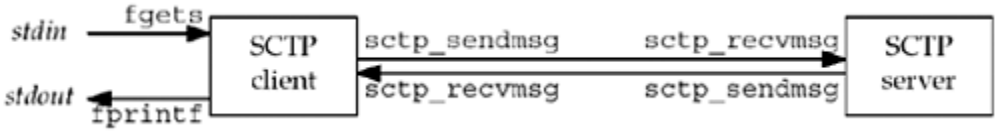
10.1 Introduction

We will now use some of the elementary functions from Chapters 4 and Chapter 9 to write a complete one-to-many SCTP client/server example. Our simple example is similar to the echo server presented in Chapter 5, and performs the following steps:

1. A client reads a line of text from standard input and sends the line to the server. The line follows the form `[#] text`, where the number in brackets is the SCTP stream number on which the text message should be sent.
2. The server receives the text message from the network, increases the stream number on which the message arrived by one, and sends the text message back to the client on this new stream number.
3. The client reads the echoed line and prints it on its standard output, displaying the stream number, stream sequence number, and text string.

Figure 10.1 depicts this simple client/server along with the functions used for input and output.

Figure 10.1. Simple SCTP streaming echo client and server.



We show two arrows between the client and server depicting two unidirectional streams being used, even though the overall association is full-duplex. The `fgets` and `fputs` functions are from the standard I/O library. We do not use the `writen` and `readline` functions defined in Section 3.9 since they are unnecessary. Instead, we use the `sctp_sendmsg` and `sctp_recvmsg` functions defined in Sections 9.9 and Sections 9.10, respectively.

For this example, we use a one-to-many-style server. We make this choice for one important reason. The examples in Chapter 5 can be modified to run over SCTP with one minor change: modify the `socket` function call to specify `IPPROTO_SCTP` instead of `IPPROTO_TCP` as the third argument. Simply making this change, however, would not take advantage of any of the additional features provided by SCTP except multihoming. Using the one-to-many style allows us to exercise all of SCTP's features.

10.2 SCTP One-to-Many-Style Streaming Echo Server: `main` Function

Our SCTP client and server follow the flow of functions diagrammed in Figure 9.2. We show an iterative server program in Figure 10.2.

Set stream increment option

13–14 By default, our server responds using the next higher stream than the one on which the message was received. If an integer argument is passed on the command line, the server interprets the argument as the value of `stream_increment`, that is, it decides whether or not to increment the stream number of incoming messages. We will use this option in our discussion of head-of-line blocking in Section 10.5.

Create an SCTP socket

15 An SCTP one-to-many-style socket is created.

Bind an address

16–20 An Internet socket address structure is filled in with the wildcard address (`INADDR_ANY`) and the server's well-known port, `SERV_PORT`. Binding the wildcard address tells the system that this SCTP endpoint will use all available local addresses in any association that is set up. For multihomed hosts, this binding means that a remote endpoint will be able to make associations with and send packets to any of the local host's routeable addresses. Our choice of the SCTP port number is based on Figure 2.10. Note that the server makes the same considerations that were made earlier in our previous example found in Section 5.2.

Set up for notifications of interest

21–23 The server changes its notification subscription for the one-to-many SCTP socket. The server subscribes to just the `sctp_data_io_event`, which will allow the server to see the `sctp_sndrcvinfo` structure. From this structure, the server can determine the stream number on which the message arrived.

Enable incoming associations

24 The server enables incoming associations with the `listen` call. Then, control enters the main processing loop.

Figure 10.2 SCTP streaming echo server.

sctp/sctpserv01.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sock_fd, msg_flags;
 6     char    readbuf [BUFFSIZE];
 7     struct sockaddr_in servaddr, cliaddr;
 8     struct sctp_sndrcvinfo sri;
 9     struct sctp_event_subscribe evnts;
10     int     stream_increment = 1;
11     socklen_t len;
12     size_t rd_sz;

13     if (argc == 2)
14         stream_increment = atoi (argv[1]);
15     sock_fd = Socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16     bzero (&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
19     servaddr.sin_port = htons (SERV_PORT);

20     Bind (sock_fd, (SA *) &servaddr, sizeof (servaddr));

21     bzero (&evnts, sizeof (evnts)) ;
```

```
22      evnts.sctp_data_io_event = 1;
23      Setsockopt (sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof (evnts)) ;

24      Listen(sock_fd, LISTENQ) ;
25      for ( ; ; ) {
26          len = sizeof(struct sockaddr_in) ;
27          rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof (readbuf) ,
28                               (SA *) &cliaddr, &len, &sri, &msg_flags) ;
29          if (stream_increment) {
30              sri.sinfo_stream++;
31              if (sri.sinfo_stream >=
32                 sctp_get_no_strms (sock_fd, (SA *) &cliaddr, len) )
33                 sri.sinfo_stream = 0;
34          }
35          Sctp_sendmsg (sock_fd, readbuf, rd_sz,
36                        (SA *) &cliaddr, len,
37                        sri.sinfo_ppid,
38                        sri.sinfo_flags, sri.sinfo_stream, 0, 0) ;
39      }
40  }
```

Wait for message

26–28 The server initializes the size of the client socket address structure, then blocks while waiting for a message from any remote peer.

Increment stream number if desired

29–34 When a message arrives, the server checks the `stream_increment` flag to see if it should increment the stream number. If the flag is set (no arguments were passed on the command line), the server increments the stream number of the message. If that number grows larger than or equal to the maximum streams, which is obtained by calling our internal function call `sctp_get_no_strms`, the server resets the stream to 0. The function `sctp_get_no_strms` is not shown. It uses the `SCTP_STATUS` SCTP socket option discussed in Section 7.10 to find the number of streams negotiated.

Send back response

35–38 The server sends back the message using the payload protocol ID, flags, and the possibly modified stream number from the `sri` structure.

Notice that this server does not want association notification, so it disables all events that would pass messages up the socket buffer. The server relies on the information in the `sctp_sndrcvinfo` structure and the returned address found in *cliaddr* to locate the peer association and return the echo.

This program runs forever until the user shuts it down with an external signal.

10.3 SCTP One-to-Many-Style Streaming Echo Client: `main` Function

Figure 10.3 shows our SCTP client main function.

Validate arguments and create a socket

9–15 The client validates the arguments passed to it. First, the client verifies that the caller provided a host to send messages to. It then checks if the "echo to all" option is being enabled (we will see this used in Section 10.5). Finally, the client creates an SCTP one-to-many-style socket.

Set up server address

16–20 The client translates the server address, passed on the command line, using the `inet_pton` function. It combines that with the server's well-known port number and uses the resulting address as the destination for the requests.

Set up for notifications of interest

21–23 The client explicitly sets the notification subscription provided by our one-to-many SCTP socket. Again, it wants no `MSG_NOTIFICATION` events. Therefore, the client turns these off (as was done in the server) and only enables the receipt of the `sctp_sndrcvinfo` structure.

Call echo processing function

24–28 If the `echo_to_all` flag is not set, the client calls the `sctpstr_cli` function, discussed in Section 10.4. If the `echo_to_all` flag is set, the client calls the `sctpstr_cli_echoall` function. We will discuss this function in Section 10.5 as we explore uses for SCTP streams.

Figure 10.3 SCTP streaming echo client main.

sctp/sctpclient01.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sock_fd;
 6     struct sockaddr_in servaddr;
 7     struct sctp_event_subscribe evnts;
 8     int     echo_to_all = 0;
 9     if (argc < 2)
10         err_quit("Missing host argument - use '%s host [echo] '\n", argv[0]) ;
11     if (argc > 2) {
12         printf("Echoing messages to all streams\n") ;
13         echo_to_all = 1;
14     }
15     sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16     bzero(&servaddr, sizeof (servaddr) ) ;
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
19     servaddr.sin_port = htons (SERV_PORT);
20     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

21     bzero(&evnts, sizeof (evnts)) ;
22     evnts.sctp_data_io_event = 1 ;
23     Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof (evnts)) ;
24     if (echo_to_all == 0)
25         sctpstr_cli (stdin, sock_fd, (SA *) &servaddr, sizeof (servaddr)) ;
26     else
27         sctpstr_cli_echoall(stdin, sock_fd, (SA *) &servaddr,
28                             sizeof (servaddr)) ;
29     Close (sock_fd) ;
30     return (0) ;
31 }
```

Finish up

29–31 On return from processing, the client closes the SCTP socket, which shuts down any SCTP associations using the socket. The client then returns from main with a return code of 0, indicating that the program ran successfully.