Linux SCTP is catching up and going above

Marcelo Ricardo Leitner
Red Hat Inc.
marcelo.leitner@gmail.com

Xin Long
Red Hat Inc.
lucien.xin@gmail.com

***Abstract*—This paper describes what has been recently done on Linux SCTP stack up to v4.19 and near future plans, from performance optimizations to new features.**

***Index Terms*—SCTP, lksctp, networking**

I. Introduction

SCTP (Stream Control Transmission Protocol) is a protocol that is currently defined in IETF RFC 4960 [1]. It originated from the IETF SIGTRAN working group, intended to carry signalling traffic over IP networks. They had requirements for many unique features back then and it was decided that neither UDP nor TCP would be a good fit and then SCTP was designed. SCTP has advanced features since day 0, such as multi-homing and multi-streaming.

The challenge has been on keeping up with the developments done on TCP, like overall stack optimizations (fast paths, for example) and congestion control developments (e.g., buffer accounting and its impact on rwnd/cwnd), and also to have more support from hardware vendors. Currently, Linux SCTP is not able to achieve similar transfer rates as TCP. While virtually every NIC out there can do TCP checksumming, there is only a few that can do SCTP CRC32c offloading.

The gap is not small, but we are trying to reduce it. In this paper we go through the recent developments done on SCTP stack and list our near future plans too.

Even though this paper was submitted by us both, we use 'we' throughout this document but meaning 'we' as in all SCTP contributors. Most of the work documented here was performed by us, Xin and Marcelo, yet there were contributions from the community, especially in the form of code reviews, and for those our sincere thanks.

A. Recap on SCTP

SCTP works with associations between endpoints. That is, an endpoint is the logical sender/receiver of SCTP packets and consists of a set of eligible address:port tuples (and a mix of IPv4 and IPv6 is allowed). The path between address:port from one endpoint to one address:port of the other endpoint is called transport. An association, a protocol relationship between endpoints, consists of one or more transports, though as per RFC 4960 only one transport can be actively used for sending new data at a time.

SCTP negotiates some features during association establishment, including the number of streams that will be used on it. Streams are a way of multiplexing the association. Messages sent on stream 0 are independent of messages sent on stream 1 and reordering amongst streams may happen, but not within a given stream.

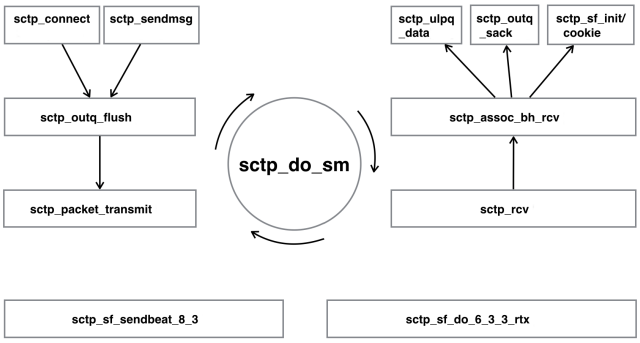

Fig. 1. Linux SCTP state machine.

B. Overview on Linux SCTP stack

Linux SCTP stack differs quite a lot from what we have for UDP or TCP. On the latter the code is more streamlined, allowing the creation of hot paths, while in the former we use a central state machine to process all incoming packets, events and operations. Fig. 1 illustrates it.

Everything goes through it. A sendmsg system call, for example, is split into two parts. The first deals with the user data and is done right away. But actually enqueueing the new message(s) for sending, that is done via the state machine depicted in Fig. 1 and is like a second step in sendmsg processing for SCTP.

Fig. 2 illustrates the main structs that are used in Linux SCTP stack. The figure is updated and includes the developments that will be described later in this paper.

II. Testing

The testing of Linux SCTP is now comprised of 3 test suites: TAHI, lksctp-tools unit tests and the rather new sctp-tests project.

TAHI [2] is a project whose objective is to ensure that the implementation is in accordance with the RFCs. It uses two systems, a control system and a DUT (Device Under Test), which is then asserted to be compliant. It will not, for example, stress test the stack.

lksctp-tools [3] is a userspace library of C helper functions to ease the usage of SCTP sockets and to abstract Linux

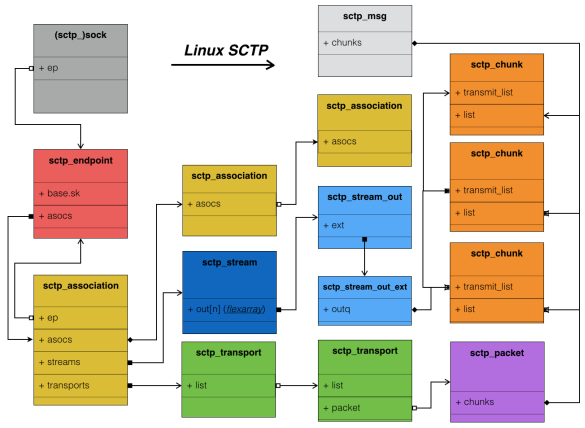

Fig. 2. Linux SCTP main structs.

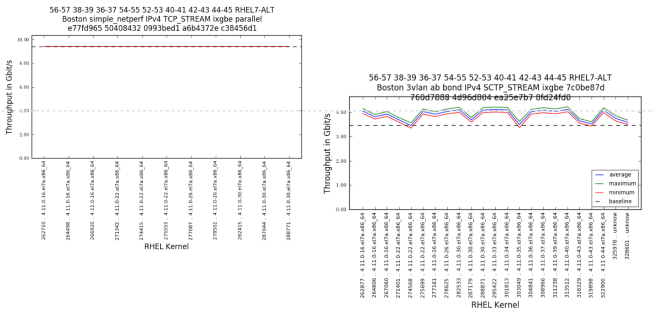

Fig. 3. Linux SCTP performance versus Linux TCP, RHEL kernel.

choices of implementation. SCTP CONNECTX operation, for example, is implemented using a sockopt, to avoid creating a new syscall for it, and then this library implements a function called 'sctp_connectx', which hides the sockopt handling within it. Its unit tests are aimed at testing the library itself.

So we had a gap on what could be done as for testing. We could not, for example, test Stream Schedulers implementation properly with only these. With that in mind, we started the new sctp-tests [4] project. In there we are collecting all test cases that do not fit the above projects.

It is implemented using bash scripts so that test cases are easy to write, while the framework does all the environment setup: create net namespaces, add proper veth pairs and IP addresses (v4 and v6). Supports client-server namespaces and client-router-server topologies.

III. Performance

As we alluded in the introduction, Linux SCTP performance is far from what it can do with TCP and Fig. 3 gives an idea of that difference.

Our current understanding is that this gap is composed of several factors. The very first one being the fact that SCTP protocol is in essence far more complex than TCP and the complications that arise from that are not small.

For TCP, sending a chunk of 16kB is a direct operation on its buffers. Segmentation is done in a way that it does not matter if the application sent more data after that or not. After all, it is one linear buffer. For SCTP, that implies in more processing. On sending side, now we support Stream Scheduling (part of RFC 8260 [5]). That means dequeuing the next to be sent is potentially a more complex operation than simply grabbing the first bytes out of the queue.

Receiving side is also more complicated. User messages on different streams may get fragmented and it is SCTP duty to defrag them and deliver to the application. Doing GRO for SCTP has limitations that likely render it not being worth doing. For example, unfragmented chunks are like UDP packets and should not be held up without prior knowledge that it will not add unwanted latency to the application, limiting which applications would benefit from it.

A. Transport rhashtable

SCTP supports what we call 'one-to-many' socket style. That is, it can use only 1 socket for N associations under it, similarly to how UDP works.

It was reported that when having more than 1000 associations under the same socket, CPU usage skyrocketed. That was because we had a simple global association hash table that hashed only local port and destination port. As all these associations under the same socket share the same local port, and the application was built to use the same port number on the other 1000 peers, it caused the hash entry to be a list of 1000 elements.

The fix for this consisted in 3 parts. First, now it also hashes destination IP address. Second, we moved to rhashtable/rhlist. Third, we are not hashing associations anymore but transports. Fig. 4 illustrates before and after the improvement.

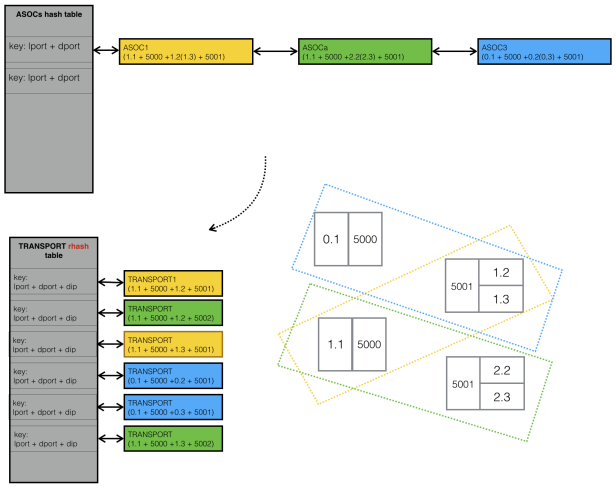

Fig. 4. Hashing done before and after the improvement.

Previously, we had to traverse that list of associations and look inside each one for a matching transport to confirm that that is the association we wanted. Now, when receiving a packet, finding to which association it belongs is way faster. A query on the rhashtable will likely return a list of possible transports. Then we look for the right local address in them too and that is it, we already know to which association this packet belongs.

We did not hash the local address too because SCTP does not negotiate which combinations of its multiple addresses the associations will use, meaning that we would have to hash all MxN combinations. Most of these combinations often goes just unused and does not justify the implementation complexity that it implies on.

One side effect of this change is that association listing on /proc was performed using the previous association hash. Now it traverses the transport rhash and uses the primary path information for not dumping the same association multiple times.

B. GSO

GSO is feasible for SCTP because it does not imply in added latencies and it does not depend on the kind of messages that the application is using. As long as the stack is willing to send, say, 4 packets at a time, for whichever streams the stream scheduler decides, we can batch them all and send it through the other layers only once.

But as usual for SCTP, some extra constraints apply. First one is that it needs to maintain packet boundaries. There is a stream scheduling algorithm, called Round-Robin Scheduler per Packet (SCTP_SS_RR_PKT, not implemented yet), which bundles only the chunks from the same stream in a packet. That means that if a stream is short on chunks and cannot fill a packet, yet there is another stream with data queued, GSO cannot work purely on packet/chunk sizes because that would cause the first packet to have chunks from multiple streams. There are other situations that lead to similar effect, like chunks that cannot be bundled with others and chunks that should appear only once in a packet. Another approach to these would be to trigger a flush in such situations and terminate the GSO packet when these conditions are detected, but that is not optimal and reduces GSO effectivity.

We want to preserve packet boundaries and gso_size is not enough for that. The options that we studied so far for using page frags were not feasible. Attempting to store packet boundaries in it led to constriction of GSO packet maximum size and/or having to parse the entire packet when splitting the GSO one, as SCTP has headers that are not always at the beginning.

For those reasons, SCTP GSO is using frag_list and not page frags. The caveat is that it is harder for drivers, including virtio, to offload frag_list.

Fig. 5–6 show how SCTP GSO works throughout the stack.

IV. Diagnostics

We added support for sock_diag and, with that, enabled ss tool to better work with SCTP sockets. Virtually any

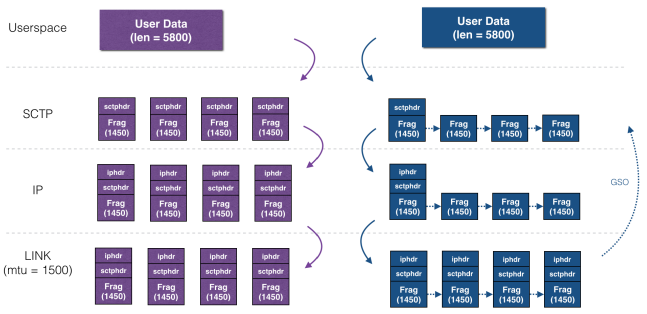

Fig. 5. SCTP without GSO and with GSO, tx side.

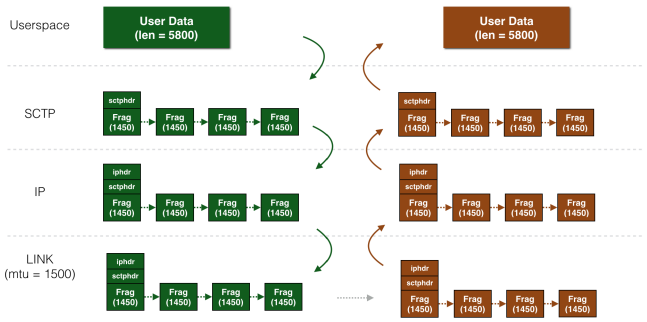

Fig. 6. SCTP with GSO, tx and rx sides.

```
[iproute2]# ss --sctp  -n -l
State       Recv-Q Send-Q   Local Address:Port       Peer Address:Port
LISTEN      0      128      172.16.254.254:8888      *:*
LISTEN      0      5        127.0.0.1:1234           *:*
LISTEN      0      5        127.0.0.1:1234           *:*
  - ESTAB   0      0        127.0.0.1%lo:1234        127.0.0.1:4321
LISTEN      0      128      172.16.254.254:8888      *:*
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.253.253:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.1.1:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.1.2:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.2.1:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.2.2:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.3.1:8888
  - ESTAB   0      0        172.16.254.254%eth1:8888 172.16.3.2:8888
LISTEN      0      0        127.0.0.1:4321           *:*
  - ESTAB   0      0        127.0.0.1%lo:4321        127.0.0.1:1234

[iproute2]# ss -Snai
State       Recv-Q Send-Q  Local Address:Port     Peer Address:Port
LISTEN      0      1           127.0.0.1:27375           *:*
locals:127.0.0.1,192.168.42.2, v4mapped:1
ESTAB       0      0           127.0.0.1:37636       127.0.0.1:27375
locals:0.0.0.0, v4mapped:1
```

Fig. 7. Sample outputs of ss using the new sock_diag interface.

information on the sockets and associations can be retrieved using it. Fig. 7 has 2 samples on ss outputs already using it.

For an extensive list on what is returned, please consult the struct sctp_info.

V. Important fixes

Dst source address selection: SCTP is likely to incorrectly choose source address if socket is bound to secondary addresses. This fix adds a new check that checks if such src address belongs to the interface that routing identified as output.

Rwnd improvements: SCTP rcvbuf handling works with a fixed ratio of overhead and payload which can result in

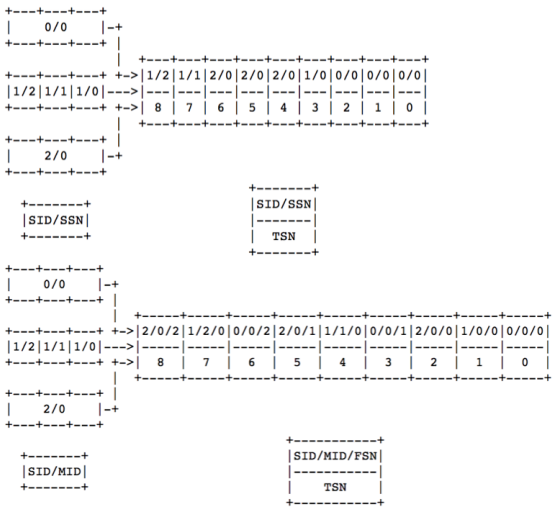

Fig. 8. Round-Robin Scheduler with User Message Interleaving.

overestimating the available buffer and thus the announcing of a rwnd bigger than what we can actually use. In some particular conditions it could trigger a situation on which it would not be able to grow the window back to a decent amount once it reached zero window.

PMTU discovery (critical) fixes: if the socket was busy when receiving the ICMP, it would not store the MTU information from the ICMP packet for later processing. Also, when receiving such ICMP on a one-to-many style socket, it was not updating the right dst entry.

Partial reliability fixes: remaining fragments should never be abandoned if any fragment has been sent out. All the remaining fragments must be abandoned if any fragment was abandoned.

Code refactors: rewrote many huge core functions like sctp_sendmsg(), sctp_outq_flush(), sctp_packet_transmit(); Improved MTU processing code; Fix many structures definition and type cast errors.

VI. New features

A. Stream Schedulers and User Message Interleaving (RFC 8260)

Stream Schedulers are used in TX side to choose which stream's data should be sent next according to a certain scheduler set by the user. There are 6 stream schedulers defined in the RFC. In the Linux stack we've currently implemented only FCFS, RR and PRIO. RR_PKT, FC and WFQ are not yet implemented although the implementation should be fairly simple.

User Message Interleaving fixes the head of line blocking issue described in the first part of Fig. 8: if the stream 1 had a higher priority than stream 0 but its messages were queued right after the stream 0 (by another thread, for instance), a head of line blocking happens because now the chunks from stream 1 must wait for the chunks from stream 0 to get sent (but not necessarily acked) to only then send the chunks from stream 1.

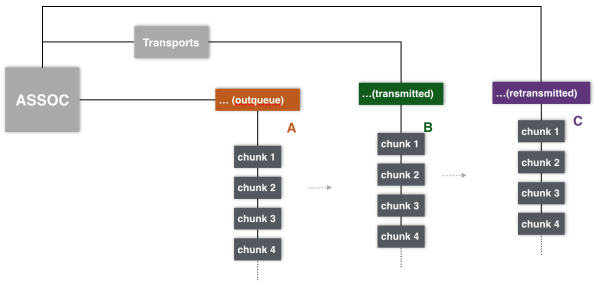

Fig. 9. The three send queues in SCTP.

In the new I-Data chunk two new fields are added (MID and FSN) and the SSN field is removed. MID (Message Identifier) is now used to identify all chunks for a given User Message and is also used to ensure the ordered delivery within the stream. The FSN (Fragment Sequence Number) is only used when fragmentation of the User Message is necessary and is a sequence number relative to only this User Message. Therefore, all fragments use the same MID number and the TSN field now is only used to ensure reliability.

With these changes, it is now possible to interrupt the sending of the message 0 on stream 0 in order to send something else. So in the previous example but now using the new I-Data chunks, they get scheduled like in the 2nd part of Fig. 8

B. Additional policies for Partially Reliable-SCTP (RFC 7496)

The Partially Reliable SCTP (PR-SCTP) extension defined in RFC 3758 [6] is another important feature, which provides a generic method for senders to abandon user messages. The decision to abandon a user message is sender-side only and the exact condition is called a "PR-SCTP policy". Three policies have been defined in RFC 7496 [7]:

- Timed Reliability, this allows the sender to specify a timeout for a user message after which the SCTP stack abandons the user message.
- Limited Retransmission Policy, allows a limitation of the number of retransmissions. Like in Fig. 9, this would be checked when dequeuing chunks from A, when dequeuing chunks from C, or when moving chunks from B to C. It will also check on B and C after receiving a SACK.
- Priority Policy, allows removal of lower priority messages if space for higher priority messages is needed in the send buffer. Like in Fig. 9, it could be checked before enqueuing chunk into A and No Enough TX Buffer, then try to drop from C to B to A.

C. Stream Reconfiguration (RFC 6525)

Many applications that use SCTP want the ability to "reset" a stream. The intention of resetting a stream is to set the SSNs of the stream back to 'zero' with a corresponding notification to the application layer that the reset has been performed. Applications requiring this feature want to "reuse" streams

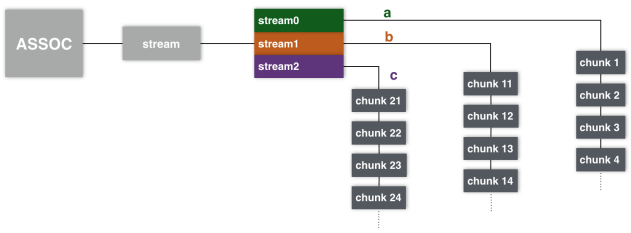

Fig. 10. One SCTP association with 3 stream queues.

for different purposes but still utilize the SSN so that the application can track the message flows.

With the feature in RFC 6525 [8], five requests could be performed, but for some of them there are a few conditions. Like in Fig. 10, when requesting:

- Add Outgoing Streams, no restrictions.
- Add Incoming Streams, no restrictions.
- Reset Outgoing Streams, before resetting stream 1, b has to be empty.
- Reset Incoming Streams, the peer will send Reset Outgoing Stream request for which it has to follow the above rule.
- Reset SSN/TSN: All queues have to be empty: A, B, C, a, b, c.

D. Additional sockets API extensions (RFC 6458)

RFC 6458 [9] adds extra socket API extensions. The function sctp_sendv() provides an extensible way for an application to communicate different send attributes to the SCTP stack when sending a message. With the new struct sctp_sendv_spa param, it would be able to set sndinfo, prinfo and authinfo at the same time for a simple msg.

```
struct sctp_sendv_spa {
  uint32_t sendv_flags;
  struct sctp_sndinfo sendv_sndinfo;
  struct sctp_prinfo sendv_prinfo;
  struct sctp_authinfo sendv_authinfo;
};
```

The function sctp_recvv() provides an extensible way for the SCTP stack to pass up different SCTP attributes associated with a received message to an application. With the new struct sctp_recvv_rn param, it is now able to receive an event with both rcvinfo and nxtinfo at once.

```
struct sctp_recvv_rn {
  struct sctp_rcvinfo recvv_rcvinfo;
  struct sctp_nxtinfo recvv_nxtinfo;
};
```

E. Full SELinux support

For security module support, three SCTP specific hooks have been implemented:

```
security_sctp_assoc_request()
security_sctp_bind_connect()
security_sctp_sk_clone()
```

Also, the following security hook has been utilized:

```
security_inet_conn_established()
```

The usage details of these hooks are described with the SELinux implementation in Documentation/security/SELinux-sctp.rst

VII. Summary of current issues

Frag pages should be used in SCTP GSO: As discussed in section III-B, SCTP has to keep its packet borders in the frags. For that it uses frag_list for its GSO process with a special gso_size GSO_BY_FRAGS to workaround the common process. This can be a problem in other subcomponents that are not aware of it and frag_list often cannot be supported by hardware, which would be a problem for SCTP hardware offload support in the future.

Reduce the number of memcpy throughout tx path: When sending a message, there will be at least two copies to do:

1) In the syscall, from userspace to datamsg.chunks.
2) When building the packet, from datamsg.chunks to the skb to be sent.

In a first glance, the second memcpy looks like not needed, but experiments indicate that generating non-linear packets by re-using datamsg.chunks instead of copying them ends up having a big impact on CRC32c calculation to the point that it defeats the purpose.

When sending large messages, SCTP will do the chunking right when copying from userspace. This triggers more memory allocations than of a single copy and if an MTU change happens in between, SCTP will have to resort to IP fragmentation.

The long-term project is SCTP zero copy, if the issues with CRC32c get solved.

VIII. Future plans

Other features from Draft RFCs: SCTP NAT and CMT are two big features that still need to be done for Linux SCTP, and both are multi-homing related. We have started them for quite a long time and have the working prototypes.

SCTP Performance Improvement: This includes rwnd/cwnd management, snd/rcvbuf auto-tuning and the optimization for some cpu-consuming functions like sctp_outq_sack(). This should also include SCTP GSO improvement and zero copy implementation.

Introduction of a congestion control framework: Without this, users can use only one algorithm. A similar one to TCP's should be added for SCTP as well.

GSO Offload on hardware proposal: There is no NIC supporting GSO hardware offload for SCTP yet so far and no standard on how to implement it either. SCTP GSO in general was not considered a good model for offloading and even virtio had issues with it. Some changes will be needed in order to make hardware offloading feasible.

Userspace tools development: lksctp-tools and sctp-tests are the two main userspace tools, which provide some interfaces for users and some unit/regression test cases for developers. We are planning to improve the code quality for lksctp-tools and add more test cases into sctp-tests.

REFERENCES

[1] "Stream Control Transmission Protocol," RFC 4960, Sep. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4960.txt

[2] "TAHI branch being used at Red Hat." [Online]. Available: https://github.com/jijianwen/deprecated-sctp

[3] "lksctp-tools project." [Online]. Available: https://github.com/sctp/lksctp-tools

[4] "sctp-tests project." [Online]. Available: https://github.com/sctp/sctp-tests

[5] R. R. Stewart, M. Txen, S. Loreto, and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol," RFC 8260, Nov. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8260.txt

[6] D. M. A. Ramalho, M. Txen, and P. Conrad, "Stream Control Transmission Protocol (SCTP) Partial Reliability Extension," RFC 3758, May 2004. [Online]. Available: https://rfc-editor.org/rfc/rfc3758.txt

[7] M. Txen, R. Seggelmann, R. R. Stewart, and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension," RFC 7496, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7496.txt

[8] R. R. Stewart, M. Txen, and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration," RFC 6525, Mar. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6525.txt

[9] M. Txen, V. Yasevich, P. Lei, R. R. Stewart, and K. Poon, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)," RFC 6458, Dec. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6458.txt