SCTP: What is it, and how to use it?

Randall Stewart
Cisco Systems Inc.
4875 Forest Drive
Suite 200
Columbia, SC 29206
USA
Email: rrs@cisco.com

Michael Tüxen
Münster University of Applied Sciences
Department of Electrical Engineering
and Computer Science
Stegerwaldstr. 39
D-48565 Steinfurt
Germany
Email: tuexen@fh-muenster.de

Peter Lei
Cisco Systems Inc.
8735 West Higgins Road
Suite 300
Chicago, IL 60631
USA
Email: peterlei@cisco.com

***Abstract*—Stream Control Transmission Protocol (SCTP) is a new transport protocol incorporated into FreeBSD 7.0. It was first standardized by the Internet Engineering Task Force (IETF) in October of 2000 in RFC 2960 and later updated by RFC 4960. SCTP is a message oriented protocol providing reliable end to end communication between two peers in an IP network. So, why would one want to use SCTP instead of just using TCP or UDP?**

This paper will try to answer that question by detailing services provided by SCTP, and illustrating how SCTP can be easily used with the socket API.

I. Introduction

Stream Control Transmission Protocol (SCTP) [4] is a reliable, message oriented transport protocol that provides new services and features for IP communication. For the past twenty years, reliable communication service has been provided by TCP [2], and unreliable service has been provided by UDP [1]. So, what has brought about the addition of a third protocol to the IP suite of protocols? Many of the features found in TCP and UDP can also be found in SCTP. See TABLE I for a comprehensive comparison of the features.

As you can see, SCTP overlaps and adds to the list of features that an application developer can draw upon. The rest of this document will be laid out as follows: In Section II we will compare and contrast the feature sets of SCTP, TCP and UDP. In Section III we will provide an overview of the socket API used with SCTP. In Section IV we will get into some of the details for using the SCTP socket API. In Section V we will close with a brief review and a few conclusions.

II. A comparison of features

SCTP, like TCP, provides a connection oriented, full duplex, reliable data communication path. Many of the standard features you will find in TCP (congestion control, flow control, etc.) can also be found in SCTP. However, unlike TCP, SCTP provides a transport of messages, not just bytes. In this section we will go through the unique and strikingly different features found in SCTP. For features that are the same (e.g. congestion control in TCP versus SCTP) we will not comment and allow the reader to research the feature in TCP (if interested) since SCTP and TCP will react in the same way.

Service/Features	SCTP	TCP	UDP
Message-Oriented	yes	no	yes
Byte-Oriented	no	yes	no
Connection-Oriented	yes	yes	no
Full Duplex	yes	yes	yes
Reliable data transfer	yes	yes	no
Partially-Reliable data transfer	opt	no	no
Ordered data delivery	yes	yes	no
Unordered delivery	yes	no	yes
Flow control	yes	yes	no
Congestion Control	yes	yes	no
ECN Capable	yes	yes	no
Selective Acknowledgments	yes	opt	no
Path MTU discovery	yes	yes	no
Application PDU fragmentation	yes	yes	no
Application PDU bundling	yes	yes	no
Multistreaming	yes	no	no
Multihoming	yes	no	no
Dynamic Multihoming	opt	no	no
SYN flooding attack prevention	yes	no	n/a
Allows half-closed state	no	yes	n/a
Reach-ability check	yes	opt	no
Pseudo-header for checksum	no	yes	yes
Time wait state	no	yes	n/a
Authentication	opt	opt	no
CRC based checksum	yes	no	no

TABLE I
Feature List

A. Ordering options

One of the two most striking features of SCTP is “multi-streaming.” When you hear this term, what is being referred to in reality is the ordering options that SCTP provides. The term is also where the “stream” in SCTP’s name comes from.

In a classic TCP connection, you have no choices in the ordering and presentation of your data to your peer. All bytes transmitted are delivered in strict transmission order. In many instances this is exactly what an application wants. Consider for a moment a set of transactions to a banking system from a client application. If a transfer of money is first sent, followed by a sizable withdrawal from that account, we would not want these two requests reversed in order; absolute order is important.

Now consider that same set of transactions with multiple

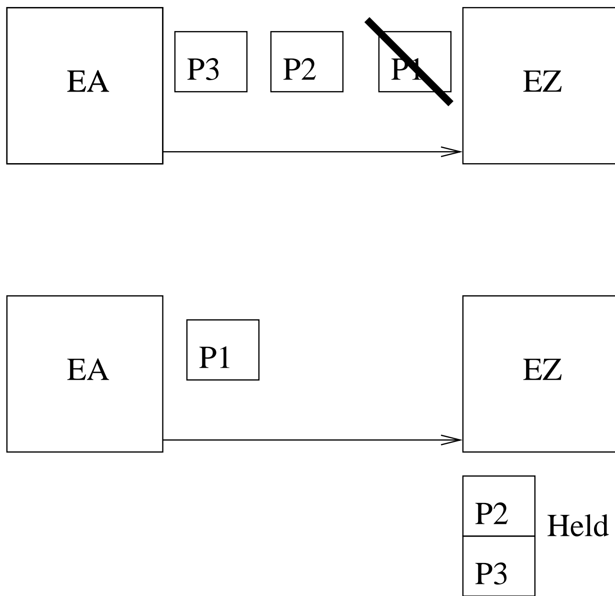

Fig. 1. A lost packet

users involved. A withdrawal or deposit for customer A is unimportant with respect to customer B (assuming the transactions are not between customers A and B). Thus, if the transactions being sent over the connection pertain to multiple users, ordering between the separate transactions is not important. In fact, strict ordering can actually cause an undesirable additional delay in processing in the presence of network loss. When such a delay happens, it is termed Head of Line (HOL) blocking. But how exactly does this occur?

Let us consider a TCP connection as shown in Fig 1. The sender, EA, sends Packets P1, P2 and P3. Note that to TCP the data in each packet is dependent on what is in its send queue at the time TCP decides to send the data. No message boundaries are implied by the packets. In this transfer, we have the unfortunate occurrence of a lost packet: P1 is lost. P2 and P3 arrive safely, but since the data has not *all* arrived in order, they are held by the TCP stack awaiting the retransmission of P1. This will usually happen within one second or so depending on TCP's retransmission algorithms and timers for the connection's network path.

Now what has happened is that any data in P1 has caused an HOL condition for data held within P2 and P3. In some cases, as we have stated, this may be very desirable. However in many instances, the information in P2 or P3 is not related to P1 (as we also described above). This can cause delays in processing of information that may be undesirable or unacceptable to the application.

SCTP's streams were designed to deal with this very problem and provide a method for applications to have finer grain control of what gets blocked by the SCTP stack when such a packet loss occurs. When an application sends data, it can specify a stream number to be used. There are up to 65,535 streams in an SCTP association[1] (the exact number is negotiated at association startup). When a message is sent in Stream N, a loss of data that does not have any Stream N data does not cause delays in delivery. In other words, when data arrives in a given stream (N) it is only held for reordering if other data is missing for that same stream. For example, let us assume that P1 contains a message for stream 11, P2 contains a message for stream 2 and P3 contains a message for stream 6. Only messages in stream 11 would be blocked by the loss of P1 so both the payloads of P2 and P3 would not be held but delivered immediately upon arrival[2]. Streams allow an application to do selective ordering of messages.

SCTP also provides another ordering constraint that mimics UDP (i.e. none). An application is allowed to send a message as "unordered" on a per message basis. When an application sends in this fashion, it is specifying that the data has *no* order with respect to any other data sent previously. An unordered message is placed into the delivery queue (often times termed a socket buffer) immediately upon arrival.

The combination of streams and unordered data provide powerful tools an application can use to provide fine grain control over the delivery of data upon arrival at the peer endpoint SCTP stack. Many applications in today's Internet can gain improved performance in the face of network loss by using these features.

B. *Multihoming*

The second of the two key features of SCTP is multihoming. A host that is multihomed has more than one point of attachment to the network. In a traditional TCP connection, one IP address and one port are chosen at each end to send and receive packets with. The two IP addresses and ports selected represent the four tuple identification of the TCP connection (IP-A + Port-A, IP-Z + Port-Z).

In SCTP, an association is composed of two "sets" of IP addresses and two ports. This means that if two hosts are multihomed, all of their IP addresses can be involved in sending and receiving data. In instances of lost packets, SCTP will often select one of the alternate addresses to send data to. This provides a form of network resilience in the face of network loss and outages. Consider Fig 2(a). The network connection to EZ via IP3 has failed, and EA sends two packets P1 and P2 using IP3 as the destination. Naturally the data will be lost, since there is an "airgap" between the network and the host. Later, when EA's SCTP stack detects the loss (via timeout) the retransmission would take the alternate path, as shown in Fig 2(b). This gives the application additional redundancy. When you combine this with the ability to fine tune the retransmission timers (including the minimum and maximum values) an application can improve both its availability and how quickly it will recover from network errors.

[1]The use of the term association is common for SCTP and represents a similar concept to a TCP connection.

[2]This assumes, of course, that no other losses have occurred.

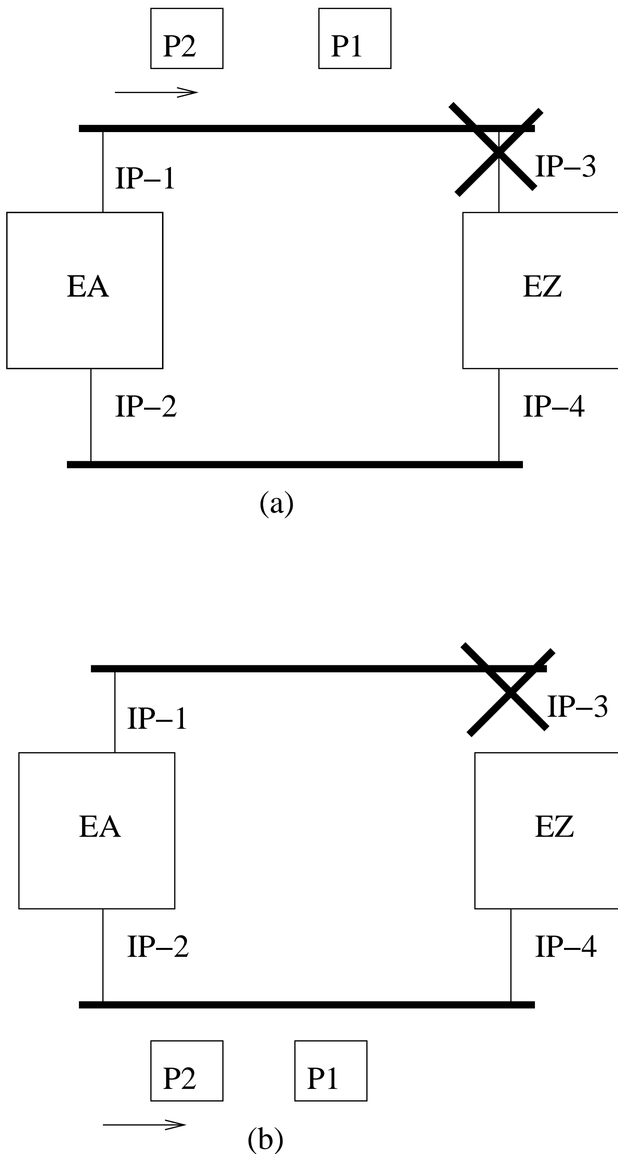

Fig. 2. Packet loss in a multi-homed scenario

An additional feature that many SCTP stacks offer is the ability to dynamically reconfigure the IP address set that makes up the association [6]. This provides the ability for an association to survive an address renumbering or take advantage of a hot-pluggable network interface without restarting the association. Again, this unique feature provides yet again more support to an application attempting to minimize down time.

Lastly, it is important to note that SCTP's address sets allow any mix of IPv4 and IPv6 addresses at each endpoint. That is, an SCTP association may use both IPv4 and IPv6 addresses, providing additional network path diversity.

C. Partial Reliability

Partial Reliability, as the name implies, allows SCTP to control the amount of reliability an application wants on a per message granularity. Consider a video game sending character position updates every 400 milliseconds. If data is lost and not retransmitted within that time period (400 ms), then retransmitting the character position makes no sense since a new updated one would already be en-queued. With the addition of [3], SCTP gained the ability to specify how long a message is "good" for. The initial RFC allows the user to specify a "time to live" for each message. The sender makes a decision based on this "time to live" as to when to stop retransmitting and skip over the data.

The actual mechanism to skip over the data is separate from the methodology used to determine when to skip over data. This means that a sender can have many different "profiles" for skipping data and the receiver does not need to have the same "profile".

Currently the authors are aware of three profiles commonly implemented:

- Time based reliability
- Buffer based reliability
- Number of retransmissions

The time based profile follows RFC3758 in that the sender specifies the number of milliseconds before the message should be skipped. As many retransmissions that are possible will be made in the time allowed. For example, if the application sends the data with a 9,000 millisecond reliability, the message will be retransmitted several times[3] if loss occurs.

Another reliability profile is buffer based. In this form, the user specifies the maximum number of bytes of data that are allowed to be on queue. If the size of the queue specified is reached, then the oldest data (marked in this manner) is looked to be skipped so that the new data can be added to the queue of outgoing data.

The number of retransmissions profile allows an application to specify how many times the data will be retransmitted. So, for example, if the application specified zero times no retransmissions would be made[4].

In all cases, no matter which profile is used, fully reliable and partially reliable data may be mixed. In effect, a single stream may have both reliable and partially reliable data en-queued on it. This also has ramifications to the buffer based method in that if the entire association is filled with fully reliable data, then the new send itself will be the one subject to "skipping."

D. Message boundaries

Message boundary preservation is a small incremental feature added to SCTP that many developers will be happy to see. When you think of an interaction between two applications, rarely do they exchange a "stream of bytes". Rather, they send, receive, and act upon messages. In a TCP connection, each message must be framed in some way as any read of a buffer may return parts of two separate messages. The application

[3]The exact number of times is based on factors such as the minimum and maximum RTO.

[4]This is similar to UDP except that this profile assures that at least one transmission occurs, whereas UDP makes no such assurance

needs to provide application layer code that parses the message based on the way the messages were framed by the sender.

With SCTP, messages are never merged together upon reading. As long as a large enough buffer is provided for reading, each read returns a single message. Each send is considered to be a message in itself[5]. As a consequence, the application does not need to track by sender nor receiver where the message boundaries are.

Note that message boundary preservation does have impact on how messages are transmitted. SCTP is capable of bundling multiple messages together and even splitting a large message over multiple Protocol Data Units (PDUs). How the bundling or splitting occurs is often driven by the size of the messages the user is sending and the Path Maximum Transport Unit (PMTU).

E. Security

Security is a topic of some importance since a transport protocol that is subject to easy attack is not acceptable in today's often hostile Internet. SCTP has several unique features that help strengthen its security to blind attacks, and an optional extension [5] that provides even more capabilities.

Every SCTP association starts with a four way handshake. This four way handshake includes a signed cookie that the passive server side sends to the active initiating client. The server holds no state, and thus is not subject to the SYN flooding attack.

In the first two packets during the setup exchange, a 32-bit random value is supplied by both endpoints. This 32-bit random nonce serves as a verification tag (vtag). All packets sent must include the vtag supplied by the peer during setup of the association, in order to be accepted by the receiving SCTP stack. This means that a blind attacker must generate 2 billion guess (on average) in order to say inject an "ABORT" chunk to tear down the association.

The authentication option RFC4895 [5] adds the ability for any chunk to be required to be signed in such a way so that the receiver can know without doubt that the source was indeed the expected sender. Even applications that decide not to use shared keys can still gain some measure of security assuming that the first two packets that setup an association are not intercepted by a man in the middle.

Note that even though SCTP uses a four way handshake, this does not cause delay in getting the first data messages to the receiver. In fact, due to the way most TCP stack socket API's work, SCTP usually can get the first data chunk to its peer one half of a round trip quicker than TCP.

F. Other differences

There are a few other differences worth noting in any comparison of IP transports. One of the obvious differences is the checksum. In both UDP and TCP, a summation is used of all bits in the message. The summation is done as a simple addition of all bytes in the message including a pseudo-header. The pseudo-header is selected parts of the IP header to help detect when a router misdirects a packet. For SCTP, a Cyclic Redundancy Check is used (CRC32c). A CRC is much stronger than a checksum and provides much better protection against bit errors and packet damage done by routers and other network devices.

The pseudo-header mentioned above is not needed in SCTP. The reason is the vtag discussed earlier. The pseudo-header, as noted, is used to detect misrouted packets in the checksum. For SCTP, the 32-bit random nonce provides this same protection without the need to embed a hidden field in the packets checksum.

Another consequence of the use of a vtag is the absence of the timed-wait state in the protocol state machine. In TCP, for some period of time after an endpoint shuts down, the TCP stack prevents the port from being bound. This period of time is called the "timed-wait" period and normally last for about two minutes. The purpose of this "timed-wait" is to allow lingering packets with the same four-tuple to drain from the network. In SCTP, the vtag, protects us from this same situation as long as vtags themselves are reused in a "timed-wait" manner. As long as a different vtags are used, the same port may be immediately re-used for a new association to the same peer endpoint.

SCTP also includes a required heartbeat mechanism for path management. In contrast, TCP has an optional keep-alive mechanism which must be explicitly enabled by the application.

One other striking difference between TCP and SCTP is the absence of the half closed state. In a TCP connection, one side is allowed to inform the peer that it will send no more data but will continue to accept data from the peer. This state is known as half closed. SCTP does not allow this behavior. If a user closes one side, then the connection will shutdown.

G. A word about optional features

In the preceding sections, we have mentioned several optional features. You might wonder which of them should I expect from my implementation. It is the authors' opinion that a complete SCTP implementation should include:

1) RFC4960 (basic SCTP)
2) RFC3758 (partial reliability)
3) RFC4895 (authentication)
4) RFC5061 (dynamic addresses)

There are other drafts and extensions that are currently being reviewed by the IETF, but those will truly be optional in our opinion. The ones listed above, however, though optional, we consider quite necessary for a full featured SCTP stack.

III. SOCKET API OVERVIEW

Now that we have discussed some of the features and function of SCTP, let us talk about how you can use this powerful new protocol. As with TCP and UDP, the socket API is the most common method of accessing and using SCTP. Before diving deeper into the details of SCTP's socket API we

[5]Note that there are mechanisms to explicitly avoid this premise, but the default behavior is such that every send is a message

will first get an overview of some of the basic elements and choices that an application writer will have when using SCTP.

A. Models

First and foremost, when we go to use SCTP, is that we now have two choices. The SCTP socket API provides two models: the one-to-one model and the one-to-many model.

The one-to-one model is based on a one to one relationship between the socket and a SCTP association (not taking the listening sockets into account). This is similar to a standard TCP socket. For the one-to-many model, there is a one-to-many relationship between the socket and the SCTP-associations. This is similar to using unconnected UDP sockets.

The one-to-one model is basically a "TCP" compatibility model. This model works the same exact way that the standard TCP socket API model works. A server will typically call `socket()`, `bind()`, and `listen()`. Then after the initial setup will sit in a loop calling `accept()` to gain new connections. Each new connection is a new socket descriptor on which the new connection is available to send and receive data on. The application must track each individual socket descriptor for each connection setup. The client will call `socket()` followed by a call to `connect()` to the address of the server.

The major advantage to this model is that a simple change to existing TCP code will allow that code to work with SCTP. To access this model, a user calls `socket(int domain, int type, int proto)` with type set to `SOCK_STREAM` and proto set to `IPPROTO_SCTP`. Note that the domain argument is generally how you choose between IPv6 and IPv4 (`PF_INET6` and `PF_INET`).

The one-to-many model is designed as a peer-to-peer type model. In this model, both sides generally call `socket()` followed by `listen()`. Then, when they wish to exchange information with a peer, they call `sendto()` or `recvfom()` (or any of the extended send or receive calls, see section IV). Note that the one single socket will have multiple associations underneath it. Only one socket descriptor is ever used in this model, calling `accept()` will return an error. One of the advantages to this model is the ability to send data on the third leg of the four way handshake[6]. Another advantage is that an application does not really need to track association state. In order to be truly free of association state, however, the application is recommended to turn on the `AUTO_CLOSE` socket option that will automatically close associations that are idle for long periods.

Accessing the one to many model is done by calling the socket system call specifying the type as `SOCK_SEQPACKET` and the proto as `IPPROTO_SCTP`.

B. Notifications

While developing the socket API for SCTP, it became quickly obvious that the transport stack itself would often have useful things to tell the application (if the application is interested). The existing socket API had no method for the transport to easily communicate events that were happening within the transport. To deal with this, the socket API for SCTP allows an application to subscribe to event "notifications". The subscription is done by setting one of the SCTP socket options. Once one or more events are turned on, and when the SCTP stack has one of those events to tell the application, it sends an event notification message to the application up the normal data path with the flags field set to `MSG_NOTIFICATION`. That is, it multiplexes the event notification messages with peer user messages. When a user application subscribes to these events, it is in effect acknowledging to the SCTP stack that it (the application) understands it must look at the flags field before interpreting a message, since it is possible it is not a message from a peer but from the transport itself.

There are currently eight types of notifications. They are:

- Association events - the starting and closing of new associations
- Address events - information about peer addresses, failures, additions, deletions, confirmations.
- Send Failures - When a send fails, the data is returned with an error if you subscribe to this event.
- Peer Error - If the peer sends an error message the stack would pass the TLV up the stack in this notification.
- Shutdown events - Indications that a peer has closed or shutdown an association.
- Partial delivery events - this notification will indicate issues that may occur in the partial delivery api.
- Adaptation layer event - this notification holds a adaptation indication.
- Authentication event - Various authentication events (such as new keys activated) would be signaled by this notification.

Often applications will not be interested in a number of these notification. Two of the most useful notifications for the one-to-many model are the Association and Shutdown events. Within these events is an association identification often called `assoc_id`. This id identifies a specific association within the one-to-many socket. It can also be used with some of the extended socket API calls used for sending messages (instead of using a socket address). Many of the socket options used with SCTP will take the `assoc_id` as an argument to identify which specific association to change or gather settings on, instead of the whole socket descriptor.

We will discuss notifications in more detail in the next section.

C. Extended calls

All of the existing socket API calls will work with SCTP seamlessly to provide most of all of the functions needed. However, there are limitations for a few corner cases in either the utility or in the ease of use of many of the common API calls. For example, the existing socket API does not adequately address the following cases:

- For binding addresses, you can have one or all addresses.

[6] Note that some implementations do allow this with the one-to-one model at the expense of breaking TCP compatibility.

```
struct sctp_event_subscribe {
 uint8_t sctp_data_io_event;
 uint8_t sctp_association_event;
 uint8_t sctp_address_event;
 uint8_t sctp_send_failure_event;
 uint8_t sctp_peer_error_event;
 uint8_t sctp_shutdown_event;
 uint8_t sctp_partial_delivery_event;
 uint8_t sctp_adaptation_layer_event;
 uint8_t sctp_authentication_event;
#ifdef __FreeBSD__
 uint8_t sctp_stream_reset_events;
#endif
};
```

Fig. 3. The SCTP Event Subscription structure

- For connecting to a peer, you can connect to one and only one address.
- For sending or receiving, getting access to the stream information is rather awkward.

To solve these problems, there are extensions to the socket API to add additional capabilities as well as make some existing features easier to use. New "system calls" have been added specifically for SCTP. Note that a new system call may be no more than a utility library routine that eases access to a specific feature.

We will discuss each of the extended socket API calls as well as other features such as notifications and socket options in IV.

IV. Socket API - details

Now that you have a general idea of the advantages of SCTP, you are probably wondering how you access these features. This section will try to highlight and provide a rough overview of the way a user can best interact with SCTP.

A. Notifications

As mentioned earlier SCTP can provide information to the user via notifications. Notifications are received in the data path i.e. by `recvmsg()` or `sctp_recvmsg()`. You could in theory use `recvfrom()` or `recv()` but then you would have no way of knowing if the message was a notification from SCTP or a real peer message. All notifications are disabled by default, so an application must set a socket option (`SCTP_EVENTS`) to turn on one or more notifications. Once you enable a notification, you are making an implicit pledge to the SCTP stack that you will not use `recvfrom()` or `recv()`. If you violate that pledge, you will most likely be confused by messages arriving from both the peer application as well as the SCTP stack.

The `SCTP_EVENTS` socket option takes has input a structure as shown in Fig. 3. Note that each `uint8_t` field is interpreted as a boolean, where a zero value turns off the notification and a non-zero value turns on the notification.

```
struct sctp_sndrcvinfo {
 uint16_t sinfo_stream;
 uint16_t sinfo_ssn;
 uint16_t sinfo_flags;
 uint16_t sinfo_pr_policy;
 uint32_t sinfo_ppid;
 uint32_t sinfo_context;
 uint32_t sinfo_timetolive;
 uint32_t sinfo_tsn;
 uint32_t sinfo_cumtsn;
 sctp_assoc_t sinfo_assoc_id;
};
```

Fig. 4. The SCTP sndrcvinfo

```
struct sctp_assoc_change {
 uint16_t sac_type;
 uint16_t sac_flags;
 uint32_t sac_length;
 uint16_t sac_state;
 uint16_t sac_error;
 uint16_t sac_outbound_streams;
 uint16_t sac_inbound_streams;
 sctp_assoc_t sac_assoc_id;
};
```

Fig. 5. Association notification

The careful reader will notice two distinctions. There is an extra event for FreeBSD (found in the #ifdefs), and there is a "notification" that was not listed previously, `sctp_data_io_event`. The data_io event is not really a notification, but the ability to receive extra information as ancillary data with the receive calls. The extra information tells you the stream number as well as other ancillary data. If you wish to receive this information you must enable the "event" just like you would for a notification.

Note when the `sctp_data_io_event` event is enabled you will receive the `sctp_sndrcvinfo` structure with every `recvmsg()` or `sctp_recvmsg()` call. You cannot receive this information with the `recv()` or `recvfrom()` calls. In the `sctp_recvmsg()` call, the `sctp_sndrcvinfo` structure is an argument passed in the call. For the `recvmsg()` call, the user will need to parse ancillary data for type `SCTP_SNDRCV`. The `sctp_sndrcvinfo` is shown in Fig. 4.

Each notification, as mentioned, provides you with a specific structure. We will examine two of the notifications and leave the rest as an exercise for the reader. [7] and [8] may also be helpful.

One of the most common notifications an application will be interested in is the association events notification. This notification tells you about changes in associations, including the arrival of new associations. Refer to Figure 5 for the full structure. Some notable fields from this structure are `sac_type`, `sac_flags`, and `sac_length`. Every notifi-

```
struct sctp_paddr_change {
 uint16_t spc_type;
 uint16_t spc_flags;
 uint32_t spc_length;
 struct sockaddr_storage spc_aaddr;
 uint32_t spc_state;
 uint32_t spc_error;
 sctp_assoc_t spc_assoc_id;
 uint8_t spc_padding[4];
};
```

Fig. 6. Address change notification

cation begins with these type, flags and length fields. This allows the receiver to cast the structure to a base notification structure and then examine the type to know exactly which notification has arrived. In this particular notification, an application may be interested in `sac_outbound_streams` and `sac_inbound_streams`, which tells the user how many streams were negotiated (note this may not be the number expected by the application and the two values are not necessarily equal). Another useful field is the `sac_assoc_id`. As indicated earlier, this field uniquely identifies this association and can actually be used as a destination when using the advanced SCTP API send calls.

Another notification often subscribed to can be found in Figure 6. This notification arrives when some event has occurred concerning one of the peer's addresses. The `spc_state` field will reflect what happened with the address. Possible address events include:

- It was added (`SCTP_ADDR_ADDED`).
- It was deleted (`SCTP_ADDR_REMOVED`).
- It is now reachable (`SCTP_ADDR_AVAILABLE`).
- It is now un-reachable (`SCTP_ADDR_UNREACHABLE`).

For fault tolerant applications, tracking the states of the peer's addresses may well be an essential job.

B. Extended system calls

SCTP provides some additional system calls:

- `sctp_recvmsg(int sd, void *msg, size_t len, struct sockaddr *from, socklen_t *fromlen, struct sctp_sndrcvinfo *sinfo, int *flags)`
- `sctp_sendmsg(int s, void *msg, size_t len, struct sockaddr *to, socklen_t tolen, uint32_t ppid, uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t context)`
- `sctp_send(int sd, void *msg, size_t len, struct sctp_sndrcvinfo *sinfo, int flags)`
- `sctp_bindx(int sd, struct sockaddr *addrs, int addrcnt, int type)`
- `sctp_connectx(int sd, struct sockaddr *addrs, int addrcnt, sctp_assoc_t *assoc_id)`
- `sctp_sendx(int sd, void *msg, size_t len, struct sockaddr *addrs, int addrcnt, struct sctp_sndrcvinfo *, int flags)`
- `sctp_sendmsgx(int s, void *msg, size_t len, struct sockaddr *to, int addrcnt, socklen_t tolen, uint32_t ppid, uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t context)`

The first three essentially provide some convenience functions to the user. Their functionality can be provided by using equivalent `sendmsg()` and `recvmsg()` calls with the appropriate handling of the required ancillary data.

The last four functions are necessary to make the support of multihoming possible. `sctp_bindx()` makes it possible to bind an arbitrary set of local addresses to a socket, rather than the "all or none" that `bind()` provides. `sctp_connectx()`, `sctp_sendx()` and `sctp_sendx()` make it possible to use multiple known addresses of the peer already *during* the association setup. Without these functions, multihoming would only be available after the association has been completely established.

C. Socket Options

SCTP provides a large set of socket options as shown in TABLE II, and some FreeBSD specific extensions as shown in TABLE III. This is due to the fact that SCTP allows a lot of protocol parameters to be controlled by the user. Also, some protocol extensions, like the SCTP-AUTH [5] extension, use socket options to control the feature (e.g. controls hash algorithms, keys, chunks to authenticate, etc.). The usage of a few of these options are described in the examples in the next section. For a detailed description, see [7].

D. Some Examples

In this section, we will provide example code for a simple client which sends a number of messages to a server, and a server which just discards all received messages. The client will use the one-to-one model and the server will use the one-to-many model. Of course, despite using different programming models, the client and server can still communicate properly.

Let us first consider the server in Figure 7. In line 22, a one-to-many style socket is created. A method of enabling all notifications is shown in lines 25–27. Then in lines 31–38, the socket is bound to the IPv4 wildcard address and the well known port for the discard service. To allow the kernel to accept SCTP associations on the discard port, the socket is put into listening mode in line 41. The reception of messages and notifications is handled by the infinite loop in lines 45–62. `sctp_recvmsg()` is used in line 51 to read event notifications or peer user messages, after initializing several variables in lines 46–49. If a notification is received, only a simple message is printed. For a received user message, the length, source address and port number, stream identifier

Option	R-W	Description
SCTP_RTOINFO	rw	RTO min/max
SCTP_ASSOCINFO	rw	Association parameters
SCTP_INITMSG	rw	Setup options
SCTP_NODELAY	rw	Nagle algorithm
SCTP_AUTOCLOSE	rw	Automatic closing
SCTP_SET_PEER_PRIMARY_ADDR	rw	Remote primary
SCTP_PRIMARY_ADDR	rw	Local primary
SCTP_ADAPTATION_LAYER	rw	AI indication
SCTP_DISABLE_FRAGMENTS	rw	Fragmentation
SCTP_PEER_ADDR_PARAM	rw	Misc parameters
SCTP_DEFAULT_SEND_PARAMS	rw	Default sendrcvinfo
SCTP_EVENTS	rw	Notifications
SCTP_I_WANT_MAPPED_V4_ADDR	rw	Mapped v4 addresses
SCTP_MAXSEG	rw	Fragmentation point
SCTP_DELAYED_SACK	rw	Delayed sack
SCTP_FRAGMENT_INTERLEAVE	rw	Receive interleave
SCTP_PARTIAL_DELIVERY_POINT	rw	receive PD point
SCTP_AUTH_CHUNK	w	Add Auth Chunk
SCTP_AUTH_KEY	w	Add Auth key
SCTP_HMAC_IDENT	rw	hmac algo
SCTP_AUTH_ACTIVE_KEY	rw	Active key
SCTP_AUTH_DELETE_KEY	w	Delete key
SCTP_USE_EXT_RCVINFO	rw	Extended sndrcvinfo
SCTP_AUTO_ASCONF	rw	Automatic IP add/del
SCTP_MAXBURST	rw	Microburst control
SCTP_CONTEXT	rw	Default context
SCTP_EXPLICIT_EOR	rw	Explicit EOR
SCTP_STATUS	r	Assoc status
SCTP_GET_PEER_ADDR_INFO	r	Info on dest
SCTP_PEER_AUTH_CHUNKS	r	Peer requires auth
SCTP_LOCAL_AUTH_CHUNKS	r	Local requires auth
SCTP_GET_ASSOC_NUMBER	r	Number of assocs
SCTP_GET_ASSOC_ID_LIST	r	Assoc ids

TABLE II
SCTP SOCKET OPTIONS

Option	R-W	Description
SCTP_RESET_STREAMS	w	Stream reset
SCTP_SET_DEBUG_LEVEL	rw	Debug output
SCTP_CMT_ON_OFF	rw	CMT on/off
SCTP_CMT_USE_DAC	rw	DAC with CMT
SCTP_PLUGGABLE_CC	rw	Set CC
SCTP_GET_SNDBUF_USE	r	Send space
SCTP_GET_NONCE_VALUES	r	Vtag pair
SCTP_SET_DYNAMIC_PRIMARY	w	Global primary
SCTP_GET_PACKET_LOG	r	Packet log
SCTP_VRF_ID	rw	Default VRF
SCTP_ADD_VRF_ID	w	Add a VRF
SCTP_GET_VRF_IDS	r	Get VRF IDs
SCTP_GET_ASOC_VRF	r	Assoc VRF ID
SCTP_DEL_VRF_ID	w	Delete VRF ID

TABLE III
SCTP FREEBSD SPECIFIC SOCKET OPTIONS

and payload protocol identifier is printed. It should be noted that this simple program does not check if the notification or message was completely received by checking for the `MSG_EOR` flag. In line 64, the socket would be closed, but this line will never be executed.

The second example is a simple client shown in Figure 8, which sends a number of messages of the same length to a discard server.

In line 27, a one-to-one style socket is created. In lines 31–36, the number of outgoing streams to be requested during association setup is set to 2048. Then the association is established in line 48. Since the number of incoming and outgoing streams is negotiated during the association setup, the number of streams is retrieved in lines 50–54 via a socket option. Then all messages are sent via a `sctp_sendmsg()` call in lines 57–60. A given payload protocol identifier is used, and the stream number to send with is used in a round robin fashion. In line 65, the association teardown procedure is started by closing the socket.

V. CONCLUSION

This paper describes the features provided by SCTP and gives a glimpse into the many ways an application can control and configure it. SCTP has been designed to be flexible and yet provide reasonable defaults for applications that do not wish to dig into the deep details of controlling the transport. For applications that need more control, SCTP provides a wide host of socket options and a multitude of ordering options.

In general, SCTP can be used any place TCP can be used and gives the application greater flexibility. No longer does the application need to frame messages, the transport does that for you. No longer does the application need to worry about connection state (when using the one to many socket model). SCTP may also be used in cases where one might consider UDP, assuming a full featured implementation of SCTP including Partial Reliability.

The authors encourage you to go and explore SCTP it will become addictive.

Happy SCTPing.

REFERENCES

[1] J. Postel, "User Datagram Protocol", *RFC 768*, August 1980.
[2] J. Postel, "Transmission Control Protocol", *RFC 793*, September 1981.
[3] M. Tüxen et al., "Stream Control Transmission Protocol Partial Reliability Extension", *RFC 3758*, May 2004.
[4] R. Stewart, "Stream Control Transmission Protocol", *RFC 4960*, September 2007.
[5] R. Stewart et al., "Authenticated Chunks for the Stream Control Transmission Protocol", *RFC 4895*, August 2007.
[6] R. Stewart et al., "Stream Control Transmission Protocol Dynamic Address Reconfiguration", *RFC 5061*, September 2007.
[7] R. Stewart et al, "Socket API Extensions for Stream Control Transmission Protocol (SCTP)", *draft-ietf-tsvwg-sctpsocket-16.txt*, work in progress.
[8] R. Stevens et al., "UNIX Network Programming Volume 1 Third Edition"', Addison Wesley, 2004.

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <netinet/sctp.h>
5  #include <arpa/inet.h>
6  #include <string.h>
7  #include <stdio.h>
8  #include <unistd.h>
9
10 #define BUFFER_SIZE (1<<16)
11 #define PORT 9
12 #define ADDR "0.0.0.0"
13
14 int main(int argc, char *argv[]) {
15         int fd, n, flags;
16         struct sockaddr_in addr;
17         socklen_t from_len;
18         struct sctp_sndrcvinfo sinfo;
19         char buffer[BUFFER_SIZE];
20         struct sctp_event_subscribe event;
21
22         if ((fd = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0) {
23                 perror("socket");
24         }
25         memset((void *)&event, 1, sizeof(struct sctp_event_subscribe));
26         if (setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS,
27                 &event, sizeof(struct sctp_event_subscribe)) < 0) {
28                 perror("setsockopt");
29         }
30
31         memset((void *)&addr, 0, sizeof(struct sockaddr_in));
32 #ifdef HAVE_SIN_LEN
33         addr.sin_len         = sizeof(struct sockaddr_in);
34 #endif
35         addr.sin_family      = AF_INET;
36         addr.sin_port        = htons(PORT);
37         addr.sin_addr.s_addr = inet_addr(ADDR);
38         if (bind(fd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in)) < 0) {
39                 perror("bind");
40         }
41         if (listen(fd, 1) < 0) {
42                 perror("listen");
43         }
44
45         while (1) {
46                 flags = 0;
47                 memset((void *)&addr, 0, sizeof(struct sockaddr_in));
48                 from_len = (socklen_t)sizeof(struct sockaddr_in);
49                 memset((void *)&sinfo, 0, sizeof(struct sctp_sndrcvinfo));
50
51                 n = sctp_recvmsg(fd, (void*)buffer, BUFFER_SIZE,
52                                  (struct sockaddr *)&addr, &from_len,
53                                  &sinfo, &flags);
54
55                 if (flags & MSG_NOTIFICATION) {
56                         printf("Notification received.\n");
57                 } else {
58                         printf("Msg of length %d received from %s:%u on stream %d, PPID %d.\n",
59                                n, inet_ntoa(addr.sin_addr), ntohs(addr.sin_port),
60                                sinfo.sinfo_stream, ntohl(sinfo.sinfo_ppid));
61                 }
62         }
63
64         if (close(fd) < 0) {
65                 perror("close");
66         }
67
68         return (0);
69 }
```

Fig. 7. Discard server using the one-to-many model.

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <netinet/sctp.h>
5  #include <arpa/inet.h>
6  #include <string.h>
7  #include <stdio.h>
8  #include <unistd.h>
9
10 #define PORT 9
11 #define ADDR "127.0.0.1"
12 #define SIZE_OF_MESSAGE    1000
13 #define NUMBER_OF_MESSAGES 10000
14 #define PPID 1234
15
16 int main(int argc, char *argv[]) {
17         unsigned int i;
18         int fd;
19         struct sockaddr_in addr;
20         char buffer[SIZE_OF_MESSAGE];
21         struct sctp_status status;
22         struct sctp_initmsg init;
23         socklen_t opt_len;
24
25         memset((void *)buffer, 'A', SIZE_OF_MESSAGE);
26
27         if ((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) < 0) {
28                 perror("socket");
29         }
30
31         memset((void *)&init, 0, sizeof(struct sctp_initmsg));
32         init.sinit_num_ostreams = 2048;
33         if (setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG,
34                        &init, (socklen_t)sizeof(struct sctp_initmsg)) < 0) {
35                 perror("setsockopt");
36         }
37
38         memset((void *)&addr, 0, sizeof(struct sockaddr_in));
39 #ifdef HAVE_SIN_LEN
40         addr.sin_len         = sizeof(struct sockaddr_in);
41 #endif
42         addr.sin_family      = AF_INET;
43         addr.sin_port        = htons(PORT);
44         addr.sin_addr.s_addr = inet_addr(ADDR);
45
46         if (connect(fd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in)) < 0) {
47                 perror("connect");
48         }
49
50         memset((void *)&status, 0, sizeof(struct sctp_status));
51         opt_len = (socklen_t)sizeof(struct sctp_status);
52         if (getsockopt(fd, IPPROTO_SCTP, SCTP_STATUS, &status, &opt_len) < 0) {
53                 perror("getsockopt");
54         }
55
56         for (i = 0; i <  NUMBER_OF_MESSAGES; i++) {
57                 if (sctp_sendmsg(fd, (const void *)buffer, SIZE_OF_MESSAGE,
58                                  NULL, 0,
59                                  htonl(PPID), 0, i % status.sstat_outstrms,
60                                  0, 0) < 0) {
61                         perror("send");
62                 }
63         }
64
65         if (close(fd) < 0) {
66                 perror("close");
67         }
68         return(0);
69 }
```

Fig. 8.   Discard client using the one-to-one model.