

Better Networking with SCTP

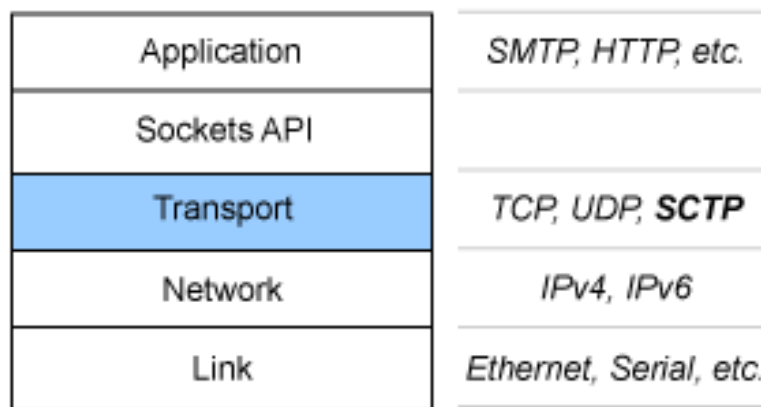
M. Tim Jones

Summary

The Stream Control Transmission Protocol combines advantages from both TCP and UDP. The Stream Control Transmission Protocol (SCTP) is a reliable transport protocol that provides stable, ordered delivery of data between two endpoints (much like TCP) and also preserves data message boundaries (like UDP). However, unlike TCP and UDP, SCTP offers such advantages as multi-homing and multi-streaming capabilities, both of which increase availability. In this article, get to know the key features of SCTP in the Linux® 2.6 kernel and take a look at the server and client source code that shows the protocol's ability to deliver multi-streaming.

SCTP is a reliable, general-purpose transport layer protocol for use on IP networks. While the protocol was originally designed for telephony signaling (under the RFC 2960), SCTP provided an added bonus -- it solved some of the limitations of TCP while borrowing beneficial features of UDP. SCTP provides features for high availability, increased reliability, and improved security for socket initiation. (Figure 1 shows the layered architecture of the IP stack.)

Figure 1: Layered Architecture of the IP Stack



This article introduces the concept of SCTP in the Linux 2.6 kernel, highlights some of the advanced features (such as multi-homing and -streaming), and provides server and client source code snippets (with a URL to more code) to demonstrate the protocol's ability to deliver multi-streaming.

Let's start with an overview of the IP stack.

The IP Stack

The Internet protocol suite is split into several layers; each layer provides specific functionality as shown in Figure 1.

Starting from the bottom:

- The link layer provides the physical interface to the communication medium (such as an Ethernet device).
- The network layer manages the movement of packets in a network, specifically making sure packets get to their destination (also called routing).
- The transport layer regulates the flow of packets between two hosts for the application layer. It also presents the application endpoint for communication, known as a port.

Better Networking with SCTP

M. Tim Jones

- Finally, the application layer provides meaning to the data transported through the socket. This data could consist of e-mail messages using the Simple Mail Transport Protocol (SMTP) or Web pages rendered through the Hypertext Transport Protocol (HTTP).

All application layer protocols use the sockets layer as their interface to the transport layer protocol. The Sockets API was developed at UC Berkeley within the BSD UNIX® operating system.

Now for a quick refresher on traditional transport layer protocols before we dive into the workings of SCTP.

The Transport Layer Protocols

The two most popular transport layer protocols are the transmission control protocol (TCP) and the user datagram protocol (UDP):

- TCP is a reliable protocol that guarantees sequenced, ordered delivery of data and manages congestion within a network.
- UDP is a message-oriented protocol that neither guarantees ordering of delivery nor manages congestion.

However, UDP is a fast protocol that preserves the boundaries of the messages it transports.

This article presents another option: SCTP. It provides the reliable, ordered delivery of data like TCP but operates in the message-oriented fashion like UDP, preserving message boundaries. SCTP also provides several advanced features:

- Multi-homing
- Multi-streaming
- Initiation protection
- Message framing
- Configurable unordered delivery
- Graceful shutdown

Key Features of SCTP

The two most important enhancements in SCTP over traditional transport layer protocols are the end-host multi-homing and multi-streaming capabilities.

Multi-homing

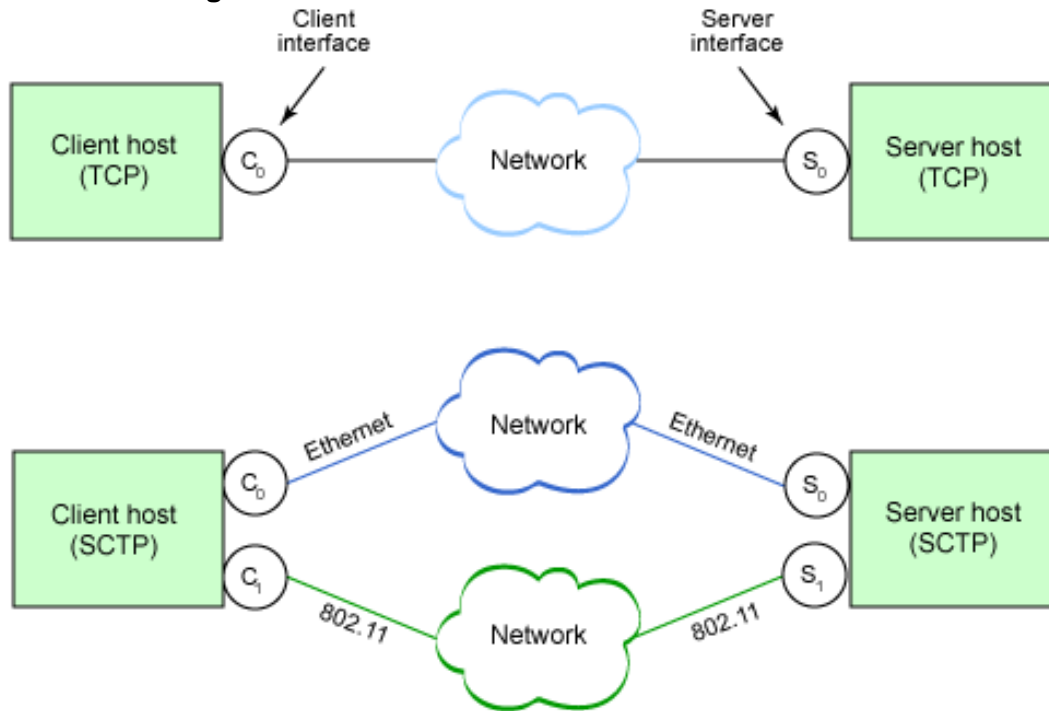
Multi-homing provides applications with higher availability than those that use TCP. A multi-homed host is one that has more than one network interface and therefore more than one IP address for which it can be addressed. In TCP, a connection refers to a channel between two endpoints (in this case, a socket between the interfaces of two hosts). SCTP introduces the concept of an association that exists between two hosts but can potentially collaborate with multiple interfaces at each host.

Figure 2 illustrates the difference between a TCP connection and an SCTP association.

Better Networking with SCTP

M. Tim Jones

Figure 2: TCP Connection vs. an SCTP Association



At the top is a TCP connection. Each host includes a single network interface; a connection is created between a single interface on each of the client and server. Upon establishment, the connection is bound to each interface.

At the bottom of the figure, you can see an architecture that includes two network interfaces per host. Two paths are provided through the independent networks, one from interface C₀ to S₀ and another from C₁ to S₁. In SCTP, these two paths would be collected into an association.

SCTP monitors the paths of the association using a built-in heartbeat; upon detecting a path failure, the protocol sends traffic over the alternate path. It's not even necessary for the applications to know that a failover recovery occurred.

Failover can also be used to maintain network application connectivity. For example, consider a laptop that includes a wireless 802.11 interface and an Ethernet interface. When the laptop is in its docking station, the higher-speed Ethernet interface would be preferred (in SCTP, called the primary address); but upon loss of this connection (removal from the docking station), connections would be failed over to the wireless interface. Upon return to the docking station, the Ethernet connection would be detected and communication resumed over this interface. This is a powerful mechanism for providing high availability and increased reliability.

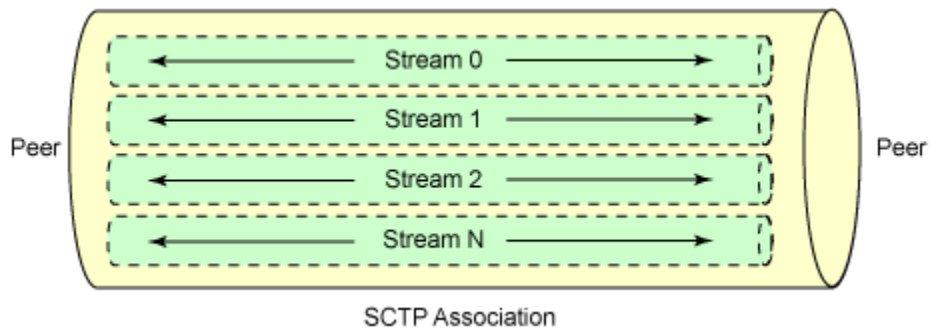
Multi-streaming

In some ways, an SCTP association is like a TCP connection except that SCTP supports multiple streams within an association. All the streams within an association are independent but related to the association (see Figure 3).

Better Networking with SCTP

M. Tim Jones

Figure 3. Relationship of an SCTP Association to Streams



Each stream is given a stream number that is encoded inside SCTP packets flowing through the association. Multi-streaming is important because a blocked stream (for example, one awaiting retransmission resulting from the loss of a packet) does not affect the other streams in an association. This problem is commonly referred to as head-of-line blocking. TCP is prone to such blocking.

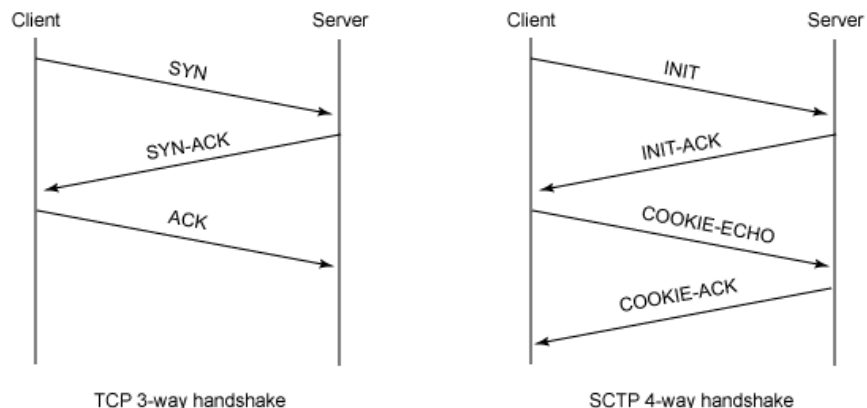
How can multiple streams provide better responsiveness in transporting data? For example, the HTTP protocol shares control and data over the same socket. A Web client requests a file from a server, and the server sends the file back over the same connection. A multi-streamed HTTP server would provide better interactivity because multiple requests could be serviced on independent streams within the association. This functionality would parallelize the responses, and while not potentially faster, would simultaneously load the HTML and graphics images, providing the perception of better responsiveness.

Multi-streaming is an important feature of SCTP, especially when you consider some of the control and data issues in protocol design. In TCP, control and data typically share the same connection, which can be problematic because control packets can be delayed behind data packets. If control and data were split into independent streams, control data could be dealt with in a more timely manner, resulting in better utilization of available resources.

Initiation Protection

Initiating a new connection in TCP and SCTP occurs with a packet handshake. In TCP, it's called a three-way handshake. The client sends a SYN packet (short for Synchronize) for which the server responds with a SYN-ACK packet (Synchronize-Acknowledge). Finally, the client confirms receipt with an ACK packet (see Figure 4).

Figure 4. The Packet Exchanges for the TCP and STCP Handshake



Better Networking with SCTP

M. Tim Jones

The problem that can occur with TCP is when a rogue client forges an IP packet with a bogus source address, then floods a server with TCP SYN packets. The server allocates resources for the connections upon receipt of the SYN, then under a flood of SYN packets, eventually runs out and is unable to service new requests. This is called a Denial of Service (DoS) attack.

SCTP protects against this type of attack through a four-way handshake and the introduction of a cookie. In SCTP, a client initiates a connection with an INIT packet. The server responds with an INIT-ACK, which includes the cookie (a unique context identifying this proposed connection). The client then responds with a COOKIE-ECHO, which contains the cookie sent by the server. At this point, the server allocates the resource for the connection and acknowledges this by sending a COOKIE-ACK to the client.

To solve the problem of delayed data movement with the four-way handshake, SCTP permits data to be included in the COOKIE-ECHO and COOKIE-ACK packets.

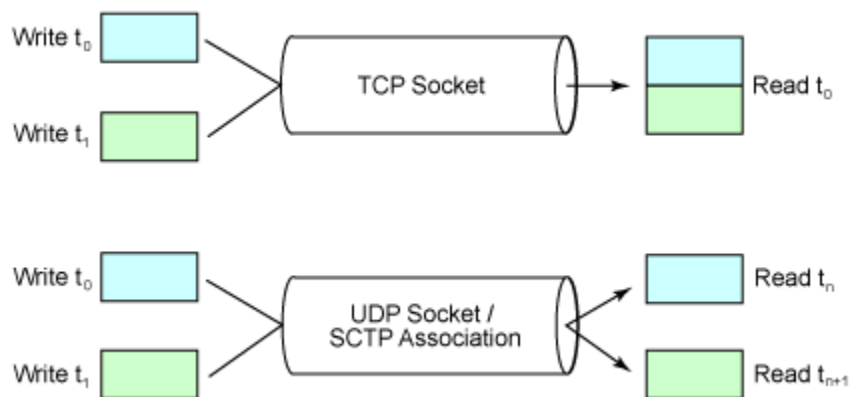
Message Framing

With message framing, the boundaries in which messages are communicated through a socket are preserved; this means that if a client sends 100 bytes to a server followed by 50 bytes, the server will read 100 bytes and 50 bytes, respectively, for two reads. UDP also operates in this way, which makes it advantageous for message-oriented protocols.

In contrast, TCP operates in a byte-stream fashion. Without framing, a peer may receive more or less than was sent (splitting up a write or aggregating multiple writes into a single read). This behavior requires that message-oriented protocols operating over TCP provide data-buffer and message framing within their application layer (a potentially complex task).

SCTP provides for message framing in data transfer. When a peer performs a write on a socket, it is guaranteed that this same-sized chunk of data will be read at the peer endpoint (see Figure 5).

Figure 5: Message Framing in UDP/SCTP vs. a Byte-Stream-Oriented Protocol



For stream-oriented data, such as audio or video data, lack of framing is acceptable.

Configurable Unordered Delivery

Messages in SCTP are transferred reliably but not necessarily in the desired order. TCP guarantees that data is delivered in order (which is a good thing, considering TCP is a stream protocol). UDP guarantees no ordering. But, you can also configure streams within SCTP to accept unordered messages if desired.

Better Networking with SCTP

M. Tim Jones

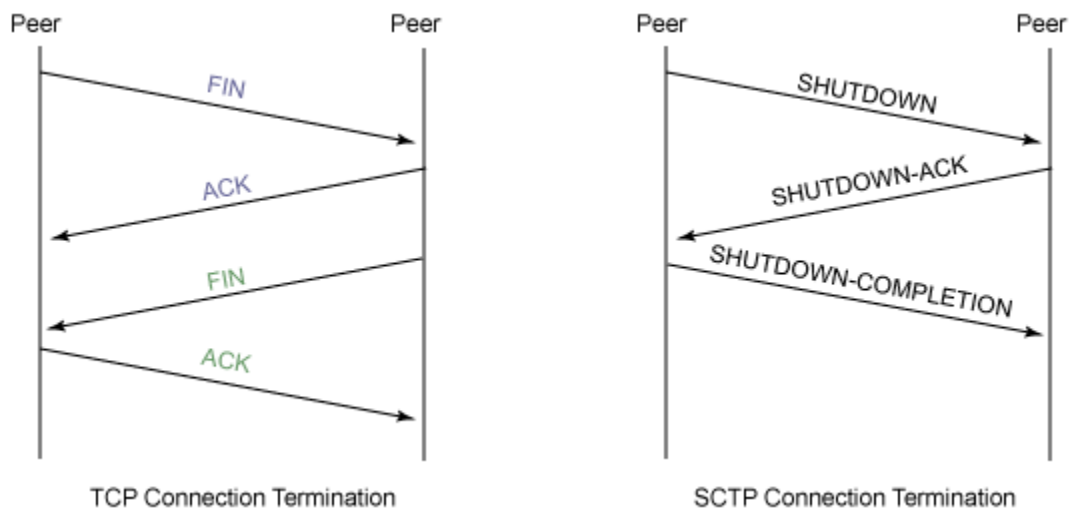
This feature can be useful in message-oriented protocols in which requests are independent and ordering is not important. Further, you can configure unordered delivery on a stream-by-stream basis within an association.

Graceful Shutdown

TCP and SCTP are connection-based protocols, while UDP is a connection-less protocol. Both TCP and SCTP require connection setup and teardown between peers. What's different about socket shutdown in SCTP is the removal of TCP's half-close.

Figure 6 shows the shutdown sequences for TCP and SCTP.

Figure 6: TCP and SCTP Connection Termination Sequences



In TCP, it's possible for a peer to close its end of a socket (resulting in a FIN packet being sent) but then to continue to receive data. The FIN indicates that no more data is to be sent by this endpoint, but until the peer closes its end of the socket, it may continue to transmit data. Applications rarely use this half-closed state, and therefore the SCTP designers opted to remove it and replace it with a cleaner termination sequence. When a peer closes its socket (resulting in the issuance of a SHUTDOWN primitive), both endpoints are required to close, and no further data movement is permitted in either direction.

Multi-streaming Demo

Now that you know the basic features of SCTP, let's look at a sample server and client that are written in the C programming language and demonstrate SCTP's multi-streaming feature.

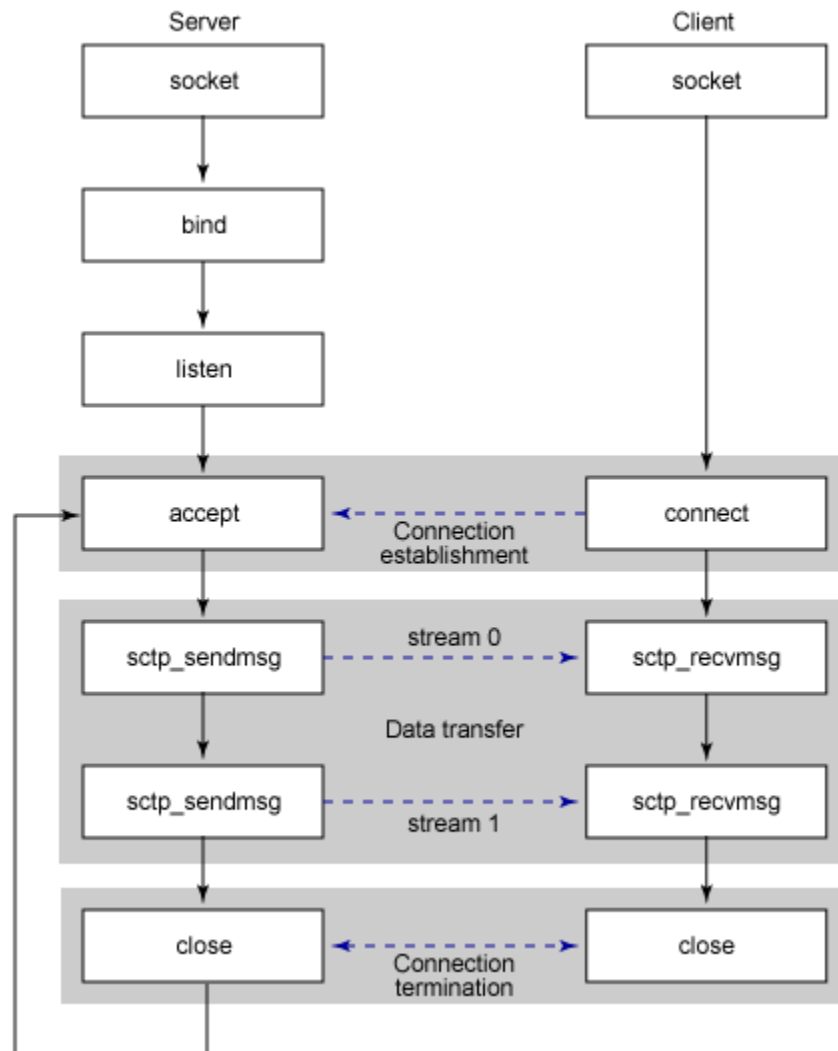
This example presents a server that implements a form of the daytime protocol. This traditional server emits the current time to a connected client, but for SCTP, I emit the local time on stream 0 and Greenwich Mean Time (GMT) on stream 1. This simple example allows me to demonstrate the APIs for stream communication.

Figure 7 outlines the entire process and shows not only the flow of the application from a sockets API perspective but also the relationships from a client and server perspective.

Better Networking with SCTP

M. Tim Jones

Figure 7: Sockets Functions Used in the Multi-Streaming Daytime Server and Client



These applications were developed on the GNU/Linux operating system with a 2.6.11 kernel and the Linux Kernel SCTP project (lksctp). The nonstandard sockets functions are provided in the lksctp tools package, which is available from SourceForge. See Resources for links.

The Daytime Server

The multi-stream daytime server is shown in Listing 1. All error checking is omitted in Listing 1 for better readability, but the code you can download below demonstrates error checking as well as other SCTP socket extensions.

Listing 1: Daytime Server Written for SCTP Using Multiple Streams

```
int main()
{
    int listenSock, connSock, ret;
    struct sockaddr_in servaddr;
    char buffer[MAX_BUFFER+1];
    time_t currentTime;

    /* Create SCTP TCP-Style Socket */
```

Better Networking with SCTP

M. Tim Jones

```
listenSock = socket( AF_INET, SOCK_STREAM, IPPROTO_SCTP );

/* Accept connections from any interface */
bzero( (void *)&servaddr, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
servaddr.sin_port = htons(MY_PORT_NUM);

/* Bind to the wildcard address (all) and MY_PORT_NUM */
ret = bind( listenSock,
           (struct sockaddr *)&servaddr, sizeof(servaddr) );

/* Place the server socket into the listening state */
listen( listenSock, 5 );

/* Server loop... */
while( 1 ) {

    /* Await a new client connection */
    connSock = accept( listenSock,
                     (struct sockaddr *)NULL, (int *)NULL );

    /* New client socket has connected */

    /* Grab the current time */
    currentTime = time(NULL);

    /* Send local time on stream 0 (local time stream) */
    snprintf( buffer, MAX_BUFFER, "%s\n", ctime(&currentTime) );

    ret = sctp_sendmsg( connSock,
                      (void *)buffer, (size_t)strlen(buffer),
                      NULL, 0, 0, 0, LOCALTIME_STREAM, 0, 0 );

    /* Send GMT on stream 1 (GMT stream) */
    snprintf( buffer, MAX_BUFFER, "%s\n",
             asctime( gmtime( &currentTime ) ) );

    ret = sctp_sendmsg( connSock,
                      (void *)buffer, (size_t)strlen(buffer),
                      NULL, 0, 0, 0, GMT_STREAM, 0, 0 );

    /* Close the client connection */
    close( connSock );

}

return 0;
}
```

The server begins in Listing 1 with the creation of the server socket (using `IPPROTO_SCTP` to create an SCTP one-to-one socket). I then create a `sockaddr` structure, specifying that connections are permitted from any local interface (using the wildcard address `INADDR_ANY`). I bind this `sockaddr` structure to the socket using the `bind` call, then place the server socket into the listening state. At this point, incoming connections are possible.

Better Networking with SCTP

M. Tim Jones

Notice that SCTP uses many of the same sockets APIs as TCP and UDP do. Some additional API functions are provided in the `lksctp` development tools (see Resources).

In the server loop, new client connections are awaited. Upon return from the `accept` function, a new client connection is identified by the `connSock` socket. I grab the current time using the `time` function, then convert it to a string with `sprintf`. With the `sctp_sendmsg` function (a nonstandard sockets call), I can send the string to the client, specifying the particular stream (`LOCALTIME_STREAM`). When the `localtime` string has been sent, I package the current time in GMT as a string, then send this on stream `GMT_STREAM`.

At this point, the daytime server has fulfilled its duty, so I close the socket and await a new client connection. Simple, right? Now let's see how the daytime client handles multi-streaming.

The Daytime Client

The multi-streaming client is shown in Listing 2.

Listing 2: Daytime Client Written for SCTP Using Multiple Streams

```
int main()
{
    int connSock, in, i, flags;
    struct sockaddr_in servaddr;
    struct sctp_sndrcvinfo sndrcvinfo;
    struct sctp_event_subscribe events;
    char buffer[MAX_BUFFER+1];

    /* Create an SCTP TCP-Style Socket */
    connSock = socket( AF_INET, SOCK_STREAM, IPPROTO_SCTP );

    /* Specify the peer endpoint to which we'll connect */
    bzero( (void *)&servaddr, sizeof(servaddr) );
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(MY_PORT_NUM);
    servaddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );

    /* Connect to the server */
    connect( connSock, (struct sockaddr *)&servaddr, sizeof(servaddr) );

    /* Enable receipt of SCTP Snd/Rcv Data via sctp_recvmsg */
    memset( (void *)&events, 0, sizeof(events) );
    events.sctp_data_io_event = 1;
    setsockopt( connSock, SOL_SCTP, SCTP_EVENTS,
                (const void *)&events, sizeof(events) );

    /* Expect two messages from the peer */
    for (i = 0 ; i < 2 ; i++) {

        in = sctp_recvmsg( connSock, (void *)buffer, sizeof(buffer),
                          (struct sockaddr *)NULL, 0,
                          &sndrcvinfo, &flags );

        /* Null terminate the incoming string */
        buffer[in] = 0;

        if (sndrcvinfo.sinfo_stream == LOCALTIME_STREAM) {
            printf("(Local) %s\n", buffer);
        }
    }
}
```

Better Networking with SCTP

M. Tim Jones

```
    } else if (sndrcvinfo.sinfo_stream == GMT_STREAM) {
        printf("(GMT  ) %s\n", buffer);
    }

}

/* Close our socket and exit */
close(connSock);

return 0;
}
```

In the client, I create an SCTP socket, then create a sockaddr structure containing the endpoint to which it will connect. The connect function then establishes a connection to the server. To retrieve the stream number of a message, SCTP requires enabling the socket option `sctp_data_io_event`).

With this enabled, when I receive a message through the `sctp_rcvmsg` API function, I also receive a `sctp_sndrcvinfo` structure that contains the stream number. This number allows me to discriminate between messages from stream 0 (localtime) and stream 1 (GMT).

The Future of SCTP

SCTP is a relatively new protocol, considering that it became an RFC in October 2000. Since then, it has found its way into all major operating systems, including GNU/Linux, BSD, and Solaris. It's also available for the Microsoft® Windows® operating systems as a third-party commercial package.

Along with availability, applications will begin to use SCTP as their primary transport. Traditional applications such as FTP and HTTP have been built on the features of SCTP. Other protocols are using SCTP, such as the Session Initiation Protocol (SIP) and the Common Channel Signaling System No. 7 (SS7). Commercially, you can find SCTP in Cisco's IOS.

With the inclusion of SCTP into the 2.6 Linux kernel, it's now possible to build and deploy highly available and reliable networked applications. As an IP-based protocol, SCTP is a seamless replacement for TCP and UDP but also extends new services, such as multi-homing, multi-streaming, and increased security. Now that you've seen some of the high-level features of SCTP, explore its other capabilities. The Linux Kernel SCTP project (`lksctp`) provides API extensions and documentation that will help you on your way.