SCTP – performance and security

Course: 2G1305 Internetworking
Gerald Q. Maguire Jr
25/5 -05

Staffan Lundström
Daniel Hassellöf

Abstract

Stream Control Transmission Protocol (SCTP) is a new general-purpose IP transport protocol, standardized by the Internet Engineering Task Force (IETF). This report about SCTP is the result of a project in the course 2G1305 Internetworking given by IMIT/KTH. [1]

The aim is to give an overview of the protocol and to further investigate some essential concepts – in particular the four-way handshake, multiple streams and multihoming – as well as test a specific SCTP implementation.

Our tests show that multiple streams provide significant performance improvement compared to many independent associations (connections).

Introduction

We choose to deeper study the Stream Control Transmission Protocol (SCTP) for several reasons. To start with, none of us had previously heard about it. Secondly, it was said to particularly be designed to enhance time critical applications. Thirdly, we had read that it was intended to replace both TCP and UDP.

The aim of this project is not to present a complete description of the protocol. Instead, we intend to give an overview as well as to further investigate some particularly interesting parts of the protocol.

In order to do so, we describe the SCTP protocol format, before highlighting some essential differences to its predecessor TCP. The end section of this document is dedicated to practical tests of an SCTP implementation.

The describing text, the performed tests, the result analysis and the final test report have all been jointly carried out by the authors.

[1] http://www.imit.kth.se/courses/2G1305

Table of contents

Overview of protocol format

SCTP is message-oriented instead of byte-oriented. The difference with TCP is that in TCP all the data is treated as a stream of bytes whereas the data block boundaries are conserved in SCTP. An *SCTP packet* is the equivalent to a segment in TCP. Packets are made up of an *SCTP common header* and *chunks*, as in Figure 1. Chunks are divided into *control chunks* and *data chunks*. Data information is carried as data chunks and control information as control chunks.

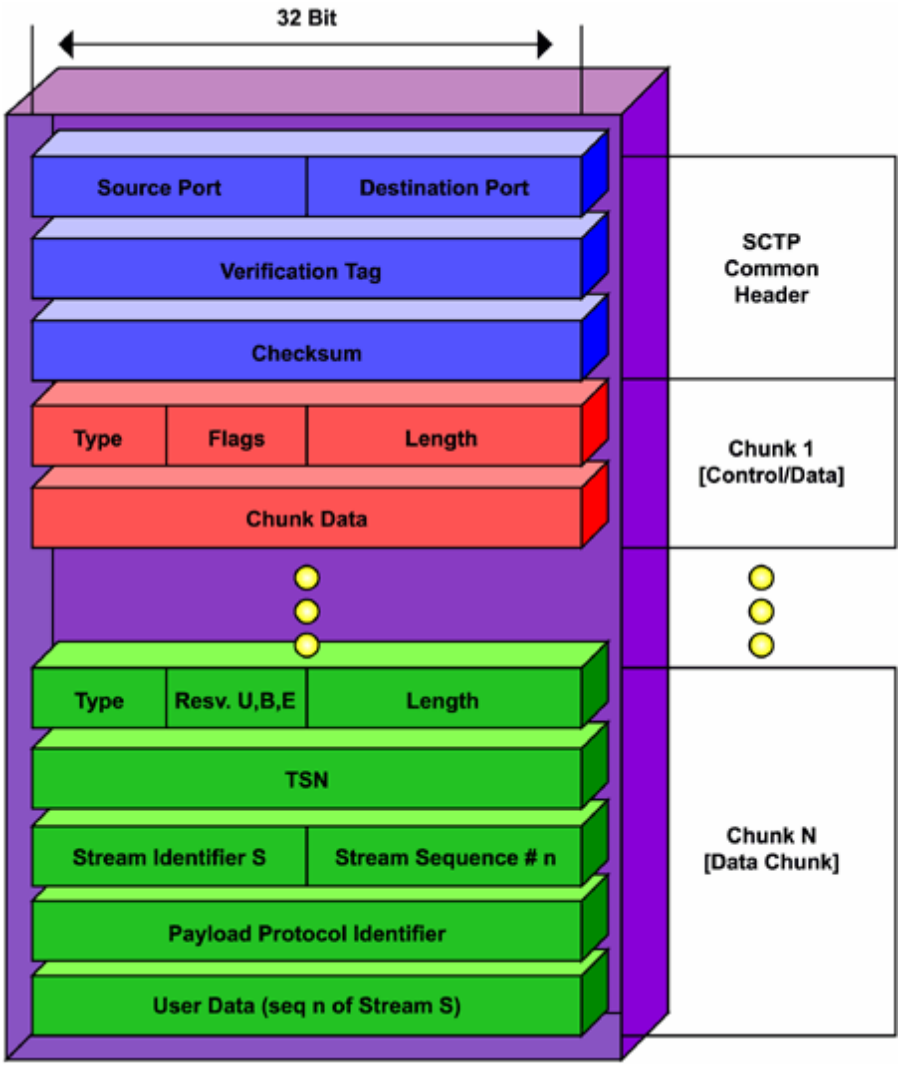


Figure 1[2] SCTP Packet format

The SCTP common header provides three things: Identification of the association that an SCTP packet belongs to and transport layer verification of data integrity. An interesting detail is that the association identification does not only use the source and destination port together with the IP addresses. The SCTP common header also includes a *verification tag* in order to

[2] http://www.iec.org/online/tutorials/sctp/topic04.html?Next.x=29&Next.y=5

distinguish between two different association instances and protect against a blind attacker that put data into an existing association. The TIME-WAIT state in TCP is therefore not necessary in SCTP. This should be compared with TCP where only the source and destination addresses together with the port numbers identify a connection. Moreover, the *Adler-32 checksum* provides better integrity check than the 16 bits used in TCP.

CHUNK	DEFINITION
Data DATA	The DATA chunk carries a user data payload
Initiation INIT	The INIT chunk is sent in order to initiate a SCTP association between two endpoints.
Initiation Acknowledgement INIT ACK	INIT ACK chunk acknowledges the receipt of an INIT chunk. The receipt of the INIT ACK chunk establishes an association.
Selective Acknowledgement SACK	SACK chunks acknowledge the receipt of DATA chunks.
Cookie Echo COOKIE ECHO	The COOKIE ECHO chunk is used exclusively during the initiation process and is sent to the peer endpoint.
Cookie Acknowledgement COOKIE ACK	The COOKIE ACK chunk acknowledges receipt of the COOKIE ECHO chunk. The COOKIE ACK chunk must take precedence over any DATA chunk or SACK chunk sent in the association. The COOKIE ACK chunk may be bundled with DATA chunks or SACK chunks
Heartbeat Request HEARTBEAT	HEARTBEAT chunks are sent from one SCTP endpoint to its peer in order to test the connectivity of a specific destination address in the association.
Heartbeat Acknowledgement HEARTBEAT ACK	Every time a HEARTBEAT chunk is received by an endpoint, a HEARTBEAT ACK chunk is sent to the source IP address in order to acknowledge receipt of the HEARTBEAT chunk.
Abort Association ABORT	The ABORT chunk is an indication to the peer endpoint to close the association. In addition, the ABORT chunk informs the receiver of the reason for aborting the association in the cause parameters.
Operation Error ERROR	The ERROR chunk is sent to the peer endpoint to report certain error conditions that may exist. The ERROR chunk may contain parameters that determine the type of error that has taken place.
Shutdown Association SHUTDOWN	The SHUTDOWN chunk triggers a graceful close of an association with a peer endpoint.
Shutdown Acknowledgement SHUTDOWN ACK	A SHUTDOWN ACK is used to acknowledge the receipt of the SHUTDOWN chunk at the end of the shutdown process.
Shutdown Complete SHUTDOWN COMPLETE	The SHUTDOWN COMPLETE concludes the shutdown procedure.

Figure 2[3] Chunk types

Each chunk must end or be padded to the next 32-bit word boundary. The format of a chunk can also be seen in Figure 1. The 8-bit Chunk type field represents the type of a chunk. Today 13 different types of 256 possible are defined; see
Figure 2 (12 control chunk types and 1 data chunk type). The Chunk flags (8 bits) are interpreted differently depending on the type of chunk. Any undefined chunk flags are set to zero. The Chunk length (16 bits) is calculated in bytes and includes these first four obligatory bytes in the count. The chunk length does not count padded bytes.

In TCP the control information is carried in the header. The SCTP module-based format is slimmer and more extensible. Packets can be mixed with control chunks and data chunks with the restriction that control chunks are always placed ahead of the data chunks. In TCP

[3] http://www.iec.org/online/tutorials/sctp/topic04.html?Next.x=29&Next.y=5

sequence numbers are used also in segments with only control information. In SCTP only the data chunks (the green chunk in Figure 1) are numbered by a *Transmission Sequence Number* (TSN), which corresponds to the sequence number in TCP. SCTP acknowledgement numbers are chunk-oriented and thus refer to the TSN. The control chunks, in the necessary cases, are acknowledged by another control chunk. For example, an INIT-ACK control chunk acknowledges the INIT control chunk. An intentional design choice is not to transfer control chunks reliably and only the data chunks are subject to a reliability mechanism (provided by control chunks).

Since there can be multiple streams in one association SCTP each stream is identified by a 16-bit *Stream Identifier* (SI). To distinguish between different data chunks belonging to the same stream, each data chunk in each stream is numbered with a *Stream Sequence Number* (SSN).

New and interesting features in SCTP

Associations in SCTP vs connections in TCP

An interesting difference between TCP and SCTP is the connection/association establishment. A well-known problem in TCP is the denial-of-service attack SYN flooding. A malicious attacker can flood a TCP server with SYN segments pretending it comes from different clients using forged IP addresses. In TCP after receiving a SYN segment the server responds with SYN + ACK and allocates state resources. Flooded with SYN segments the server will collapse or at least not be able to serve incoming connection requests.

As the observant reader already has noticed, a connection is in fact called an association in SCTP, even though the protocol is connection-oriented just like TCP. The initial handshake procedure with SCTP differs considerably from the procedure used by TCP. Instead of, like in the case of TCP, a three-way handshake with limited security, SCTP adds a step as well as the ability to further control the process.

Appendix A – SCTP state diagram shows the states in SCTP, including the state transitions for establishing an SCTP association. The client initiates an active open of an association by sending an INIT chunk to the peer endpoint. The client moves from the CLOSED state to the COOKIE_WAIT.

The server responds with an INIT_ACK packet containing two fields not present in the case of TCP – a verification tag and a cookie. The cookie contains the necessary state information, which the server would otherwise have to allocate resources for. The cookie also includes a signature so that it can be verified as authentic, and a timestamp to prevent replay attacks using old cookies. Note that the server therefore, at this point, neither moves into a new state nor allocates resources.

If the client has a forged IP address it will never receive the INIT_ACK chunk, and therefore it cannot send the third message, a COOKIE_ECHO chunk. The net result is that the conversation ends without any resources having been allocated at the server side.

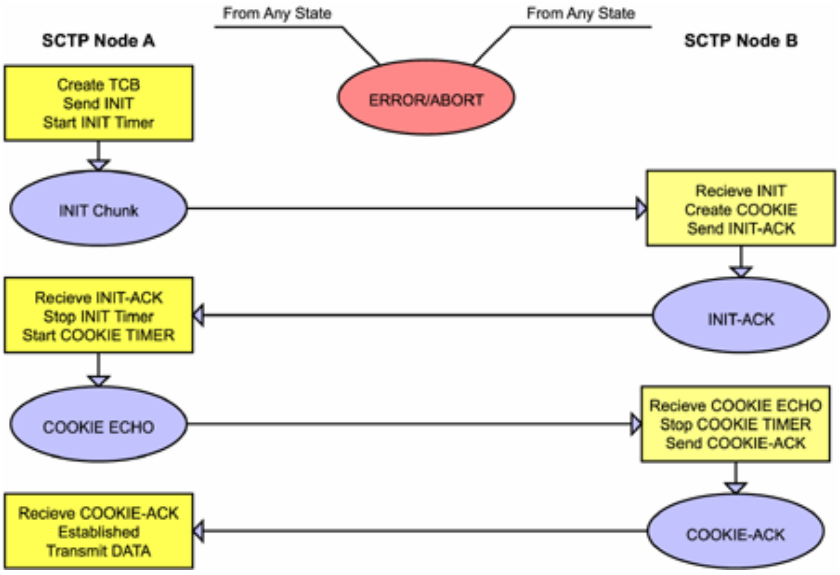


Figure 3[4] Association initiation

However, in the case of an authentic sender, the server will respond to the COOKIE_ECHO with a COOKIE_ACK chunk and move into the ESTABLISHED state.

Finally, as the client receives the COOKIE_ACK chunk, it will also enter the ESTABLISHED state, and the association establishment is completed. The whole procedure is illustrated in Figure 3.

A four-way handshake might seem to be less efficient than a three-way handshake. However, data chunks can also be exchanged in the third and fourth packet in SCTP, whereas in TCP the first possible data segment is in the fourth packet.

There are still potential vulnerabilities in SCTP. If the IP address is not forged or no encryption is used and an attacker can sniff the network, he can still get the cookie and reply with the third message (COOKIE-ECHO). In order to see how the SCTP.de implementation respond to INIT/COOKIE_ECHO flooding using valid cookies in the described way, we have set up a test. See *Test 2 – INIT/COOKIE_ECHO flooding*.

Multiple streams

According to [STEW] page 209 a web browser uses multiple connections for the simultaneous transfer of images and other multimedia objects. This way if a piece of an object is lost during transfer, the other objects will continue loading while the last piece is retransmitted.

One problem with this solution is that the bandwidth will be unfairly shared between different applications. Other applications, like a typical FTP-client, use only one connection for data transfer. A second problem is that congestion information is not passed to other connections. A third problem is that the server's connection queue can more easily overflow.

[4] http://www.iec.org/online/tutorials/sctp/topic05.html?Next.x=42&Next.y=12

A new feature introduced with SCTP to overcome these problems is the concept of multiple streams. Each association can have multiple streams within it. With one association per application the scheduling problem is solved. In and *Test 4 – Multiple streams performance during heavy traffic* we test the performance of this concept.

Multihoming

Administrators of hosts with high requirements on fault-tolerance often like to introduce redundancy in the systems. One important concept to achieve this is called multihoming. Multihoming is when a host can be addressed by more than one IP address, usually assigned from more than one ISP. Practically, this will normally mean that multiple network interface cards are used, with each assigned a different IP address. When one path fails another interface can be used without interruption. Different paths can also be used for load balancing.

In TCP a connection involves the source and one destination IP address. This means that even if the host is multihomed, only one of the available IP addresses can be used during a connection. In contrast to TCP, SCTP associations support multihoming; multiple IP addresses for each end are allowed in an association. An SCTP endpoint is a subset of the IP addresses on a machine, which forms the SCTP transport address.

Load balancing is not supported in today's implementations. For example, in the sctp.de implementation, that we used for testing, a primary must be chosen and the other paths will only be used if the primary fails.

SCTP tests

Test environment

The test environment consisted of two laptops connected on an Ethernet LAN via a switch. Both hosts ran the operating system Fedora Linux (kernel version 2.4.22).

Switch:
D-Link DI-804, 4 ports

Host 1 – server:
IBM Thinkpad T40
Pentium M, 1.3 GHz
768 MB DDR-RAM
Intel PRO/100 VE Network Connection

Host 2 – client:
Dell Inspiron 500m
Pentium M, 1.3 GHz
512 MB DDR-RAM
Intel(R) PRO/100 VE Network Connection

We chose an SCTP implementation developed at the Computer networking technology group of the University of Essen and the Münster University of Applied Sciences in cooperation with Siemens[5]. Current version of the SCTP library is 1.0.3. This implementation will hereinafter be referred to as the sctp.de implementation.

To analyze the network traffic used *Ethereal*[6]. To analyze the memory and CPU-usage we used the Linux *Top* command.

Test 1 – The four-way handshake

The objectives of the first test were to:

1. Get familiar with the SCTP API
2. To learn how to set up an SCTP association using a C-program
3. Be able to analyze SCTP packets using Ethereal

Description and methods

The first attempt was to install the sctp.de implementation in Windows. We soon realized that the suggesting way of doing this was ad-hoc; installing in Linux seemed more appropriate. After numerous hours spent on trying to install in Debian Linux, we made a radical decision and installed Fedora Linux on the laptops instead. This change turned out to be successful and we soon had the implementation of SCTP correctly set up for a first dump.

Supplied with the SCTP library was a set of client and server example programs. We used the program named *terminal* as the client together with *echo_server* program.

Result

Using the previously given programs we could easily set up an SCTP association. Here follows an excerpt of the network traffic, where we clearly see the four-way handshake that establishes the association.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.0.134	192.168.0.155	SCTP	INIT
2	0.000873	192.168.0.155	192.168.0.134	SCTP	INIT_ACK
3	0.020310	192.168.0.134	192.168.0.155	SCTP	COOKIE_ECHO
4	0.020839	192.168.0.155	192.168.0.134	SCTP	COOKIE_ACK

[5] http://www.sctp.de/sctp.html
[6] http://www.ethereal.com/

Conclusion and discussion

We could never have foreseen how problematic the work related to the prerequisites would be. However, when set up, the implementation worked flawlessly.

The SCTP primitives in the API being most essential for setting up an SCTP association were the following:

On the server side:

- **sctp_registerInstance()** is called to initialize one SCTP instance.
- **sctp_eventLoop()** is then called to listen for incoming events

On the client side:

- **sctp_associate()** is called to set up an association.
- **sctp_eventLoop()**

After familiarizing ourselves with and testing the SCTP API we found the functional level low enough to be powerful, but still easy to use. A more detailed reference of the API is given by [JUNG].

The sctp.de implementation lacks some documentation, and a few functions were not yet fully implemented. Though, the overall impression is positive.

Test 2 – INIT/COOKIE_ECHO flooding

The objectives of the second test:

1. Can the concept of a TCP SYN flooding be developed and applied on SCTP?
2. Has the sctp.de implementation any limit concerning the number of allowed simultaneous incoming association attempts from a given hosts?

Description and methods

As the objective indicates we will repeatedly initiate associations from our client host. However, we do not want to allocate any resources on the client side.

For this test we had two different approaches. The first included using the GNU Bash shell to iteratively execute a modified version of the *terminal* program. The change was to exit the program right after the COOKIE_ECHO chunk had been sent. This allows us to perform a repeated association establishment without allocating more resources than one association at a time on the client side.

The other approach was to add a loop to the *terminal* program, letting it repeatedly create new associations. We could not free the resources because the API function required a SHUTDOWN or ABORT chunk to be sent in order to move to the state where resource could be freed. Obviously this was not an option to us, as the server also would tear down the associations, and we intended to overflow it. We solved it by manually restarting the client program when needed.

Result

The two methods gave similar results. The difference was that the non-Bash method could send the packets in a much higher rate. After starting the flooding, we noticed the user CPU-time increase towards 95%. Possibly even more interesting, the memory usage increased approximately proportionally to time. However, the increase reached a ceiling at 102 MByte (13.6% of primary memory), after which the value stabilized. Here follows more details from *Top*.

```
[root@localhost local]# top

 12:01:41  up  1:09,  5 users,  load average: 1.88, 1.36, 0.70
79 processes: 76 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   90.8%    2.3%    6.7%   0.0%     0.0%    0.0%    0.0%
Mem:   773084k av,  692736k used,   80348k free,       0k shrd,   15824k buff
       131856k active,              482700k inactive
Swap:  377960k av,       0k used,  377960k free                  359512k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 4501 root      25   0  102M 102M   560 R    82.2 13.6   5:29   0 echo_server
```

We did not succeed in collapsing the server.

We could see no limitations in the sctp.de implementation concerning the number of allowed associations between two hosts, ie the number of associations allowed from a certain client IP address to the server.

We noticed, however not during each flooding attack, how the memory usage dropped after having stopped the client program.

Conclusion and discussion

Why did we not succeed in collapsing the server? The memory increase stopped due to dropped associations, which were indicated by the server debug output. This could unlikely be because of a limit in the number of allowed simultaneous associations, because new associations should rather have been disallowed than established associations being dropped.

We instead believe the memory increase stopped due to the server sending HEARTBEAT chunks. These monitor the reachability of the peer host when there is no user data being transferred. In our case the destination host failed to reply with HEARTBEAT_ACK and the server considered the association dead. According to [STEW] page 209 an idle association will be probed with the interval given by the parameter *HB.interval*. When the number of reattempts reaches *Assoc.Max.Retrans*, the server will mark the association inactive. The source code shows what the book indicates; resources can at this point be freed.

The only conclusion we can draw is that it could withstand this attack. It remains to be examined how well the SCTP implementation performs on a distributed attack.

Test 3 – Multiple streams

The objectives of the third test:

1. Measure the benefits in terms of startup time using multiple streams instead of multiple associations.
2. Measure the benefits in terms of memory needs of starting multiple streams instead of multiple associations.

This test is interesting because multiple streams is a feature not available in TCP. In TCP multiple connections must be set up to achieve the same effect (see section Multiple streams).

Description and methods

To measure the benefits of using multiple streams instead of multiple associations we set up a test with 10000 associations with 1 stream in each and compare it with 10000 streams over 1 association. The source code is included in Appendix B – Source code.

Results

	10000 associations with one stream each:	10000 streams over one association:
Memory usage:	8.3 %	0.1 %
CPU usage:	0 % idle	95.8 % idle
Run time:		
real	0 m 28.644 s	0 m 0.134 s
user	0 m 25.220 s	0 m 0.090 s
sys	0 m 1.930 s	0 m 0.020 s

Here follows more details from *Top*.

```
10000 associations:
68 processes: 62 sleeping, 6 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   95.3%    0.0%    4.6%   0.0%     0.0%    0.0%    0.0%
Mem:   773084k av,  750656k used,   22428k free,       0k shrd,   20148k buff
       192056k active,             495432k inactive
Swap:  377960k av,   52924k used,  325036k free                  300352k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 4081 root      16   0  265M 262M  6664 S     0.0 34.7   4:26   0 ethereal
26950 root      16   0 59752  58M   556 R    33.1  8.3   0:12   0 echo_server
```

```
10000 streams:
 22:55:23  up 10:00,  6 users,  load average: 0.94, 0.29, 0.09
69 processes: 68 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total    2.3%    0.0%    1.7%   0.0%     0.0%    0.0%   95.8%
Mem:   773084k av,  674060k used,   99024k free,       0k shrd,   20140k buff
       203596k active,             407536k inactive
Swap:  377960k av,   52928k used,  325032k free                  311600k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
26468 root      15   0   984  984   556 S     0.0  0.1   0:06   0 echo_server
26467 root      15   0 12080  11M  8548 S     0.0  1.5   0:03   0 ethereal
```

Conclusion and discussion

There is a dramatic difference in resource allocation. First of all, 10000 associations take 25 seconds to initiate whereas the number of streams is only a field proposed in the INIT and INIT-ACK chunks for the two channels respectively. In terms of memory the usage is almost unnoticeable for 10000 streams over 1 association. This can be compared with setting up 10000 associations, which requires a considerable amount of memory.

One can argue that as many as 10000 associations/streams is an unrealistic example. A comparison between 10000 streams in SCTP and 10000 connections in TCP would also be more interesting.

We conclude that multiple streams can very efficiently be initiated.

Test 4 – Multiple streams performance during heavy traffic

The objectives of the fourth test:

1. To measure the data transfer performance with multiple streams and compare with multiple associations with one stream.

Description and methods

To further examine the usage of multiple streams we extended the third test to include the transfer of data.

We developed two new programs, *terminal5* and *terminal6*. Terminal5 adds code for sending 10,000 chunks of data using one association and 10,000 streams in a round-robin fashion.

Terminal6 adds code for sending 10,000 chunks of data using 10,000 associations with one stream in each.

Both programs have timers measuring the time consumption of the data sending. The startup time is in other words excluded.

Result

	10k data chunks/ 10k associations/ 1 chunk/association *terminal6*	10k data chunks/ 1 association/ 10k streams *terminal5*	20k data chunks/ 1 association/ 10k streams *terminal5*
Memory usage:	65 000 KByte	988 Kbyte	988 Kbyte
CPU usage:	0 % idle	72.2 % idle	56.8 % idle
Run time:	9 s	4 s	9 s

Here follows more details from *Top*.

10k data chunks, 10k associations, chunk/association terminal6

```
68 processes: 65 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   94.0%    0.0%    5.9%   0.0%     0.0%    0.0%    0.0%
Mem:   773084k av,  471868k used,  301216k free,       0k shrd,    6204k buff
        89408k active,              342392k inactive
Swap:  377960k av,       0k used,  377960k free                  175660k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 6545 root      25   0 67308  65M   540 R    99.0  8.7   0:24   0 echo_server
 6126 root      16   0  1124 1124   916 R     0.9  0.1   0:03   0 top
```

10k data chunks, 1 association, 10k streams, terminal5

```
72 processes: 71 sleeping, 1 running, 0 zombie, 0 stopped
 01:37:34  up 12:42,  6 users,  load average: 0.16, 0.19, 0.32
72 processes: 71 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   23.7%    0.0%    3.9%   0.0%     0.0%    0.0%   72.2%
Mem:   773084k av,  474340k used,  298744k free,       0k shrd,   12712k buff
       378212k active,              45224k inactive
Swap:  377960k av,   53672k used,  324288k free                   96540k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
28085 root      16   0  1208 1208   912 R     0.9  0.1   0:00   0 top
28072 root      15   0   988  988   544 S    24.7  0.1   0:01   0 echo_serv
```

20k data chunks, 1 association, 10k streams, terminal5

```
23:27:49  up 24 min,  5 users,  load average: 0.50, 0.54, 0.47
69 processes: 66 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   37.2%    0.0%    5.8%   0.0%     0.0%    0.0%   56.8%
Mem:   773084k av,  410136k used,  362948k free,       0k shrd,    6204k buff
        94332k active,              275668k inactive
Swap:  377960k av,       0k used,  377960k free                  175716k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 6770 root      16   0   984  984   544 R    37.3  0.1   0:02   0 echo_server
 5575 root      15   0 54668  14M 11156 S     0.9  1.8   0:01   0 kdeinit
```

Conclusion and discussion

Using multiple streams during 9 seconds, we could send twice as many data chunks as we could using multiple associations. In addition, the memory resources needed for multiple streams were only a fraction of those needed for multiple associations.

Conclusion

SCTP introduces several new features compared to its predecessors TCP and UDP. The chunks provide a more modular and extensible packet format, which is advantageous in particular for real time applications. The four-way handshake using a cookie provides better resistance against flooding attacks in comparison with TCP. Multihoming provides better fault tolerance. Finally, our third and fourth tests prove that multiple streams provide significantly improved performance than multiple associations.

The used SCTP implementation lacks some documentation, but is mainly fully functional.

References

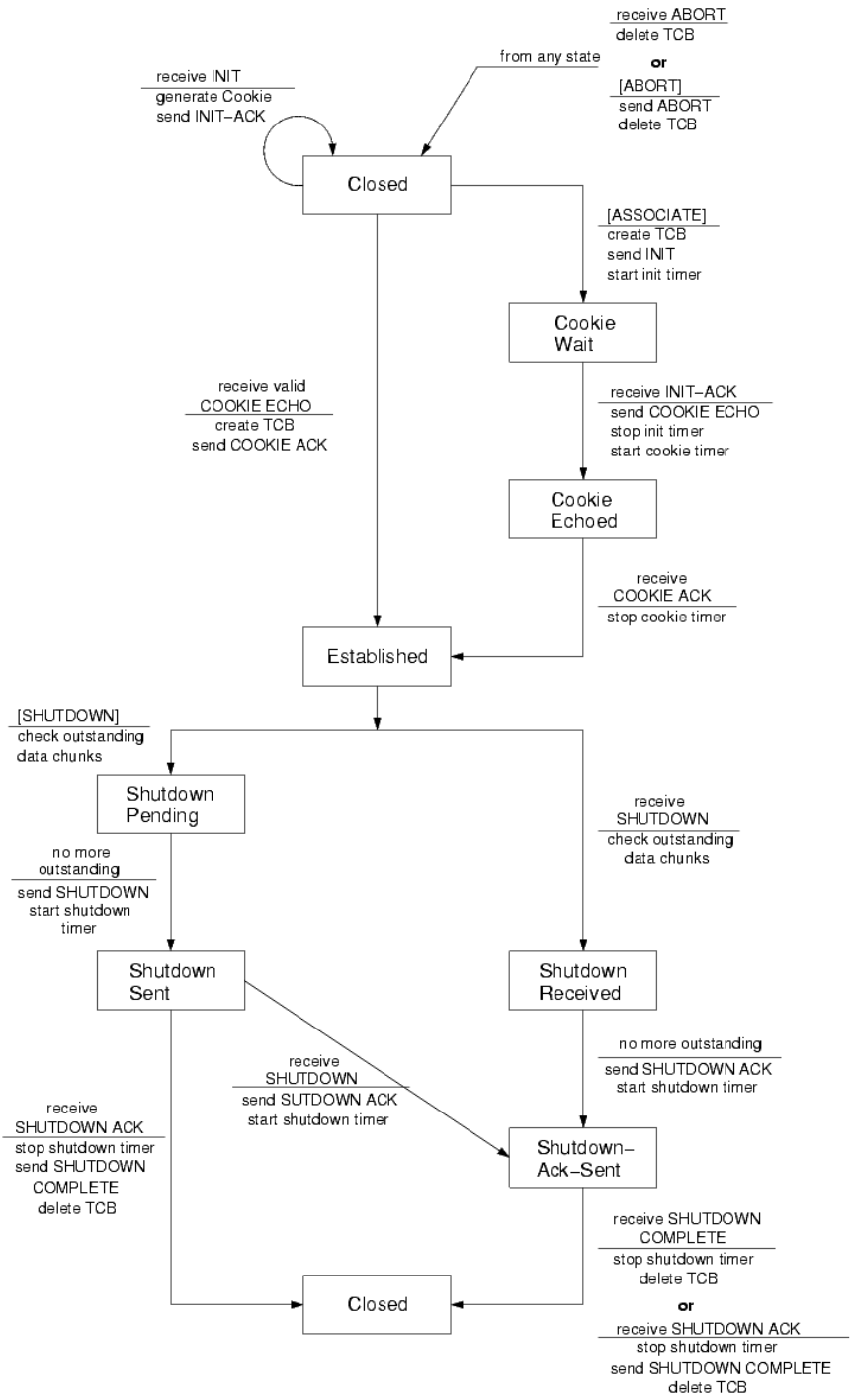
RFC2960 Stream Control Transmission Protocol, *Stewart et al*, RFC 2960, IETF, 2000

STEW Stream Control Transmission Protocol – A Reference Guide, *Randall R. Stewart, Qiaobing Xie,* Addison-Wesley, 2002

FOUR TCP/IP Protocol Suite, *Behrouz A. Forouzan*, McGraw Hill, 2006

JUNG Documentation of the SCTP-Implementation, *Andreas Jungmaier et al*, http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/sctp_fb/sctp-api.pdf, release: sctlib-1.0, 2001

Appendix A – SCTP state diagram



Created by Andreas Jungmaier
with XFig

Appendix B – Source code links

Test 1

Client - http://www.student.nada.kth.se/~u1djajou/SCTP/terminal.c
Server - http://www.student.nada.kth.se/~u1djajou/SCTP/echo_server.c

Test 2

Bash script - http://www.student.nada.kth.se/~u1djajou/SCTP/flood.script
Client - http://www.student.nada.kth.se/~u1djajou/SCTP/terminal2.c
Server - http://www.student.nada.kth.se/~u1djajou/SCTP/echo_server.c

Test 3 & 4

Client (multiple streams) - http://www.student.nada.kth.se/~u1djajou/SCTP/terminal5.c
Client (multiple assoc) - http://www.student.nada.kth.se/~u1djajou/SCTP/terminal6.c
Server - http://www.student.nada.kth.se/~u1djajou/SCTP/echo_server.c