peer at least once, but was never acknowledged. In this case, the peer *may* have received the message, but it was unable to acknowledge it.

This distinction may be important to a transaction protocol, which might perform different actions to recover from a broken connection based on whether or not a given message might have been received. *ssf_error*, if not zero, holds an error code specific to this notification. The *ssf_info* field provides the information passed (if any) to the kernel when the data was sent (e.g., stream number, context, etc.). *ssf_assoc_id* holds the association identification, and *ssf_data* holds the undelivered message.

SCTP_SHUTDOWN_EVENT

This notification is passed to an application when a peer sends a SHUTDOWN chunk to the local endpoint. This notification informs the application that no new data will be accepted on the socket. All currently queued data will be transmitted, and at the completion of that transmission, the association will be shut down. The notification format is as follows:

```
struct sctp_shutdown_event {
  uint16_t sse_type;
  uint16_t sse_flags;
  uint32_t sse_length;
  sctp_assoc_t sse_assoc_id;
};
```

sse_assoc_id holds the association identification for the association that is shutting down and can no longer accept data.

SCTP_ADAPTION_INDICATION

Some implementations support an adaption layer indication parameter. This parameter is exchanged in the INIT and INIT-ACK to inform each peer what type of application adaption is being performed. The notification will have the following form:

```
struct sctp_adaption_event {
  u_int16_t sai_type;
  u_int16_t sai_flags;
  u_int32_t sai_length;
  u_int32_t sai_adaption_ind;
  sctp_assoc_t sai_assoc_id;
};
```

The *sai_assoc_id* identifies of association that this adaption layer notification. *sai_adaption_ind* is the 32-bit integer that the peer communicates to the local host in the INIT or INIT-ACK message. The outgoing adaption layer is set with the `SCTP_ADAPTION_LAYER` socket option (Section 7.10). The adaption layer INIT/INIT-ACK option is described in [Stewart et al. 2003b], and a sample usage of the option for remote direct memory access/direct data placement is described in [Stewart et al. 2003a].

SCTP_PARTIAL_DELIVERY_EVENT

The partial delivery application interface is used to transport large messages to the user via the socket buffer. Consider a user writing a single message of 4MB. A message of this size would tax or exhaust system resources. An SCTP implementation would fail to handle such a message unless the implementation had a mechanism to begin delivering the message before all of it arrived. When an implementation does this form of delivery, it is termed "*the partial delivery API*." The partial delivery API is invoked by the SCTP implementation sending data with the `msg_flags` field remaining clear until the last piece of the message is ready to be delivered. The last piece of the message will have the `msg_flags` set to `MSG_EOR`. Note that if an application is going to receive large messages, it should use either `recvmsg` or `sctp_recvmsg` so that the `msg_flags` field can be examined for this condition.

In some instances, the partial delivery API will need to communicate a status to the application. For example, if the partial delivery API needs to be aborted, the `SCTP_PARTIAL_DELIVERY_EVENT` notification must be sent to the receiving application. This notification has the following format:

```
struct sctp_pdapi_event {
  uint16_t pdapi_type;
  uint16_t pdapi_flags;
  uint32_t pdapi_length;
  uint32_t pdapi_indication;
  sctp_assoc_t pdapi_assoc_id;
};
```

The *pdapi_assoc_id* field identifies the association upon which the partial delivery API event has occurred. The *pdapi_indication* holds the event that has occurred. Currently, the only valid value found in this field is `SCTP_PARTIAL_DELIVERY_ABORTED`, which indicates that the currently active partial delivery has been aborted.

9.15 Summary

SCTP provides the application writer with two different interface styles: the one-to-one style, mostly compatible with existing TCP applications to ease migration to SCTP, and the one-to-many style, allowing access to all of SCTP's features. The `sctp_peeloff` function provides a method of extracting an association from one style to the other. SCTP also provides numerous notifications of transport events to which an application may wish to subscribe. These events can aid an application in better managing the associations it maintains.

Since SCTP is multihomed, not all the standard sockets functions introduced in Chapter 4 are adequate. Functions like `sctp_bindx`, `sctp_connectx`, `sctp_getladdrs`, and `sctp_getpaddrs` provide methods to better control and examine the multiple addresses that can make up an SCTP association. Utility functions such as `sctp_sendmsg` and `sctp_recvmsg` can simplify the use of these advanced features. We will explore many of the concepts introduced in this chapter in more detail through examples in Chapters 10 and 23.