The Impact of SCTP on Server Scalability and Performance

Kumiko Ono and Henning Schulzrinne
Dept. of Computer Science, Columbia University.
Email: {kumiko, hgs}@cs.columbia.edu

Abstract

The Stream Control Transmission Protocol (SCTP) is a relatively recent transport protocol, offering features beyond TCP. Although SCTP is an alternative transport protocol for the Session Initiation Protocol (SIP), we do not know how SCTP features influence SIP server scalability and performance. To estimate this, we measured the scalability and performance of two servers, an echo server and a simplified SIP server on Linux, for both SCTP and TCP. Our measurements found that using SCTP does not significantly affect data transfer latency. However, the number of sustainable associations drops to 17-21% or to 43% of the TCP value if we adjust the acceptable gap size of unordered data delivery.

1 Introduction

The Stream Control Transmission Protocol (SCTP) [1] was originally designed for carrying telephony signaling protocol, Signaling Systems No.7 (SS7), over IP. Similar to TCP, SCTP is reliable and connection-oriented, but it is message-oriented like UDP, and has additional features such as multi-streaming and multi-homing. SCTP is defined as an alternative transport protocol to UDP or TCP for the Session Initiation Protocol (SIP) [2] [3], which is a major Internet telephony signaling protocol. Understanding how SCTP affects scalability and performance is important for building SIP servers and designing large voice over IP (VoIP) systems.

Depending on its role in a VoIP network, a SIP proxy server may connect to a large number of user agents or to a, typically smaller, number of other proxy servers, as shown in Figure 1. Proxy servers maintaining connections to user agents are often called edge proxy servers Therefore, if a connection-oriented transport protocol is used, the server is required to manage a large number of concurrent connections, making server scalability as important as request throughput and latency. On the other hand, between proxy servers, the server needs to manage a smaller number of connections, since connections can be shared among user agents with the same signaling destination.

Even though SCTP is not as commonly used as TCP or UDP, it has been implemented as a kernel module in Linux, so that it can be easily used between user agents and a proxy server as well as between proxy servers. This broad deployment requires a server with enough scalability to accommodate a large number of user agents. We present how SCTP affects scalability and the performance of an echo server and a simplified SIP server that only implements the message handling parts of a SIP proxy server. From our measurements, we estimate the effect of choosing SCTP as a transport protocol for SIP.

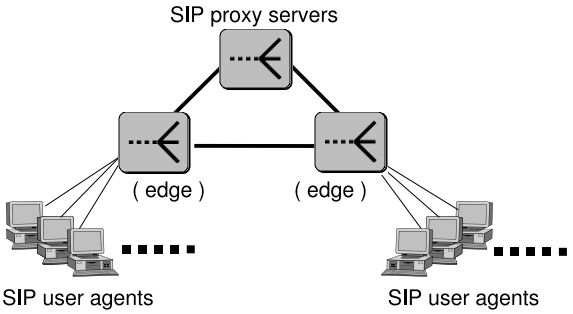

Figure 1: Two types of connections at SIP proxy servers

The remainder of this paper is organized as follows. Section 2 discusses which SCTP features benefit SIP. Section 3 describes our measurement objectives and environment. Section 4 compares socket memory usage between SCTP and TCP and suggests how to save memory with SCTP. Section 5 compares the number of sustainable connections and Section 6 compares data transfer latency for an echo server between SCTP and TCP. Section 7 compares data transfer latency for a simplified SIP server, namely, SIP front-end server among SCTP, TCP and UDP. We conclude with a discussion of the influence of SCTP on SIP server scalability and performance in Section 8. Appendices describe the metrics, tools and configurations for our measurements and the data structures used by the Linux SCTP implementation.

2 SCTP Features

Table 1 compares the features of SCTP, TCP and UDP. The following SCTP features potentially benefit SIP applications and server scalability, respectively.

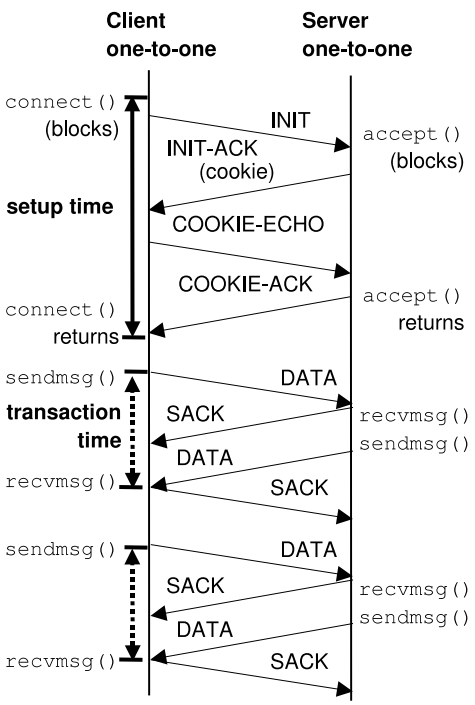

Figure 2: Message exchanges using one-to-one style sockets for both a server and a client

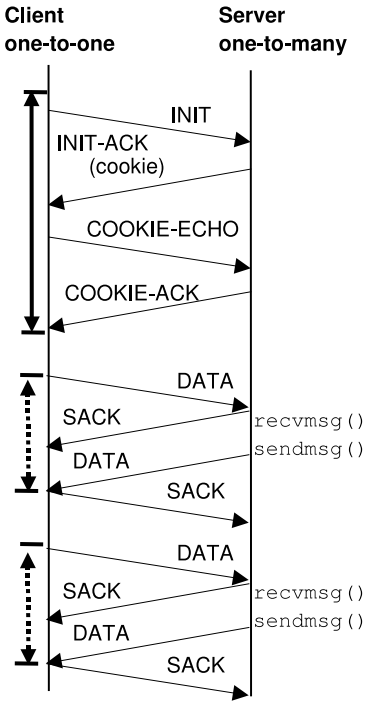

Figure 3: Message exchanges using a one-to-many style socket for a server

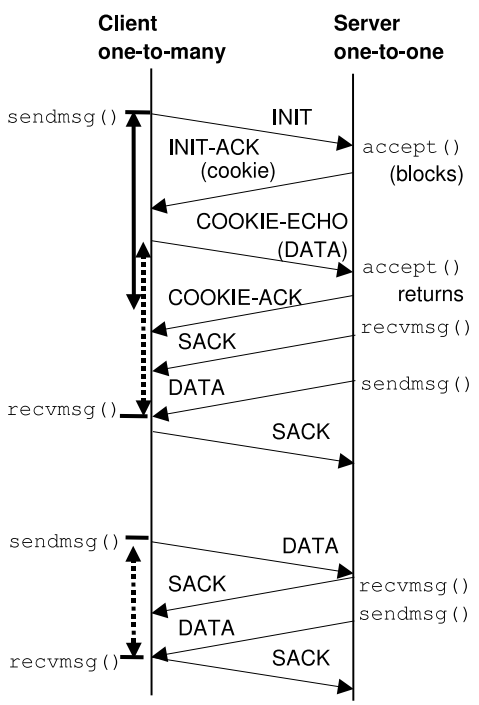

Figure 4: Message exchanges using a one-to-many style socket for a client

2.1 Beneficial Features for SIP Applications

Piggyback setup in four-way handshake: SCTP establishes a connection with a four-way handshake using INIT, INIT-ACK, COOKIE-ECHO, and COOKIE-ACK messages, as shown in Figure 2. Clearly, this four-way handshake requires two round trip times (RTTs), which is one more than that for the three-way handshake in TCP. However, the piggyback setup allows to bundle user data into the COOKIE-ECHO message, as shown in Figure 4, so that it reduces the RTTs combined in the handshake and in sending user data to the same as those of TCP. In a SIP session using non-persistent connection, where a new connection is established, a larger RTTs in the handshake causes a longer setup delay. Thus, the piggyback setup is expected to mitigate this setup delay.

Table 1: Comparison of transport protocols

	SCTP	TCP	UDP
Connection-oriented	Yes: establish using four-way handshake including cookie to resist flood attack	Yes: establish using three-way handshake	No
Socket style corresponding to connections	one-to-one or one-to-many	one-to-one	(one-to-many)
Message-oriented	Yes	No	Yes
Message exceeding MTU	Yes	Yes	No
Reliability	Yes	Yes	No
Flow control	Yes	Yes	No
Congestion control	Yes	Yes	No
Multi-streaming	Yes: minimize HOL blocking	No	No
Multi-homing	Yes	No	No

Message orientation: Similar to UDP, SCTP preserves message boundaries. Applications can extract a single received message and determine if the original message is fully delivered through the socket API, while they need to parse received messages over TCP using the Content-Length header in SIP. However, message parsing is necessary for SIP applications. Thus, we suspect that this message orientation has a negligible benefit.

Message exceeding MTU size: Similar to TCP, SCTP supports the delivery of message exceeding Maximum Transfer Unit (MTU) size by segmentation. Since some networks or services require SIP extension headers or signatures, the message size of a SIP request may grow beyond the Ethernet MTU of 1,500 bytes. For example, an INVITE request in IP Multimedia System (IMS), which contains privacy headers and many routing-related headers, is at least 1,550 byte long.

2.2 Features improving Server Scalability

Two features of SCTP, one-to-many style sockets and multi-streaming, potentially help server scale by reducing memory usage and increasing throughput.

SCTP provides one-to-one and one-to-many socket interfaces. These two interfaces differ in representing *associations*, which mean connections in SCTP, as shown in Figures 5 and 6. While a one-to-one style socket can represent a single association, a one-to-many style socket can represent multiple associations, similar to UDP, where a socket can receive messages from multiple clients. Other SCTP-related data structures are described in Appendix C.2.

A server using a one-to-many style socket can receive messages from different associations at a single listening socket, i.e., without creating a new socket by calling the `accept()` system call to create a new association, as shown in Figure 3. Thus, using the one-to-many style socket can drastically reduce the number of sockets at a server.

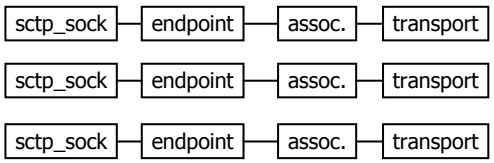

Figure 5: One-to-one style socket data for three associations

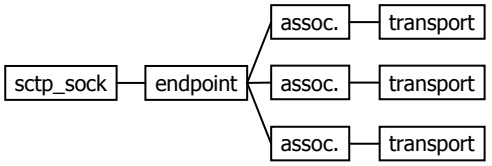

Figure 6: One-to-many style socket data for three associations

Additionally, a client using a one-to-many style socket can utilize piggyback setup to reduce the setup delay of the four-way handshake described in Section 2.1. To use piggyback setup, a client has to create a one-to-many style socket and invoke the `sendmsg()` system call to send a message without calling the `connect()` system call, as shown in Figure 4.

Thus, we can expect to benefit from one-to-many style sockets for both server and client, although the socket style can be set independently to each other. At the same time, however, using one-to-many style sockets potentially decreases server throughput. By sharing a single socket buffer, the server receives and sends messages for all associations and de-multiplexes the messages by four tuples: source and destination IP addresses and ports. This is similar to that of UDP, but messages sent are kept longer than for UDP, since they cannot be removed until the SCTP ACK has been received. Therefore, the send buffer may be exhausted at high request rates. We will evaluate the effects of this one-to-many style socket by comparing three sequences shown in Figure 2, 3 and 4.

The other feature, multi-streaming, can minimize head-of-line (HOL) blocking, as evaluated by Camarillo, Kantola and Schulzrinne [4]. The HOL blocking occurs in TCP when a segment is lost and a subsequent segment arrives out of order. The receiving application needs to wait for the lost segment to read the arrived segment. In SCTP, however, by breaking multi-session streams into separate streams, the HOL blocking can be minimized, even though it occurs in the same stream. This multi-streaming feature is effective especially for inter-proxy servers, where multiple SIP sessions can share an association. We can expect this multi-streaming feature to improve throughput especially in a congested network, but do not discuss this feature here.

3 Measurements

We measured the effects of three SCTP features, namely, one-to-many style sockets, piggyback setup and message orientation. To evaluate the effect of the one-to-many style sockets, we first identified the data structures of the two styles of SCTP sockets, then we compared memory usage among these two styles of SCTP sockets and TCP sockets. Then, to evaluate its effect on server scalability, we compared the number of sustainable associations for an echo server between two SCTP socket styles. To evaluate its effect on server performance, we compared the setup time and transaction time. By comparing the setup time, we identified the effect of the piggyback setup. By comparing the transaction time, we identified the effect of the SCTP message orientation using a SIP front-end server, which simply receives a SIP request and responds with a 200 OK response without any substantial SIP operation.

3.1 Measurement Environment

We use two servers, an echo server and a SIP front-end server using a single process and single thread. Table 2 compares them with a SIP server, which usually use multi process and/or multi threads. The servers under test (SUT) for both run on a dedicated host with Pentium IV 3 GHz 32-bit dual-core CPU and 4 GB of memory. The SUT runs Linux 2.6.23 configured with the default virtual memory split of 1G/3G, where the kernel space is 1 GB and the user space is 3 GB. When the server needs to wait for events on more than 1,024 sockets, it uses the `epoll()` system call.

For the echo clients or SIP user agents, we use six hosts with Pentium IV 3 GHz 32-bit CPUs and 1 GB of mem-

Table 2: Comparison of servers

	Echo server	SIP front-end server	SIP server
Receive/Send msg.	Yes	Yes	Yes
Forward msg.	No	No	No for registrar, yes for proxy
Msg. parsing	No	Yes	Yes
SIP operation	No	No	Yes
Database access	No	No	Yes
Number of msg. per request	2	2	2 for registrar, 14 for proxy
Number of transactions	1	1	1 for registrar, 6 for proxy
Process/Thread	Single/ Single	Single/ Single	(Multi/ Multi)

ory running Redhat Linux 2.6.9, except the measurement applying a patch in Section 6.1, where we use two hosts which have the same hardware and software specification with the echo server. These hosts communicate over a 100 Mb/s Ethernet connection at light load. The round trip time measured by the `ping` command is roughly 0.1 ms. Appendix A details measurement tools and configurations.

4 SCTP Data Structures

We first identified the data structures for establishing and maintaining an SCTP socket and calculated the memory usage. Then, we evaluated server scalability by measuring the number of sustainable associations at the SUT.

4.1 Comparison of the Data Size of SCTP and TCP Sockets

Table 3 itemizes the sizes of socket-related data structures and those allocated from cache objects called slab cache in Linux. The slab cache is implemented for frequent allocations and deallocations of data. The socket-related data structures detail used in both two transport protocols, which are listed in Appendix C.1, and for SCTP-specific in Appendix C.2. SCTP requires larger protocol specific data including the 5120-byte *sctp_association* which stores data linked to an association, while TCP requires only the 1096-byte *tcp_socket*. Table 3 also shows that the amount of memory using one-to-many style sockets significantly increases as a function of the number of associations. The amount of memory using multi-streams increases slightly as a function of the number of streams.

Furthermore, each socket-related data object consumes more than the data size when allocated from slab cache objects, since the slab cache object size is a power of two. For example, the dominant object, *sctp_association* is allocated from a 8,192 byte slab object. As a result, maintaining an SCTP socket statically consumes 10,812 bytes, approximately five times of the amount needed for a TCP socket, even in the simplest case, that is, with a one-to-one style socket and a single stream each. The dominant data is association-related data, which consumes approximately 80% of the total memory usage. Even when using one-to-many style sockets, a server needs to allocate the same number of associations with when using one-to-one style sockets. Therefore, we cannot expect drastic memory saving by using one-to-many style sockets, although we expected that in Section 2.2. Section 4.2 will investigate how to reduce association-related data in order to increase the effect of one-to-many style sockets on memory saving.

4.2 Memory-conscious Usage of Association Data

To cut down the memory footprint of association-related data, it is crucial to reduce the size of the dominant *sctp_association* data. Although the size of this data structure is 5,120 bytes, it grows to 8,192 bytes when it is allocated from the slab cache objects. This is because the smallest slab cache object to store more than 4,096 bytes is 8,192 bytes. This means that the memory footprint will

Table 3: Itemized memory usage for an SCTP and a TCP socket at a server

Data Structure		Slab		
Name	Size (B)	Size (B)	# of objects SCTP	# of objects TCP
dentry	128	128	1	1
file	136	192	1	1
inode	328	384	1	1
socket	40			
sock(tcp_socket)	1,096	1,152	0	1
sock(sctp_socket)	772	896	1	0
eppoll_entry	36	36	1	1
epitem	80	128	1	1
sctp_endpoint	176	256	1	0
sctp_bind_addr	40	64	1	0
Subtotal for a socket (bytes)			2,044	2,020
sctp_association	5,120	8,192	1 or n[a]	0
sctp_transport	284	512	1 or n[a]	0
sctp_ssnmap	60	64	1 or m[b]	0
Subtotal for an association (bytes)			8,768	0
Total memory usage (bytes)			10,812	2,020

[a] one-to-many
[b] multi-streaming

be halved if we can reduce 5,120 bytes to 4,096 bytes.

Appendix C.2 lists the members of the *sctp_association* structure. The total size is 5,120 bytes and the dominant sub-member is *map[sctp_tsnmap_storage_size(SCTP_TSN_MAP_SIZE)]*, which accounts for 4,096 bytes. This is a byte mapping array, namely, TSN (Transmission Sequence Number) map, each byte of which indicates the number of chunks for each TSNs to trace received TSNs for unordered data delivery. The TSN map size, i.e., the SCTP_TSN_MAP_SIZE, is set to allow a gap of 2,048 segments between the cumulative ACK and the highest TSN by default. This mapping array is twice the size of the TSN map to allow an overflow map. Therefore, if we do not need to handle such large gap, we can reduce the TSN map size.

Figure 7 shows how memory usage for a socket can decrease by adjusting the TSN map size. When we adjust it to 512, the mapping array would decrease to 1,024 bytes, and the *sctp_association* would decrease to 2,048 bytes. This reduced data could then be allocated from a 2,048 byte slab object. As a result, the total amount of memory for an SCTP socket could be reduced to less than half, 4,668 bytes, although the amount is still more than twice the size of a TCP socket. Adjusting the TSN map size to 256 does not affect the data size of the *sctp_association*, since the size of mapping array is still larger than 1,024 bytes. Therefore, if we need to support unordered data delivery, adjusting the TSN map size to 512 is most effective to reduce the size of the *sctp_association*. If we do not need to support any unordered data delivery, which is unrealistic though, we could adjust the TSN map size to zero. The data size of the *sctp_association* would then decrease to 1,024 bytes. As a result, the total amount of memory could be reduced to 3,664 bytes which is still around twice the size of a TCP socket.

In a SIP session between a user agent and the server, SIP messages do not have to allow such large gap, since a SIP request requires the SIP response before sending a new SIP request. However, in a SIP session between proxy servers, a number of aggregated SIP messages are transmitted over a single association. For this situation, the SIP servers need to allow a certain gap depending on the traffic model between proxy servers.

5 Number of Sustainable Associations

As a metric of server scalability, we measured the number of sustainable associations for the echo server. After the echo clients request new associations, they send a SIP INVITE message and receive a copy and maintain the associations. This measurement was performed without the tuning proposed in Section 4.2.

To allow a large number of concurrent associations, we raised an upper limit of the number of file descriptors per process to 60,000 for the clients and to 1,000,000 for the SUT. In addition, to increase concurrent associations for each client, we expanded the range of ephemeral local port to 10,000 - 65,535. Specifically for SCTP, for simplification, we disabled the heartbeat mechanism, which probes the reachability of remote associations every 30 seconds by default. Appendix D describes the detailed system configuration.

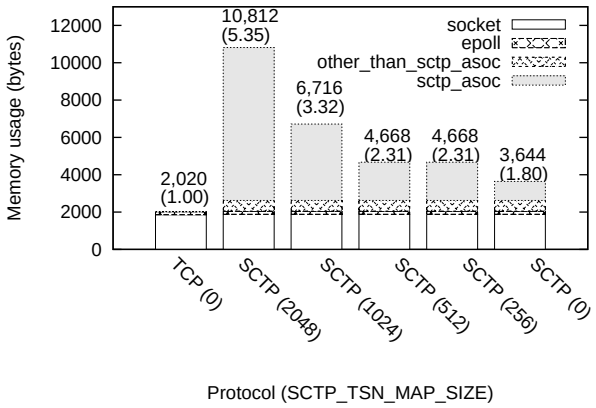

Figure 7: Memory usage for a socket as a function of SCTP_TSN_MAP_SIZE

5.1 The Effect of One-to-many Style Sockets

Table 4 compares the numbers of sustainable associations or connections among the one-to-one and one-to-many style sockets of SCTP and TCP. The number is measured just before the system yields an error, out of memory for sockets, and the "oom killer" process kills heavy processes based on the memory usage and lifetime. From our measurements, the number of sustainable associations of SCTP is only 17-21% that of TCP. Although a server using a one-to-many style socket can increase the number of concurrent associations by about 15,000 compared to one-to-one style sockets, this improvement makes barely a difference in the comparison with TCP.

Table 4 also compares the memory usage per association or connection by measuring the memory usage for the slab cache objects. The memory usage agrees with our analysis in Section 4.1. Thus, if we adjust the TSN map size depending on the requirement for handling unordered data delivery, we could improve the number of sustainable associations up to approximately 50% of that of TCP.

6 Data Transfer Latency for an Echo Server

As a metric of server performance, we measured data transfer latency for the echo server to compare the setup time and transaction time of SCTP and TCP to identify how using a single socket buffer in a one-to-many style

Table 4: Sustainable associations and memory usage per association for SCTP and TCP

	SCTP		TCP
Socket Style at server	One-to-one	One-to-many	
Number of assoc.	74,680	90,607	419,019
Ratio	0.17	0.21	1.00
Memory usage per assoc. (KB)	11.12	8.90	2.05

socket affects and how much piggybacking data in the initial handshake reduces them. For this measurement, we configured the echo server and clients to eliminate unnecessary delay and errors, in addition to allowing a large number of connections as shown in Section 5. Appendix D describes the details.

The setup time of an association is the elapsed time from the instant that a client invokes the `connect()` system call to returning from it. The transaction time is the elapsed time from the instant that a client invokes the `sendmsg()` system call to send a 1,550 byte INVITE request to its invoking the `recvmsg()` system call to receive a copy. The echo clients send the requests at 2,500 requests/second and the echo server accumulates the SCTP associations or TCP connections until their number reaches 50,000.

6.1 The Effect of One-to-many Style Sockets

To identify the effect of one-to-many style sockets for the echo server, we compare the setup and transaction times between the two SCTP sockets whose message exchanges are shown in Figures 2 and 3. Table 5 compares the setup and transaction times of the two SCTP socket styles and TCP for the echo server. Significantly, the setup time for the one-to-many style sockets grows linearly with the number of associations, while the setup time as a function of the number of associations for the one-to-one style sockets remains constant. Similarly, the transaction time for one-to-many style sockets significantly differs from the one-to-many style sockets, while the transaction time for one-to-one style sockets does not significantly differ from that using TCP.

To investigate the reason of the linear increase with the number of associations using a one-to-many style socket, we traced the kernel source code, and found that when receiving INIT and COOKIE_ECHO messages, a linear search is used to look up a matching association by endpoint. Such search always fails. Therefore, the one-to-many style socket, where an endpoint links to multiple associations, increases the setup time as a function of the number of associations, while one-to-one style sockets, where each endpoint links to a single association, do not increase the setup time. Also, when sending a message, the `sctp_sendmsg()` function in kernel calls a lookup function which performs a linear search. Thus, the transaction time increases linearly. However, the increase is not so drastic as that in the setup time. Unlike to the setup time, this association search is always successfully performed and takes time depending where the matching association is stored in the list of associations. Since the linear search clearly causes the cost of using a one-to-many style socket, both setup and transaction times could be improved by replacing it with a hash table lookup.

After applying a patch to replace the search algorithm, we re-measured the setup time. Figure 8 indicates that the setup time using a one-to-many style socket remains constant at 0.34 ms, similar to that using one-to-one style sockets.

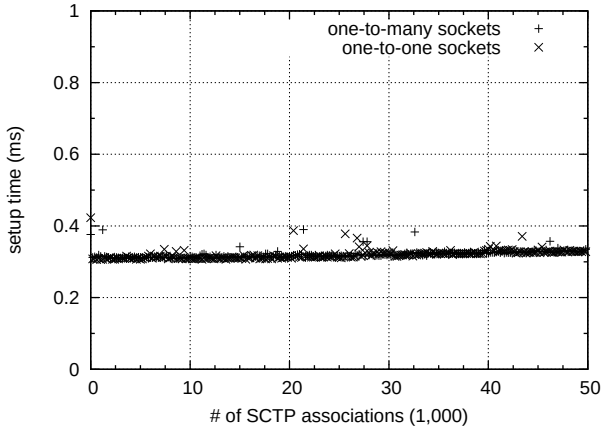

Figure 8: SCTP setup time as a function of the number of associations with one-to-one and one-to-many style sockets after replacing search algorithm

6.2 The Effect of Piggyback Setup

With the piggyback setup mechanism, we can expect to reduce the total number of messages and the combined time for the setup and the transaction, not the setup time itself. To identify the effect of using piggyback setup, we compare the combined time between the two styles of

Table 5: Setup and transaction times for SCTP and TCP

	SCTP			TCP
Socket style at server	One-to-one		One-to-many	
Setup style	Regular	Piggyback		
Setup (ms)	0.34	0.84	0.38-170.91	0.17
Transaction (ms)	0.54		0.65-34.14	0.48
Total (ms)	0.88	0.84	1.03-205.05	0.65

Table 6: Elapsed time between messages for SCTP and TCP

SCTP			TCP	
Messages	Regular setup (ms)	Piggyback setup (ms)	(ms)	Messages
s:INIT	0.00	0.00	0.00	s:SYN
r:INIT-ACK	0.14	0.14	0.09	r:SYN, ACK
s:COOKIE-ECHO	0.01	0.01	0.01	s:ACK
r:COOKIE-ACK	0.13	0.23		
s:DATA	0.00	N/A[a]	0.00	s:DATA
r:DATA	0.37	0.26[b]	0.34	r:DATA

[a] DATA is piggybacked with COOKIE-ECHO.
[b] This indicates the elapsed time from receiving COOKIE-ACK.

SCTP sockets for the clients, whose message exchanges are shown in Figures 2 and 4. Table 5 compares the setup, the transaction and the combined times among two SCTP setup styles and TCP. This table indicates that using piggyback setup reduces the combined time only by 0.04 ms in our local area network.

To investigate the cost of the piggyback setup, we monitored the elapsed time for each RTT at an echo client using the `tcpdump` program. Table 6 shows that the elapsed time between sending `COOKIE-ECHO` and receiving `COOKIE-ACK` grows by 0.1 ms beyond that of the SCTP regular setup. Therefore, in spite of reducing the elapsed time between receiving `COOKIE-ACK` and receiving a copied user message by 0.11 ms, the overall effect of using piggyback setup is slight: 0.01 ms at the network layer and 0.04 ms at the application layer. Table 6 also indicates that the setup with processing signed cookies is originally expensive. As long as SCTP processes signed cookies to protect against a INIT flooding attack, the SCTP setup is more expensive than that for TCP, which processes no cookies by default. Although the effect of using piggyback setup is slight in our measurement environment, the effect of reducing one RTT would be larger in a wide area network.

7 Data Transfer Latency for a SIP Front-end Server

To identify the effect of the message orientation, we used a SIP front-end server that parses SIP messages to determine the message boundary for TCP. This SIP front-end server is halfway of an echo server and a SIP registrar server as shown in Table 2.

As described in Section 2.1, the message orientation over SCTP potentially affects processing time for a SIP server, but we suspected that it is small. Thus, to identify such small effect of the message orientation, we eliminated the processing of SIP operation including database access, which dominates processing time in the SIP registrar server.

7.1 The Effect of Message Orientation

When calling the `recvmsg()` system call, we can determine whether or not the received message is fully delivered by checking the message flag. If the full message is delivered, the message flag is set to `MSG_EOR`. Figure 9 compares the setup and transaction times among the three SCTP cases, TCP and UDP, for the SIP front-end server. As we suspected, we cannot see any effect of the message orientation. We summarized that parsing a SIP message is not a heavy task, since the typical message size is approximately 1,550 bytes. If a SIP message conveys a bulk data as much as for a file transfer application, the message orientation could be effective. Also, message parsing is required for SIP operations regardless of the transport protocol. Therefore, the message orientation feature has little effect on a SIP server.

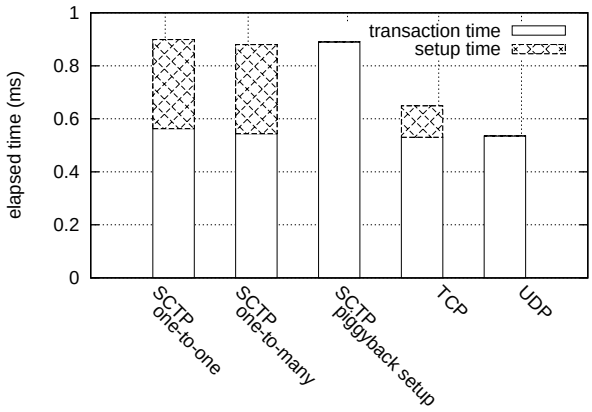

Figure 9: Setup and transaction times for SCTP and TCP for SIP front-end server

8 Conclusion

We have shown how using SCTP impacts on server scalability and performance by evaluating the effect of the three SCTP features and to estimate the impact on SIP server scalability and performance. We suspected that using SCTP has negative impact on server scalability, since an SCTP association, which corresponds to a TCP connection, requires significantly more storage to support the additional features. Then, we expected using a one-to-many style socket enables to increase the number of sustainable associations by saving the number of sockets. Our measurement results indicates that the echo server scalability decreases to one fifth of that using TCP, because of a large

amount of association-related data. By adjusting the capacity for accepting out of order data delivery, we could mitigate the decrease to a half. Yet, the scalability gap between SCTP and TCP remains significant. This gap surely limits the usability of SCTP for edge servers.

On the other hand, data transfer latency, such as the setup and transaction times, does not significantly differ between using one-to-one style sockets for SCTP and using TCP. When a client uses the piggyback setup, the difference in the combined time can slightly decrease. We failed to identify the effect of the message orientation using the SIP front-end server. For one-to-many style sockets of SCTP, we had to fix the current lookup function in order to limit the latency for a scalable server. Since the SCTP kernel implementation is far less mature than the TCP implementation, we suspect that there is a significant room for improvement.

Acknowledgement

This work was supported by NTT Corporation. The authors would like to thank Vlad Yasevich of HP for providing a patch for the association lookup function.

References

[1] R. Stewart. Stream Control Transmission Protocol. RFC 4960, IETF, September 2007.

[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, IETF, June 2002.

[3] J. Rosenberg, H. Schulzrinne, and G. Camarillo. The Stream Control Transmission Procotol (SCTP) as a Transport for the Session Initiation Protocol (SIP). RFC 4168, IETF, October 2005.

[4] G. Camarillo, R. Kantola, and H. Schulzrinne. Evaluation of Transport Protocols for the Session Initiation Protocol. In *IEEE Network*, September 2003.

A Measurement Metrics and Tools

Figure 10 shows that the controller starts the echo server and monitoring tools at the SUT and echo client programs at multiple echo clients. After starting all processes at the SUT and the clients, the controller waits for the responses from the clients so that it can synchronize the time the clients start to send messages to the SUT.

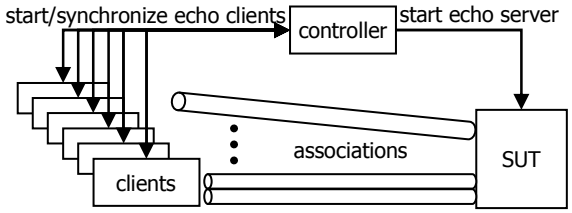

Figure 10: Measurement environment

To monitor memory usage of sockets and the number of slab objects allocated for socket-related data, we monitored the `/proc/slabinfo` file every second. Also, we monitored the overall memory usage of slab objects using the `/proc/meminfo` file.

To monitor CPU time, we monitored the total CPU time and that of the echo server process using the `top` command. Also, we used a profiler, the `oprofile` tool, to identify heavy functions including the kernel functions. By this, we identified `__sctp_endpoint_lookup_assoc()` as the heaviest function call.

To monitor the number of concurrent sockets, we scanned the `/proc/net/sctp/sctp_dbg_objcnt` file every second. To enable this monitor, we changed the kernel configuration parameter and recompiled the kernel.

To monitor the setup and the transaction times, we added time stamps at the echo client program. To avoid the slowdown caused by calculating and printing the elapsed time, it only calculates every one thousand messages. Additionally, we monitored message exchanges using the `tcpdump` command to assess the reliability of the elapsed time measured by the timestamps.

B SCTP Message Exchanges

B.1 Initiating an Association

SCTP uses a four-way handshake to initiate an association. When the server receives an SCTP INIT chunk, the implementation in Linux allocates temporarily a new association, i.e., sctp_association object, sends an SCTP INIT-ACK chunk, and frees the temporal association. When receiving an SCTP COOKIE-ECHO chunk, it allocates a new association.

B.2 Heartbeat

This mechanism is to probe the reachability of a particular destination transport address defined in the present association. In other words, this is used to see which destination address is active or idle and to measure round trip time (RTT) via the active destination address. by waiting for the response within the Retransmission Timeout (RTO).

RFC 4960 [1] specifies that each endpoint should send HEARTBEAT requests to its peer endpoint. When receiving the HEARTBEAT request, the endpoint should respond with the HEARTBEAT-ACK. This mechanism is enabled by default and the recommended interval is 30 seconds. When increasing the interval, we need to consider that the detection of lost ABORT messages, which have no ACK, will take longer.

This is a similar mechanism to the TCP keepalive mechanism, which is not enabled by default, and which has a default interval of two hours.

C Date Structures for SCTP Sockets

C.1 Generic Data Structures for Transport Protocols

The **inode** structure represents all the information of a file or directory needed by the kernel. The **file** structure represents an open file associated with a process. The **dentry** is an entry containing the path information such as the parent and child directories.

The **socket** is a general structure to hold the type and state in the socket layer. The **sock** structure is used common in all transport layer protocols and the socket options. This sock is cast each transport protocol, e.g., sctp_socket.

The **sk_buff** is a socket buffer to provide buffering and flow control. The buffered data in the socket is separately allocated and linked to a skbuff_head_cache. When cloning sk_buff for TCP segmentation, this is allocated from the slab cache, skbuff_fclone_cache.

There are two additional data structures for polling using the `epoll()` system call. The **eppoll_entry** structure holds a wait queue used by the event poll. The **epitem** is an event entry for each file descriptor to be added the eventpoll interface.

C.2 Date Structures for SCTP Sockets

The **sctp_socket** stores SCTP information per socket, e.g., the socket options, the heartbeat interval to be inherited by all new associations, and the pointer to the endpoint. The **sctp_endpoint** stores the logical sender or receiver of SCTP packets per socket. On a multihomed host, an SCTP endpoint is represented as a set of source/destination transport addresses. A one-to-one style socket, a TCP-style socket, has exactly one association on one endpoint, while a one-to-many style socket, a UDP-style socket, has multiple associations on one endpoint.

The**sctp_bind_addr** stores bind addresses (the local port and the list of IP addresses) common between the associations and endpoints. The **sctp_bind_bucket** holds the local port number only used at an SCTP client.

The **sctp_association** holds a recommended set of parameters in TCB (Transmission Control Block), e.g., the local bind address, cookie, the SCTP state, the current TSNs (Transmission Sequence Numbers), the initial parameters of the socket, and all information about the peers. It has approximately 100 attributes, as the data definition in `include/net/sctp/structs.h` is shown in Figures 11 and 12. We added data size for not well-known data structures.

The **sctp_transport** represents a remote transport address (IP address and port) and tracks the current retransmission timeout (RTO) value, RTT, current congestion window and heartbeat interval. The **sctp_ssnmap** tracks both the outbound and inbound stream sequence numbers, which assures sequenced delivery of the user messages within a give stream.

The **sctp_chunk** is a unit of information within an SCTP packet containing a pointer to the sk_buff. The **sctp_datamsg** is for a single message to track chunk fragments some of which has been acked, but the remaining has not.

D System Configuration

D.1 Server Configuration

The following command line increases the number of file descriptors in the system:

```
% echo 1048576 > /proc/sys/fs/file-max
```

1,048,576 (= 1024*1024) is the system limit defined as a constant, `NR_FILE` in `include/linux/fs.h`. To increase this limit when enough memory is installed, we need to modify and recompile the kernel.
To change the `/proc/sys` parameters at boot time, we need to add them to `/etc/sysctl.conf` as follows:

```
fs.file-max=1048576
```

The `ulimit` command can be used to increase the number of file descriptors per process:

```
% ulimit -n 1000000
```

To allow a remote shell to access a large number of file descriptors for our measurement, we need to specify the user name and the parameter in `/etc/security/limits.conf`:

```
special_user       soft   nofile   1000000
special_user       hard   nofile   1000000
```

To allow a remote shell to access a larger number of file descriptors invoked by `ssh`, we need to restart `sshd` in `/etc/rc.local`:

```
% ulimit -n 1000000
% /etc/rc.d/init.d/sshd restart
```

The memory space for TCP socket buffers is configured as follows:

```
net.ipv4.tcp_rmem = 4096        87380   174760
net.ipv4.tcp_wmem = 4096        16384   131072
net.ipv4.tcp_mem = 98304        131072  196608
```

These parameters are automatically configured at boot time based on available memory as well as TCP established and bind hash table entries. The variables are logged in `/var/sys/message` during boot:

```
kernel: TCP established hash table entries: 524288 (order: 10, 4194304 bytes)
kernel: TCP bind hash table entries: 65536 (order: 7, 524288 bytes)
kernel: TCP: Hash tables configured (established 524288 bind 65536)
```

D.2 Client Configuration

To increase the number of file descriptors for a shell, we again use the `ulimit` command:

```
% ulimit -n 60000
```

To increase the range of local ports, we modify the `ip_local_port_range` file:

```
% echo 10000   65535 > /proc/sys/net/ipv4/ip_local_port_range
```

```
struct sctp_association {

        struct sctp_ep_common base;                  //80 bytes
        struct list_head asocs;                      // 8 bytes
        sctp_assoc_t assoc_id;                       // 4 bytes
        struct sctp_endpoint *ep;
        struct sctp_cookie c;                        //168 bytes
        struct {
                __u32 rwnd;
                struct list_head transport_addr_list;//8 bytes
                __u16 transport_count;
                __u16 port;
                struct sctp_transport *primary_path;
                union sctp_addr primary_addr;        //28 bytes
                struct sctp_transport *active_path;
                struct sctp_transport *retran_path;
                struct sctp_transport *last_sent_to;
                struct sctp_transport *last_data_from;

                /*
                 * Mapping  An array of bits or bytes indicating which out of
                 * Array    order TSN's have been received (relative to the
                 *          Last Rcvd TSN). If no gaps exist, i.e. no out of
                 *          order packets have been received, this array
                 *          will be set to all zero. This structure may be
                 *          in the form of a circular buffer or bit array.
                 *
                 * Last Rcvd   : This is the last TSN received in
                 * TSN         : sequence. This value is set initially by
                 *             : taking the peer's Initial TSN, received in
                 *             : the INIT or INIT ACK chunk, and subtracting
                 *             : one from it.
                 *
                 * Throughout most of the specification this is called the
                 * "Cumulative TSN ACK Point".  In this case, we
                 * ignore the advice in 12.2 in favour of the term
                 * used in the bulk of the text.  This value is hidden
                 * in tsn_map--we get it by calling sctp_tsnmap_get_ctsn().
                 */
                struct sctp_tsnmap tsn_map;                          // 168 bytes
                __u8 _map[sctp_tsnmap_storage_size(SCTP_TSN_MAP_SIZE)];//4096 bytes

                __u8    sack_needed;
                __u8    ecn_capable;
                __u8    ipv4_address;
                __u8    ipv6_address;
                __u8    hostname_address;
                __u8    asconf_capable;
                __u8    prsctp_capable;
                __u32   adaptation_ind;
                __be16 addip_disabled_mask;
                struct sctp_inithdr_host i;             //16 bytes
                int cookie_len;
                void *cookie;
                __u32 addip_serial;
        } peer;
```

Figure 11: Data structure of *sctp_association* (contd.)

```
        sctp_state_t state;           //4 bytes
        struct timeval cookie_life;   //8 bytes
        int overall_error_count;
        unsigned long rto_initial;
        unsigned long rto_max;
        unsigned long rto_min;
        int max_burst;
        int max_retrans;
        __u16 max_init_attempts;
        __u16 init_retries;
        unsigned long max_init_timeo;
        unsigned long hbinterval;
        __u16 pathmaxrxt;
        __u32 pathmtu;
        __u32 param_flags;
        unsigned long sackdelay;
        unsigned long timeouts[SCTP_NUM_TIMEOUT_TYPES];
        struct timer_list timers[SCTP_NUM_TIMEOUT_TYPES]; //24 bytes
        struct sctp_transport *shutdown_last_sent_to;
        struct sctp_transport *init_last_sent_to;
        __u32 next_tsn;
        __u32 ctsn_ack_point;
        __u32 adv_peer_ack_point;
        __u32 highest_sacked;
        __u16 unack_data;
        __u32 rwnd;
        __u32 a_rwnd;
        __u32 rwnd_over;
        int sndbuf_used;
        atomic_t rmem_alloc;
        wait_queue_head_t       wait;                    //12 bytes
        __u32 frag_point;
        int init_err_counter;
        int init_cycle;
        __u16 default_stream;
        __u16 default_flags;
        __u32 default_ppid;
        __u32 default_context;
        __u32 default_timetolive;
        __u32 default_rcv_context;
        struct sctp_ssnmap *ssnmap;
        struct sctp_outq outqueue;                       //60 bytes
        struct sctp_ulpq ulpq;                           //40 bytes
        __u32 last_ecne_tsn;
        __u32 last_cwr_tsn;
        int numduptsns;
        __u32 autoclose;
        struct sctp_chunk *addip_last_asconf;
        struct sctp_chunk *addip_last_asconf_ack;
        struct list_head addip_chunk_list;               //8 bytes
        __u32 addip_serial;
        char need_ecne;
        char temp;
};
```

Figure 12: Data structure of *sctp_association*

D.3 Common Configuration for Server and Clients

Disabling Heartbeat

To eliminate extra process cost for the heartbeat, we disabled the SCTP heartbeat mechanism for both server and clients. For one-to-one sockets, we used the `setsockopt()` system call to disable the heartbeat mechanism just after creating the socket. The option name is `SCTP_PEER_ADDR_PARAM`. For Linux 2.6.23, the member of *sctp_paddrparams*, *spp_flags* should be set to `SPP_HB_DISABLE`. For Linux 2.6.9, *spp_flags* does not exit. Instead, *spp_hbinterval* should be set to 0.

However, it is not the same for one-to-many style socket, since the `SCTP_PEER_ADDR_PARAM` is valid for an association, not multiple associations for the socket. We can call the same `setsockopt()` system call with a valid association_id after an association is created and added to an existing socket. To get the association id, we need to use the notification mechanism, which is costly. To avoid the notification cost, we simply extended the interval of the heartbeat from 3,600 seconds to 360,000 seconds at the system level of the SUT.

```
% echo net.sctp.hb_interval=360000 >> /etc/sysctl.conf
```

Expanding the Size of Socket Buffers

For one-to-many sockets, multiple associations share a single socket buffer. This sharing easily exhausts the send and receive socket buffer and causes the error “Resource temporarily unavailable”. To avoid this resource starvation, we expanded the buffer size using the `setsockopt` system call. These option names are `SO_SNDBUF` for a send buffer and `SO_RCVBUF` for a receive buffer, respectively.

In our measurement, when an echo server handles 2,500 requests/second via a one-to-many socket whose send buffer is expanded to 262,142 bytes, the echo server couldn’t send echoed message due to the send buffer starvation. Since the maximum size of socket buffers is limited in `net.core.wmem_max`, we need to expand the value of `net.core.wmem_max` in order to raise the size of a send buffer.

Setting no Delay Option

Using the `setsockopt` system call, we set ”no delay” to a socket to disable the Nagel algorithm. The option name is `SCTP_NODELAY`. Note that SCTP preserves message boundaries, so it does not bundle multiple short messages into a single large IP packet, unlike TCP.