Linux Kernel SCTP : The Third Transport

La Monte H.P. Yarroll
Motorola GTSS
piggy@acm.org

Karl Knutson
Motorola GTSS
karl@athena.chicago.il.us

Abstract

The Stream Control Transmission Protocol (SCTP) is a reliable message-oriented protocol with transparent support for multihoming. It allows multiple independent complex exchanges which all share a single connection and congestion context.

We provide an overview of the protocol, the UDP-style API and the details of the Linux kernel reference implementation. The brief API discussion is intended for developers wishing to use SCTP. The detailed implementation discussion is for developers interested in contributing to the kernel development effort.

1 Introduction

The developers at the Linux 2.5 Kernel Summit in San Jose achieved a rough consensus that 2.5 should probably support SCTP, a new transport protocol from the IETF. This paper introduces the ongoing work on such an implementation, providing some details for both the application developer and the kernel developer.

The Stream Control Transmission Protocol (SCTP) is a reliable message-oriented protocol with transparent support for multihoming. It allows multiple independent complex exchanges which all share a single connection and congestion context.

1.1 History of SCTP

The SIGTRAN (Signalling Transport) Working Group of the IETF is concerned with the transport of telephony signalling data over IP. Upon reviewing the available standard transport protocols, they concluded that none of them met the transport requirements of signalling data.

SIGTRAN concluded that they needed a new transport protocol which could provide reliable message delivery, tolerate network failures, and avoid the head-of-line-blocking problem. We will discuss this problem later.

The WG selected a proposal from Randall Stewart and Qiaobing Xie of Motorola as a starting point. Stewart and Xie had developed a Distributed Processing Environment, Quantix, aimed at telephony applications. This DPE had been successfully demonstrated at Geneva Telecom in 1999.

The Working Group took great care in constructing the new protocol, SCTP, incorporating many lessons learned from TCP, such as congestion control, selective ACK, message fragmentation and bundling.

The core transport protocol from Quantix brought support for multihoming, message framing, and streams. We discuss all of these features at length later.

The IESG decided that the resulting protocol was robust enough to be elevated from a specialised transport for telephony signalling to a new general purpose transport to stand beside UDP and TCP. To this end, they moved the work from SIGTRAN to TSVWG, the general transport group.

As of this writing, the core specification, [RFC2960], is at Proposed Standard. There have been three successful bakeoffs covering over 25 separate implementations. Lessons learned from the most recent bakeoff are being written up in an "Implementor's Guide", [SCTPIMPL].

1.2 SCTP in the Linux kernel

Shortly before the first bakeoff, the IESG asked SIGTRAN to move SCTP from riding on UDP to riding directly on top of IP. The long term goal was clearly was to move SCTP from user space into the kernel.

Aside from the obvious performance gains, this has the effect of reducing the number of implementations to roughly one per operating system. This makes it easier to verify the stability of most of the implementations which appear on the Internet.

Randall Stewart saw the importance of this and started one of the authors of this paper working on a port of the user space implementation to the Linux kernel. This port was intended as a reference for developers of implementations for other kernels to examine. The Linux kernel implementation has since diverged significantly from the user space reference, but maintains the standards of a reference implementation (see Coding Standards, below).

1.3 SCTP examples

SCTP is a reliable message-oriented protocol with transparent support for multihoming. It allows multiple independent complex exchanges which all share a single connection and congestion context.

Many network applications operate by exchanging simultaneously, short, similar sequences of data continuously. The traffic produced by these operations can be characterised as MICE (Multiple Independent Complex Exchanges). It is also true that many applications which use MICE also have high network reliability requirements.

1.3.1 A database app

One example is a client/server database application. Each request and each response is a message. Each transaction is a sequence of dependent request/response pairs.

Implemented over TCP, this application would have to provide its own message boundaries, since TCP sends bytes, not messages. How do we implement MICE with TCP? We have two ways of doing this: multiple connections, or a single multiplexed and reused connection.

With each transaction over a separate TCP connection, we gain the independence of transactions, but at a cost in performance. Since TCP (as a general purpose transport protocol) uses congestion control, each of the connections would have to go through slow-start and if most transactions were short, they would never get out of slow-start.

With all transactions over a single TCP connection, we make efficient use of the network bandwidth, but open ourselves up to the head-of-line blocking problem. This means that if one segment in one transaction is lost, this blocks all transactions, not just the one with the lost segment.

If we use SCTP for the same application we gain the benefits of using TCP, as well as advantages peculiar to SCTP. SCTP directly supports messages and guarantees TCP-like levels of bandwidth efficiency via bundling and fragmentation. Each database transaction can be represented as an ordered stream of messages, which are independent in SCTP for retransmission purposes. This means that while SCTP has the same congestion control mechanisms as TCP, it does not have to resort to multiple connections nor is it vulnerable to the head-of-line blocking problem.

1.3.2 A free clinic

Another example of SCTP use is for a free* clinic which needs a reliable way to use its IP-networked patient monitoring software.

This has many similarities to the example above in that different monitoring devices would need to send simultaneous information—multiple independent complex exchanges. The main difference is in the higher network reliability requirements.

A reasonable way to improve the network reliability is to set up a parallel network and use multihoming for the client and server applications. However, if the application is TCP-based, the multihoming needs to be added to the application. With SCTP, the multihoming ability is built into the protocol. All that is necessary is to make the appropriate socket calls and SCTP will take advantage of the

*Free as in "free beer".

addresses available in the existing network. This also applies if one side of the connection has more addresses than the other.

2 The UDP-style API

Any new protocol needs an API. In particular for an Internet protocol, it's important to have the API match the API normally used for IP networks. This is the Berkeley sockets model—the SCTP version is defined in the Internet Draft "Sockets API Extensions for SCTP"[SCTPAPI]. The API draft defines two complementary interfaces to SCTP–one for compatibility with older TCP-based applications, and another for new applications designed expressly to use SCTP. The Linux Kernel SCTP stack does not yet implement the former, so we discuss only the UDP-style interface.

The conceptual model of the UDP-style API is (naturally) that of plain UDP. To send a message in UDP, you create a socket, bind an address to it and send your message using `sendmsg()`. To receive a message in UDP, you create a socket, bind an address to it and use `recvmsg()`. It's much the same with the UDP-style API for SCTP. To send a message, you create a socket, bind *addresses* to it and use `sendmsg()`. The SCTP stack underlying the API handles association startup and shutdown automatically. The same goes for message reception. To receive a message in UDP-style, you create a socket, bind *addresses* to it and use `recvmsg()`.

The important API differences between UDP and UDP-style SCTP are: multihoming; ancillary data; and the option of notifications from the SCTP stack.

2.1 Multihoming and `bindx()`

There are three ways to work with multihoming with SCTP. One is to ignore multihoming and use one address. Another way is to bind all your addresses through the use of `INADDR_ANY` or `IN6ADDR_ANY`. This will "associate the endpoint with the optimal subset of available local interfaces."(Section 3.1.2, [SCTPAPI]) The most flexible way is through the use of `sctp_bindx()`, which allows additional addresses to be added to a socket after the first one is bound with `bind()`, but before the socket is used to transfer or receive data. The function `sctp_bindx()` is further described in section 8.1 of [SCTPAPI].

2.2 Ancillary data

To use streams with the UDP-style API, you use ancillary data in the `struct cmsghdr` part of the `struct msghdr` argument to both `sendmsg()` and `recvmsg()`. Ancillary data is used for initialisation data (`struct sctp_initmsg` and for header data (`struct sctp_sndrcvinfo`).

Ancillary data are manipulated with the macros `CMSG_FIRSTHDR, CMSG_NEXTHDR, CMSG_DATA, CMSG_SPACE,` and `CMSG_LEN`. These are all defined in [**?**]. [SCTPAPI] provides a nice example in section 5.4.2.

```
struct sctp_initmsg {
   uint16_t sinit_num_ostreams;
   uint16_t sinit_max_instreams;
   uint16_t sinit_max_attempts;
   uint16_t sinit_max_init_timeo;
};
```

The initialisation ancillary data sets information for starting new associations.

```
struct sctp_sndrcvinfo {
   uint16_t sinfo_stream;
   uint16_t sinfo_ssn;
   uint16_t sinfo_flags;
   uint32_t sinfo_ppid;
   uint32_t sinfo_context;
   uint8_t sinfo_dscp;
   sctp_assoc_t sinfo_assoc_id;
};
```

The header ancillary data reports information gleaned from the SCTP headers. If requested with the `SCTP_RECVDATAIOEVNT` socket option, this ancillary data is provided with every inbound data message. There is a handy key (`sinfo_assoc_id`) which identifies the association for this particular message. It also provides the flags needed to implement partial delivery of very large messages.

Outbound messages should include an `sctp_sndrcvinfo` ancillary data structure to

tell SCTP which SCTP stream to put this datagram into. It is also possible to set a default stream so that this ancillary data may be omitted.

2.3 Notifications

SCTP provides for the concept of optional notifications. These are messages delivered in-band about events inside the SCTP stack, such as a destination transport address failure or a new association coming up. The notifications are marked with the `MSG_NOTIFICATION` flag in the `msg_flags` field of the `sctp_sendrcvinfo` ancillary data. The notification is delivered as the body of the message returned by `recvmsg()`.

In **??** we find a table of notifications. Each notification delivers its own data structure which shares the same name (lower case, naturally) as the notification type itself. The first field of every notification is a `uint16_t` which caries the notification type.

3 The lksctp Project

A critical factor in the success of any new IETF protocol is of course a Linux implementation. Fortunately, key personnel at Motorola recognised this and encouraged us to tackle such a project. Months later, we have a core implementation with an ever-expanding feature set. We now have significant participation from developers at IBM and Intel and the pace is picking up.

3.1 Coding standards

In addition to the usual requirements of kernel code, our code seeks to be a useful reference for people making their own kernel implementations of SCTP. If a reader has some question about how to implement a particular section of the RFC, they need only grep for the relevant text in our code and they can find an example. As much as practical, we draw names directly from the RFC. We made the state machine into an explicit table (see 2 for an excerpt) with names that refer directly back to the relevant section numbers. Clarity is a compelling requirement for our code.

3.2 Extreme Programming

As the project grew and we added developers, we clearly needed some way of coordinating our work. We decided to experiment with Extreme Programming, [XP].

XP is a collection of practices aimed at controlling risk in a small to medium-sized software development project. One important principle is that you should do the simplest thing that could possibly work. A second important principle is to take advantage of the fact that programmers like to code.

We use a range of XP practices, but the practices which are most visible to anybody who reads or works on lksctp are the tests and the metaphors.

4 The Tests

One of the XP practices we use is code-to-the-test. XP asks, "If testing is good, why don't we do it all the time?" Instead of writing tests for working code, write tests first, and then write code to pass the tests. This practice leads to a large automated test suite which runs several times per day.

We use three kinds of test, unit tests, test frame functional tests, and live kernel functional tests.

The most basic form of test is the unit test. Unit tests exercise all the interfaces of a particular object and confirm that it behaves correctly. They also encode regression checks for fixed bugs. These tests all have names beginning with `test_`.

The second form of test is the test frame functional tests. These are the tests with names beginning with `ft_frame_`. These tests check for external behaviours of the system, but with a simulated kernel. The simulated kernel is very light weight and gives us very fine control over things like timing and network properties.

Ideally, functional tests should be written by the customer for a system—they encode the behaviours that the customer expects. In our case, we play the role of customer on behalf of the RFC. We also use test frame functional tests to define work items for off-site development groups. The off-site group

Type	Socket Option	Description
SCTP_ASSOC_CHANGE	SCTP_RECVASSOCEVNT	Change of association
SCTP_PEER_ADDR_CHANGE	SCTP_RECVADDREVNT	Change in status of a given address
SCTP_REMOTE_ERROR	SCTP_RECVPEERERR	An error received from a peer
SCTP_SEND_FAILED	SCTP_RECVSENDFAILEVNT	A failure to send
SCTP_SHUTDOWN_EVENT	SCTP_RECVDOWNEVNT	The reception of a SHUTDOWN chunk

Figure 1: Useful notifications for an SCTP socket

writes tests which describe the feature they intend to implement and submits those tests as a proposal. This has proven an excellent medium for describing work.

The final form of test we use is the live kernel functional test. We have many fewer of these than we would like—they are difficult to run since we must install and boot a kernel to test. This is much more work than simply running `make unit_test`. We are exploring UML as a possible way to automate our kernel functional tests. These tests have names beginning with `ft_kern_`.

Code-to-the-test is a practice which you can introduce at any point in a project. When you first start, it seems that you are spending more time writing tests than writing code, but once you begin to have a critical mass of interacting tests you begin to see significant payoffs in both code quality and development velocity.

We have had several incidents where interactions between unit tests and functional tests have uncovered complimentary masking bugs.

Tests are not a substitute for understanding code—they are a mechanism for encoding that understanding to share with other developers, including future versions of yourself. You can learn nearly as much about our code by reading our tests as by reading the code itself.

Lately, we have begun using functional tests to encode major bugs. These are among the best of all possible bug reports—they describe the failure precisely and tell exactly when the problem is gone. After the bugs are fixed the tests serve as part of the regression suite.

5 The Metaphors

XP projects are built around a unifying metaphor rather than an elaborate architecture. In our case, we chose two metaphors which could serve quite well for nearly any protocol development project.

Our metaphors are the state machine and the smart pipe. Most readers are probably familiar with the state machine, but the smart pipe is a twist on a familiar concept. The idea behind a smart pipe[†] is that raw stuff goes in one end and cooked stuff comes out the other end.

5.1 The State Machine

The state machine in our implementation is quite literal. We have an explicit state table which keys to specific state functions which are tied directly back to parts of the RFC. The core of the state machine (found in `sctp_do_sm()`) is almost purely functional—only header conversions are permitted. Each state function produces a description of the side effects (in the form of a `struct sctp_sm_retval`) needed to handle the particular event. A separate side effect processor, `sctp_side_effects()`, converts this structure into actions.

Events fall into four categories. The RFC is very explicit about state transitions associated with arriving chunks. The RFC discusses transitions due to primitive requests from upper layers, but many of these are implementation dependent. The third category of events is timeouts. The final category is a catch-all for odd events like queues emptying.

In order to create an explicit state machine, it was necessary to first create an explicit state table. The

[†]An alternate term may be “oven”.

State:	CLOSED	COOKIE-WAIT	COOKIE-ECHOED	ESTABLISHED
Chunks				
INIT	do_5_1B_init	do_5_2_1_siminit	do_5_2_1_siminit	do_5_2_2_dupinit
INIT ACK	discard(5.2.3)	do_5_1C_ack	discard(5.2.3)	discard(5.2.3)
COOKIE ECHO	do_5_1D_ce	do_5_2_4_dupcook	do_5_2_4_dupcook	do_5_2_4_dupcook
COOKIE ACK	discard	discard(5.2.5)	do_5_1E_ca	discard(5.2.5)
DATA	tabort_8_4_8	discard(6.0)	discard(6.0)	eat_data_6_2
SACK	tabort_8_4_8	discard(6.0)	eat_sack_6_2_1	eat_sack_6_2_1
Timeouts				
T1-INIT TO	bug	do_4_2_reinit	bug	bug
T3-RTX TO	bug	bug	do_6_3_3_retx	do_6_3_3_retx
Primitives				
PRM_ASSOCIATE	do_PRM_ASOC	error	error	error
PRM_SEND	error	do_PRM_SENDQ6.0	do_PRM_SENDQ6.0	do_PRM_SEND

Figure 2: Portion of SCTP state table showing association initialisation

process of creating this table uncovered a few minor contradictions in one of the drafts of the RFC. These mostly involved conflicting catch-all cases. In Figure 1 we have an excerpt which shows the state functions involved in initialising a new association.

5.2 The Smart Pipes

Each smart pipe has one or more structures which define its internal data, and a set of functions which define its external interactions. In this respect these smart pipes can be considered a type of object, in the OO sense. All of these definitions can be found in the include file `<net/sctp/sctpStructs.h>`.

Most of our smart pipes have push inputs—external objects explictly put things in by calling methods directly. A pull input is possible—the smart pipe would need to have a way to register a callback function which can fetch more input in response to some other stimulus.

Some of our pipes use pull outputs. E.g. `SCTP_ULPqueue` passes data and notifications up the protocol stack through explicit calls to the socket functions, usually `readmsg(2)`. Some of our smart pipes use push outputs. E.g. `SCTP_outqueue` has a set of callback functions which it invokes when it needs to send chunks out toward the wire.

There are four smart pipes in lksctp. They are `SCTP_inqueue`, `SCTP_ULPqueue`, `SCTP_outqueue`, and `SCTP_packet`. The first two carry information up the stack from the wire to the user; the second two carry information back down the stack.

5.2.1 SCTP_inqueue

`SCTP_inqueue` accepts packets and provides chunks. It is responsible for reassembling fragments, unbundling, tracking received TSN's for acknowledgement, and managing rwnd for congestion control. There is an `SCTP_inqueue` for each endpoint (to handle chunks not related to a specific association) and one for each association.

The function `sctp_v4_rcv()` (which is the receiving function for SCTP registered with IPv4) calls `sctp_push_inqueue()` to push packets into the input queue for the appropriate association or endpoint. The function `sctp_push_inqueue()` schedules either `sctp_bh_rcv_asoc()` or `sctp_bh_rcv_ep()` on the immediate queue to complete delivery. These functions call `sctp_pop_inqueue()` to pull data out of the `SCTP_inqueue`. This function does most of the work for this smart pipe.

The functions `sctp_bh_rcv_ep()` and `sctp_bh_rcv_asoc()` run the state machine on incoming chunks. Among many other side effects, the state machine can generate events for an upper-layer-protocol (ULP), and/or chunks to go back out on the wire.

5.2.2 SCTP_ULPqueue

SCTP_ULPqueue is the smart pipe which accepts events (either user data messages or notifications) from the state machine and delivers them to the ULP through the sockets layer. It is responsible for delivering streams of messages in order. There is one SCTP_ULPqueue for every endpoint, but this is likely to change at some point to one SCTP_ULPqueue for each socket. This smart pipe uses a data structure distributed between the struct SCTP_endpoint and the struct SCTP_association.

The state machine, sctp_do_sm(), pushes data into an SCTP_ULPqueue by calling sctp_push_chunk_ULPqueue(). It pushes notifications with sctp_push_event_ULPqueue(). The sockets layer extracts events from an SCTP_ULPqueue with sctp_pop_ULPqueue().

5.2.3 SCTP_outqueue

SCTP_outqueue is responsible for bundling logic, transport selection, outbound congestion control, fragmentation, and any necessary data queueing. It knows whether or not data can go out onto the wire yet. With one exception noted below, every outbound chunk goes through an SCTP_outqueue attached to an association. The state machine injects chunks into an SCTP_outqueue with sctp_push_outqueue(). They automatically push out the other end through a small set of callbacks which are normally attached to an SCTP_packet.

The state machine is capable of putting a fully-formed packet directly on the wire. At this point only ABORT uses this feature. It is likely that we will refactor INIT ACK generation again to use this feature.

5.2.4 SCTP_packet

An SCTP_packet is a lazy packet transmitter associated with a specific transport. The upper layer pushes data into the packet, usually with sctp_transmit_chunk(). The packet blindly bundles the chunks. If the it fills (hits the PMTU for its transport), it transmits the packet to make room for the new chunk. SCTP_packet rejects packets which need fragmenting. It is possible to force a packet to transmit immediately with sctp_transmit_packet(). SCTP_packet tracks the congestion counters, but handles none of the congestion logic.

6 More Data Structures

Not everything is a state table or a smart pipe—after all, this is the kernel and we ARE programming in C. Here again, we have followed the RFC very closely. Most of the key concepts in the RFC manifest themselves as explicit data structures. For convenience, we refer to these data structures as "nouns".

Nearly all of the "noun" structures are designed for use with the sk_buff macros for list manipulation. These macros provide a doubly-linked list with locking.

6.1 struct SCTP_proto

The entire lksctp universe is grounded in an instance of struct SCTP_proto accessible through sctp_get_protocol(). This structure holds system-wide defaults for things like the maximum number of permitted retransmissions. It contains a list of all endpoints on the system.

6.2 struct SCTP_endpoint

Each UDP-style SCTP socket has an endpoint, represented as a struct SCTP_endpoint. Once we implement high-bandwidth sockets and TCP-style sockets, it will be possible for multiple sockets to share a single endpoint structure. The endpoint structure contains a local SCTP socket number and a list of local IP addresses. These two items define the endpoint uniquely. In addition to endpoint-wide default values and statistics, the endpoint maintains a list of associations.

6.3 struct SCTP_association

Each association structure, struct SCTP_association) is defined by a local

endpoint (a pointer to a `struct SCTP_endpoint`), and a remote endpoint (an SCTP port number and a list of transport addresses). This is one of the most complicated structures in the implementation as it includes a great deal of information mandated by the RFC. Among many other things, this structure holds the state of the state machine. The list of transport addresses for the remote endpoint is more elaborate than the simple list of IP addresses in the local endpoint data structure since SCTP needs to maintain congestion information about each of the remote transport addresses.

6.4 `struct SCTP_transport`

A `struct SCTP_transport` is defined by a remote SCTP port number and an IP address. The structure holds congestion and reachability information for the given address. This is also where we get the list of functions to call to manipulate the specific address family. For TCP you would find this information way up in the socket, but this is not possible for SCTP.

6.5 `struct SCTP_chunk`

Possibly the most fundamental data structure in lksctp is `struct SCTP_chunk`. This holds SCTP chunks both inbound and outbound. It is essentially an extension to `struct sk_buff`. It adds pointers to the various possible SCTP subheaders and a few flags needed specifically for SCTP. One strict convention is that `chunk->skb->data` is the demarcation line between headers in network byte order and headers in host byte order. All outbound chunks are ALWAYS in network byte order. The first function which needs a field from an inbound chunk converts that full header to host byte order *in situ*.

7 Acknowledgements

The authors are members of a team at Motorola dedicated to producing open source implementations in support of IETF standardisation. We would like to thank the people who make these efforts possible, specifically Maureen Govern, Stephen Spear, Qiaobing Xie, and Irfan Ali. We are of course deeply indebted to Randall Stewart and Qiaobing Xie for having created SCTP and for starting the Linux Kernel SCTP Implementation Project. We wish to recognizee the ongoing and significant contributions from developers outside Motorola, especially Jon Grimm and Daisy Chang of IBM, and Xingang Guo of Intel.

8 Availability

All the code discussed in this paper is available from the lksctp project on Source Forge:

`http://sourceforge.net/projects/lksctp/`

References

[RFC2960] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and, V. Paxson, *Stream Control Transmission Protocol*, RFC 2960 (Oct 2000).

[SCTPAPI] R. Stewart, Q. Xie, L. H. P. Yarroll, J. Wood, K. Poon, K. Fujita., *Sockets API Extensions for SCTP*, Work In Progress, `draft-ietf-tsvwg-sctpsocket-00.txt` (Jun 2001).

[SCTPIMPL] R. Stewart. *et al*, *SCTP Implementor's Guide*, Work In Progress, `draft-ietf-tsvwg-sctpimpguide-00.txt` (Jun 2001).

[SCTPMIB] J. Pastor, M. Belinchon. *Stream Control Transmission Protocol Management Information Base using SMIv2*, Work In Progress, `draft-ietf-sigtran-sctp-mib-03.txt` (Feb 2001).

[XP] K. Beck. *Extreme Programming Explained: Embrace Change*, Addison-Wesley Publishers (2000).

[SCTPORG] *Randall Stewart's SCTP site*, `http://www.sctp.org`, (2001).

[SCTPDE] *Tüxen/Jungmeier SCTP site*, `http://www.sctp.de`, (2001).