Stream:	Internet Engineering Task Force (IETF)
RFC:	9260
Obsoletes:	4460, 4960, 6096, 7053, 8540
Category:	Standards Track
Published:	May 2022
ISSN:	2070-1721
Authors:	R. Stewart	M. Tüxen	K. Nielsen
---	---	---	---
	Netflix, Inc.	*Münster Univ. of Appl. Sciences*	*Kamstrup A/S*

RFC 9260
Stream Control Transmission Protocol

Abstract

This document describes the Stream Control Transmission Protocol (SCTP) and obsoletes RFC 4960. It incorporates the specification of the chunk flags registry from RFC 6096 and the specification of the I bit of DATA chunks from RFC 7053. Therefore, RFCs 6096 and 7053 are also obsoleted by this document. In addition, RFCs 4460 and 8540, which describe errata for SCTP, are obsoleted by this document.

SCTP was originally designed to transport Public Switched Telephone Network (PSTN) signaling messages over IP networks. It is also suited to be used for other applications, for example, WebRTC.

SCTP is a reliable transport protocol operating on top of a connectionless packet network, such as IP. It offers the following services to its users:

- acknowledged error-free, non-duplicated transfer of user data,
- data fragmentation to conform to discovered Path Maximum Transmission Unit (PMTU) size,
- sequenced delivery of user messages within multiple streams, with an option for order-of-arrival delivery of individual user messages,
- optional bundling of multiple user messages into a single SCTP packet, and
- network-level fault tolerance through supporting of multi-homing at either or both ends of an association.

The design of SCTP includes appropriate congestion avoidance behavior and resistance to flooding and masquerade attacks.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9260.

Copyright Notice

Table of Contents

1. Introduction
   1.1. Motivation
   1.2. Architectural View of SCTP
   1.3. Key Terms
   1.4. Abbreviations
   1.5. Functional View of SCTP
      1.5.1. Association Startup and Takedown
      1.5.2. Sequenced Delivery within Streams
      1.5.3. User Data Fragmentation
      1.5.4. Acknowledgement and Congestion Avoidance

1.5.5. Chunk Bundling
1.5.6. Packet Validation
1.5.7. Path Management
1.6. Serial Number Arithmetic
1.7. Changes from RFC 4960
2. Conventions
3. SCTP Packet Format
3.1. SCTP Common Header Field Descriptions
3.2. Chunk Field Descriptions
3.2.1. Optional/Variable-Length Parameter Format
3.2.2. Reporting of Unrecognized Parameters
3.3. SCTP Chunk Definitions
3.3.1. Payload Data (DATA) (0)
3.3.2. Initiation (INIT) (1)
3.3.2.1. Optional or Variable-Length Parameters in INIT chunks
3.3.3. Initiation Acknowledgement (INIT ACK) (2)
3.3.3.1. Optional or Variable-Length Parameters in INIT ACK Chunks
3.3.4. Selective Acknowledgement (SACK) (3)
3.3.5. Heartbeat Request (HEARTBEAT) (4)
3.3.6. Heartbeat Acknowledgement (HEARTBEAT ACK) (5)
3.3.7. Abort Association (ABORT) (6)
3.3.8. Shutdown Association (SHUTDOWN) (7)
3.3.9. Shutdown Acknowledgement (SHUTDOWN ACK) (8)
3.3.10. Operation Error (ERROR) (9)
3.3.10.1. Invalid Stream Identifier (1)
3.3.10.2. Missing Mandatory Parameter (2)
3.3.10.3. Stale Cookie (3)
3.3.10.4. Out of Resource (4)
3.3.10.5. Unresolvable Address (5)
3.3.10.6. Unrecognized Chunk Type (6)

3.3.10.7. Invalid Mandatory Parameter (7)
3.3.10.8. Unrecognized Parameters (8)
3.3.10.9. No User Data (9)
3.3.10.10. Cookie Received While Shutting Down (10)
3.3.10.11. Restart of an Association with New Addresses (11)
3.3.10.12. User-Initiated Abort (12)
3.3.10.13. Protocol Violation (13)
3.3.11. Cookie Echo (COOKIE ECHO) (10)
3.3.12. Cookie Acknowledgement (COOKIE ACK) (11)
3.3.13. Shutdown Complete (SHUTDOWN COMPLETE) (14)
4. SCTP Association State Diagram
5. Association Initialization
5.1. Normal Establishment of an Association
5.1.1. Handle Stream Parameters
5.1.2. Handle Address Parameters
5.1.3. Generating State Cookie
5.1.4. State Cookie Processing
5.1.5. State Cookie Authentication
5.1.6. An Example of Normal Association Establishment
5.2. Handle Duplicate or Unexpected INIT, INIT ACK, COOKIE ECHO, and COOKIE ACK Chunks
5.2.1. INIT Chunk Received in COOKIE-WAIT or COOKIE-ECHOED State (Item B)
5.2.2. Unexpected INIT Chunk in States Other than CLOSED, COOKIE-ECHOED, COOKIE-WAIT, and SHUTDOWN-ACK-SENT
5.2.3. Unexpected INIT ACK Chunk
5.2.4. Handle a COOKIE ECHO Chunk When a TCB Exists
5.2.4.1. An Example of an Association Restart
5.2.5. Handle Duplicate COOKIE ACK Chunk
5.2.6. Handle Stale Cookie Error
5.3. Other Initialization Issues
5.3.1. Selection of Tag Value

5.4. Path Verification
6. User Data Transfer
6.1. Transmission of DATA Chunks
6.2. Acknowledgement on Reception of DATA Chunks
6.2.1. Processing a Received SACK Chunk
6.3. Management of Retransmission Timer
6.3.1. RTO Calculation
6.3.2. Retransmission Timer Rules
6.3.3. Handle T3-rtx Expiration
6.4. Multi-Homed SCTP Endpoints
6.4.1. Failover from an Inactive Destination Address
6.5. Stream Identifier and Stream Sequence Number
6.6. Ordered and Unordered Delivery
6.7. Report Gaps in Received DATA TSNs
6.8. CRC32c Checksum Calculation
6.9. Fragmentation and Reassembly
6.10. Bundling
7. Congestion Control
7.1. SCTP Differences from TCP Congestion Control
7.2. SCTP Slow-Start and Congestion Avoidance
7.2.1. Slow-Start
7.2.2. Congestion Avoidance
7.2.3. Congestion Control
7.2.4. Fast Retransmit on Gap Reports
7.2.5. Reinitialization
7.2.5.1. Change of Differentiated Services Code Points
7.2.5.2. Change of Routes
7.3. PMTU Discovery
8. Fault Management
8.1. Endpoint Failure Detection

8.2. Path Failure Detection
8.3. Path Heartbeat
8.4. Handle "Out of the Blue" Packets
8.5. Verification Tag
8.5.1. Exceptions in Verification Tag Rules
9. Termination of Association
9.1. Abort of an Association
9.2. Shutdown of an Association
10. ICMP Handling
11. Interface with Upper Layer
11.1. ULP-to-SCTP
11.1.1. Initialize
11.1.2. Associate
11.1.3. Shutdown
11.1.4. Abort
11.1.5. Send
11.1.6. Set Primary
11.1.7. Receive
11.1.8. Status
11.1.9. Change Heartbeat
11.1.10. Request Heartbeat
11.1.11. Get SRTT Report
11.1.12. Set Failure Threshold
11.1.13. Set Protocol Parameters
11.1.14. Receive Unsent Message
11.1.15. Receive Unacknowledged Message
11.1.16. Destroy SCTP Instance
11.2. SCTP-to-ULP
11.2.1. DATA ARRIVE Notification
11.2.2. SEND FAILURE Notification

11.2.3. NETWORK STATUS CHANGE Notification
11.2.4. COMMUNICATION UP Notification
11.2.5. COMMUNICATION LOST Notification
11.2.6. COMMUNICATION ERROR Notification
11.2.7. RESTART Notification
11.2.8. SHUTDOWN COMPLETE Notification
12. Security Considerations
12.1. Security Objectives
12.2. SCTP Responses to Potential Threats
12.2.1. Countering Insider Attacks
12.2.2. Protecting against Data Corruption in the Network
12.2.3. Protecting Confidentiality
12.2.4. Protecting against Blind Denial-of-Service Attacks
12.2.4.1. Flooding
12.2.4.2. Blind Masquerade
12.2.4.3. Improper Monopolization of Services
12.3. SCTP Interactions with Firewalls
12.4. Protection of Non-SCTP-capable Hosts
13. Network Management Considerations
14. Recommended Transmission Control Block (TCB) Parameters
14.1. Parameters Necessary for the SCTP Instance
14.2. Parameters Necessary per Association (i.e., the TCB)
14.3. Per Transport Address Data
14.4. General Parameters Needed
15. IANA Considerations
15.1. IETF-Defined Chunk Extension
15.2. IETF-Defined Chunk Flags Registration
15.3. IETF-Defined Chunk Parameter Extension
15.4. IETF-Defined Additional Error Causes
15.5. Payload Protocol Identifiers

15.6. Port Numbers Registry

16. Suggested SCTP Protocol Parameter Values

17. References

17.1. Normative References

17.2. Informative References

Appendix A. CRC32c Checksum Calculation

Acknowledgements

Authors' Addresses

1. Introduction

This section explains the reasoning behind the development of the Stream Control Transmission Protocol (SCTP), the services it offers, and the basic concepts needed to understand the detailed description of the protocol.

This document obsoletes [RFC4960]. In addition to that, it incorporates the specification of the chunk flags registry from [RFC6096] and the specification of the I bit of DATA chunks from [RFC7053]. Therefore, [RFC6096] and [RFC7053] are also obsoleted by this document.

1.1. Motivation

TCP [RFC0793] has performed immense service as the primary means of reliable data transfer in IP networks. However, an increasing number of recent applications have found TCP too limiting and have incorporated their own reliable data transfer protocol on top of UDP [RFC0768]. The limitations that users have wished to bypass include the following:

- TCP provides both reliable data transfer and strict order-of-transmission delivery of data. Some applications need reliable transfer without sequence maintenance, while others would be satisfied with partial ordering of the data. In both of these cases, the head-of-line blocking offered by TCP causes unnecessary delay.
- The stream-oriented nature of TCP is often an inconvenience. Applications add their own record marking to delineate their messages and make explicit use of the push facility to ensure that a complete message is transferred in a reasonable time.
- The limited scope of TCP sockets complicates the task of providing highly available data transfer capability using multi-homed hosts.
- TCP is relatively vulnerable to denial-of-service attacks, such as SYN attacks.

Transport of PSTN signaling across the IP network is an application for which all of these limitations of TCP are relevant. While this application directly motivated the development of SCTP, other applications might find SCTP a good match to their requirements. One example of this is the use of data channels in the WebRTC infrastructure.

1.2. Architectural View of SCTP

SCTP is viewed as a layer between the SCTP user application ("SCTP user" for short) and a connectionless packet network service, such as IP. The remainder of this document assumes SCTP runs on top of IP. The basic service offered by SCTP is the reliable transfer of user messages between peer SCTP users. It performs this service within the context of an association between two SCTP endpoints. Section 11 of this document sketches the API that exists at the boundary between SCTP and the SCTP upper layers.

SCTP is connection oriented in nature, but the SCTP association is a broader concept than the TCP connection. SCTP provides the means for each SCTP endpoint (Section 1.3) to provide the other endpoint (during association startup) with a list of transport addresses (i.e., multiple IP addresses in combination with an SCTP port) through which that endpoint can be reached and from which it will originate SCTP packets. The association spans transfers over all of the possible source/destination combinations that can be generated from each endpoint's lists.

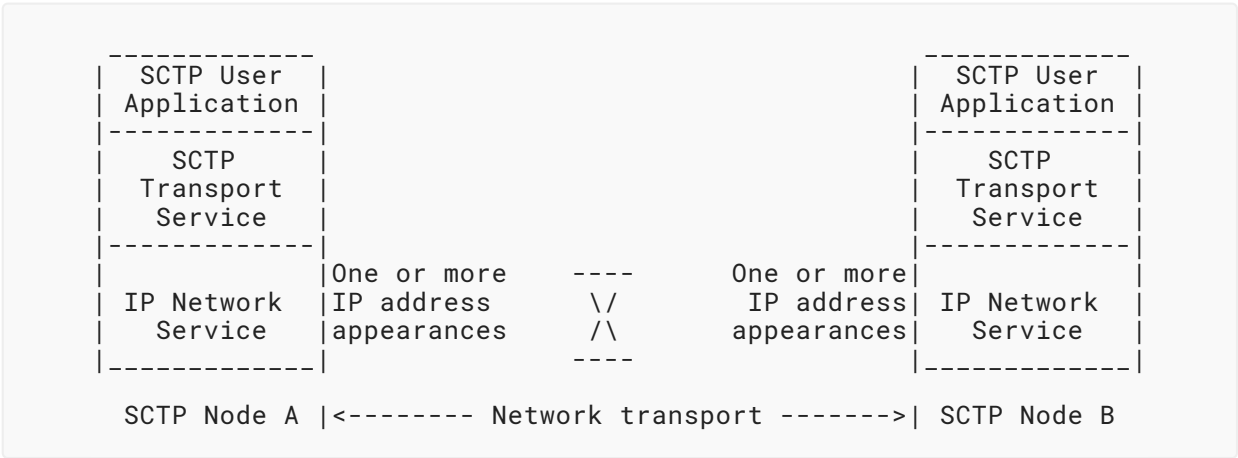


Figure 1: An SCTP Association

In addition to encapsulating SCTP packets in IPv4 or IPv6, it is also possible to encapsulate SCTP packets in UDP as specified in [RFC6951] or encapsulate them in DTLS as specified in [RFC8261].

1.3. Key Terms

Some of the language used to describe SCTP has been introduced in the previous sections. This section provides a consolidated list of the key terms and their definitions.

Active Destination Transport Address: A transport address on a peer endpoint that a transmitting endpoint considers available for receiving user messages.

Association Maximum DATA Chunk Size (AMDCS): The smallest Path Maximum DATA Chunk Size (PMDCS) of all destination addresses.

Bundling of Chunks: An optional multiplexing operation, whereby more than one chunk can be carried in the same SCTP packet.

Bundling of User Messages: An optional multiplexing operation, whereby more than one user message can be carried in the same SCTP packet. Each user message occupies its own DATA chunk.

Chunk: A unit of information within an SCTP packet, consisting of a chunk header and chunk-specific content.

Congestion Window (cwnd): An SCTP variable that limits outstanding data, in number of bytes, that a sender can send to a particular destination transport address before receiving an acknowledgement.

Control Chunk: A chunk not being used for transmitting user data, i.e., every chunk that is not a DATA chunk.

Cumulative TSN Ack Point: The Transmission Sequence Number (TSN) of the last DATA chunk acknowledged via the Cumulative TSN Ack field of a SACK chunk.

Flightsize: The number of bytes of outstanding data to a particular destination transport address at any given time.

Idle Destination Address: An address that has not had user messages sent to it within some length of time, normally the 'HB.interval' or greater.

Inactive Destination Transport Address: An address that is considered inactive due to errors and unavailable to transport user messages.

Message (or User Message): Data submitted to SCTP by the Upper-Layer Protocol (ULP).

Network Byte Order: Most significant byte first, a.k.a., big endian.

Ordered Message: A user message that is delivered in order with respect to all previous user messages sent within the stream on which the message was sent.

Outstanding Data (or Data Outstanding or Data In Flight): The total size of the DATA chunks associated with outstanding TSNs. A retransmitted DATA chunk is counted once in outstanding data. A DATA chunk that is classified as lost but that has not yet been retransmitted is not in outstanding data.

Outstanding TSN (at an SCTP Endpoint): A TSN (and the associated DATA chunk) that has been sent by the endpoint but for which it has not yet received an acknowledgement.

"Out of the Blue" (OOTB) Packet: A correctly formed packet, for which the receiver cannot identify the association it belongs to. See Section 8.4.

Path: The route taken by the SCTP packets sent by one SCTP endpoint to a specific destination transport address of its peer SCTP endpoint. Sending to different destination transport addresses does not necessarily guarantee getting separate paths. Within this specification, a path is identified by the destination transport address, since the routing is assumed to be stable. This includes, in particular, the source address being selected when sending packets to the destination address.

Path Maximum DATA Chunk Size (PMDCS): The maximum size (including the DATA chunk header) of a DATA chunk that fits into an SCTP packet not exceeding the PMTU of a particular destination address.

Path Maximum Transmission Unit (PMTU): The maximum size (including the SCTP common header and all chunks including their paddings) of an SCTP packet that can be sent to a particular destination address without using IP-level fragmentation.

Primary Path: The destination and source address that will be put into a packet outbound to the peer endpoint by default. The definition includes the source address since an implementation **MAY** wish to specify both destination and source address to better control the return path taken by reply chunks and on which interface the packet is transmitted when the data sender is multi-homed.

Receiver Window (rwnd): An SCTP variable a data sender uses to store the most recently calculated receiver window of its peer, in number of bytes. This gives the sender an indication of the space available in the receiver's inbound buffer.

SCTP Association: A protocol relationship between SCTP endpoints, composed of the two SCTP endpoints and protocol state information, including Verification Tags and the currently active set of Transmission Sequence Numbers (TSNs), etc. An association can be uniquely identified by the transport addresses used by the endpoints in the association. Two SCTP endpoints **MUST NOT** have more than one SCTP association between them at any given time.

SCTP Endpoint: The logical sender/receiver of SCTP packets. On a multi-homed host, an SCTP endpoint is represented to its peers as a combination of a set of eligible destination transport addresses to which SCTP packets can be sent and a set of eligible source transport addresses from which SCTP packets can be received. All transport addresses used by an SCTP endpoint **MUST** use the same port number but can use multiple IP addresses. A transport address used by an SCTP endpoint **MUST NOT** be used by another SCTP endpoint. In other words, a transport address is unique to an SCTP endpoint.

SCTP Packet (or Packet): The unit of data delivery across the interface between SCTP and the connectionless packet network (e.g., IP). An SCTP packet includes the common SCTP header, possible SCTP control chunks, and user data encapsulated within SCTP DATA chunks.

SCTP User Application (or SCTP User): The logical higher-layer application entity that uses the services of SCTP, also called the Upper-Layer Protocol (ULP).

Slow-Start Threshold (ssthresh): An SCTP variable. This is the threshold that the endpoint will use to determine whether to perform slow-start or congestion avoidance on a particular destination transport address. Ssthresh is in number of bytes.

State Cookie: A container of all information needed to establish an association.

Stream: A unidirectional logical channel established from one to another associated SCTP endpoint, within which all user messages are delivered in sequence, except for those submitted to the unordered delivery service.

Note: The relationship between stream numbers in opposite directions is strictly a matter of how the applications use them. It is the responsibility of the SCTP user to create and manage these correlations if they are so desired.

Stream Sequence Number: A 16-bit sequence number used internally by SCTP to ensure sequenced delivery of the user messages within a given stream. One Stream Sequence Number is attached to each ordered user message.

Tie-Tags: Two 32-bit random numbers that together make a 64-bit nonce. These tags are used within a State Cookie and TCB so that a newly restarting association can be linked to the original association within the endpoint that did not restart and yet not reveal the true Verification Tags of an existing association.

Transmission Control Block (TCB): An internal data structure created by an SCTP endpoint for each of its existing SCTP associations to other SCTP endpoints. TCB contains all the status and operational information for the endpoint to maintain and manage the corresponding association.

Transmission Sequence Number (TSN): A 32-bit sequence number used internally by SCTP. One TSN is attached to each chunk containing user data to permit the receiving SCTP endpoint to acknowledge its receipt and detect duplicate deliveries.

Transport Address: A transport address is typically defined by a network-layer address, a transport-layer protocol, and a transport-layer port number. In the case of SCTP running over IP, a transport address is defined by the combination of an IP address and an SCTP port number (where SCTP is the transport protocol).

Unordered Message: Unordered messages are "unordered" with respect to any other message; this includes both other unordered messages as well as other ordered messages. An unordered message might be delivered prior to or later than ordered messages sent on the same stream.

User Message: The unit of data delivery across the interface between SCTP and its user.

Verification Tag: A 32-bit unsigned integer that is randomly generated. The Verification Tag provides a key that allows a receiver to verify that the SCTP packet belongs to the current association and is not an old or stale packet from a previous association.

1.4. Abbreviations

MAC Message Authentication Code [RFC2104]

RTO Retransmission Timeout

RTT Round-Trip Time

RTTVAR Round-Trip Time Variation

SCTP Stream Control Transmission Protocol

SRTT Smoothed RTT

TCB Transmission Control Block

TLV Type-Length-Value coding format

TSN Transmission Sequence Number

ULP Upper-Layer Protocol

1.5. Functional View of SCTP

The SCTP transport service can be decomposed into a number of functions. These are depicted in Figure 2 and explained in the remainder of this section.

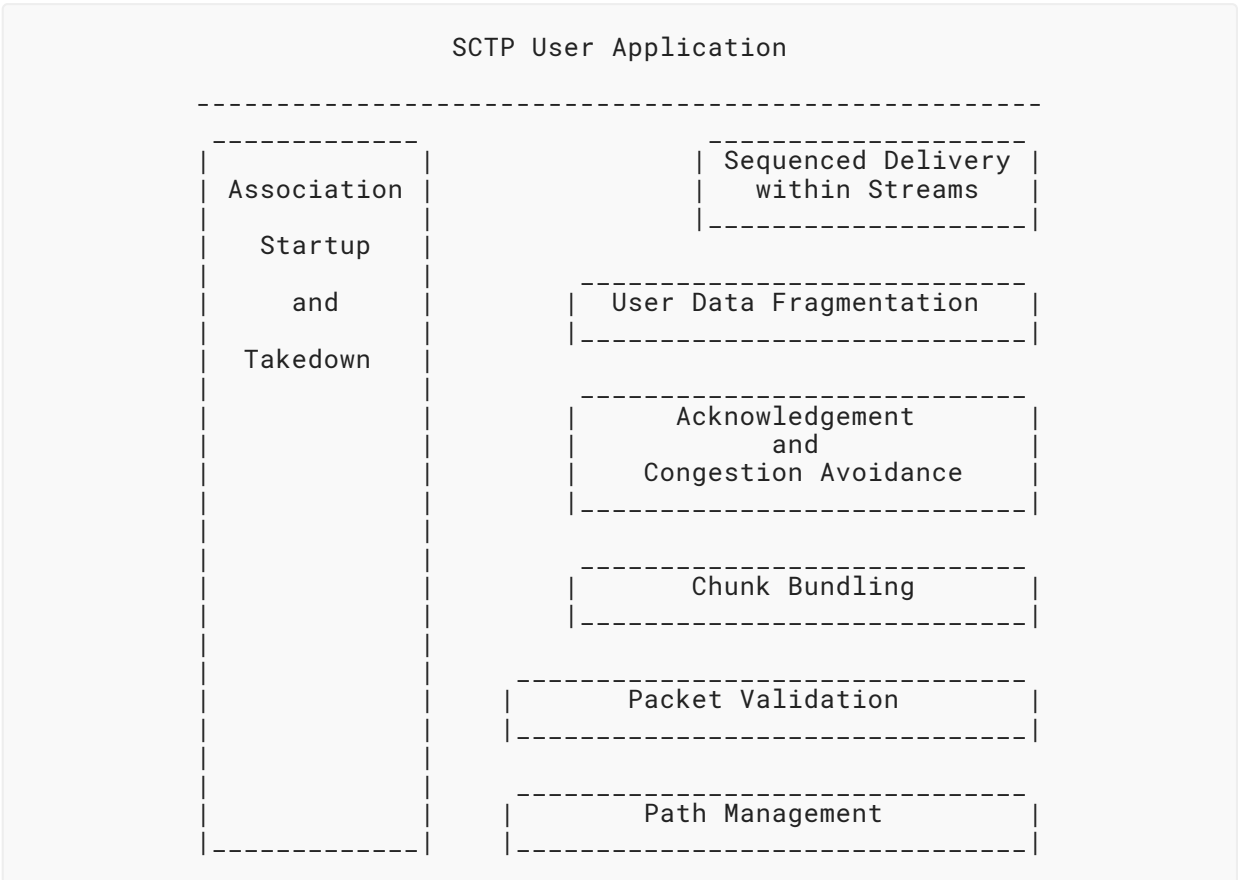


Figure 2: Functional View of the SCTP Transport Service

1.5.1. Association Startup and Takedown

An association is initiated by a request from the SCTP user (see the description of the ASSOCIATE (or SEND) primitive in Section 11).

A cookie mechanism, similar to one described by Karn and Simpson in [RFC2522], is employed during the initialization to provide protection against synchronization attacks. The cookie mechanism uses a four-way handshake, the last two legs of which are allowed to carry user data for fast setup. The startup sequence is described in Section 5 of this document.

SCTP provides for graceful close (i.e., shutdown) of an active association on request from the SCTP user. See the description of the SHUTDOWN primitive in Section 11. SCTP also allows ungraceful close (i.e., abort), either on request from the user (ABORT primitive) or as a result of an error condition detected within the SCTP layer. Section 9 describes both the graceful and the ungraceful close procedures.

SCTP does not support a half-open state (like TCP) wherein one side continues sending data while the other end is closed. When either endpoint performs a shutdown, the association on each peer will stop accepting new data from its user and only deliver data in queue at the time of the graceful close (see Section 9).

1.5.2. Sequenced Delivery within Streams

The term "stream" is used in SCTP to refer to a sequence of user messages that are to be delivered to the upper-layer protocol in order with respect to other messages within the same stream. This is in contrast to its usage in TCP, where it refers to a sequence of bytes (in this document, a byte is assumed to be 8 bits).

At association startup time, the SCTP user can specify the number of streams to be supported by the association. This number is negotiated with the remote end (see Section 5.1.1). User messages are associated with stream numbers (SEND, RECEIVE primitives; Section 11). Internally, SCTP assigns a Stream Sequence Number to each message passed to it by the SCTP user. On the receiving side, SCTP ensures that messages are delivered to the SCTP user in sequence within a given stream. However, while one stream might be blocked waiting for the next in-sequence user message, delivery from other streams might proceed.

SCTP provides a mechanism for bypassing the sequenced delivery service. User messages sent using this mechanism are delivered to the SCTP user as soon as they are received.

1.5.3. User Data Fragmentation

When needed, SCTP fragments user messages to ensure that the size of the SCTP packet passed to the lower layer does not exceed the PMTU. Once a user message has been fragmented, this fragmentation cannot be changed anymore. On receipt, fragments are reassembled into complete messages before being passed to the SCTP user.

1.5.4. Acknowledgement and Congestion Avoidance

SCTP assigns a Transmission Sequence Number (TSN) to each user data fragment or unfragmented message. The TSN is independent of any Stream Sequence Number assigned at the stream level. The receiving end acknowledges all TSNs received, even if there are gaps in the sequence. If a user data fragment or unfragmented message needs to be retransmitted, the TSN assigned to it is used. In this way, reliable delivery is kept functionally separate from sequenced stream delivery.

The acknowledgement and congestion avoidance function is responsible for packet retransmission when timely acknowledgement has not been received. Packet retransmission is conditioned by congestion avoidance procedures similar to those used for TCP. See Sections 6 and 7 for detailed descriptions of the protocol procedures associated with this function.

1.5.5. Chunk Bundling

As described in Section 3, the SCTP packet as delivered to the lower layer consists of a common header followed by one or more chunks. Each chunk contains either user data or SCTP control information. An SCTP implementation supporting bundling on the sender side might delay the sending of user messages to allow the corresponding DATA chunks to be bundled.

The SCTP user has the option to request that an SCTP implementation does not delay the sending of a user message just for this purpose. However, even if the SCTP user has chosen this option, the SCTP implementation might delay the sending due to other reasons (for example, due to congestion control or flow control) and might also bundle multiple DATA chunks, if possible.

1.5.6. Packet Validation

A mandatory Verification Tag field and a 32-bit checksum field (see Appendix A for a description of the 32-bit Cyclic Redundancy Check (CRC32c) checksum) are included in the SCTP common header. The Verification Tag value is chosen by each end of the association during association startup. Packets received without the expected Verification Tag value are discarded, as a protection against blind masquerade attacks and against stale SCTP packets from a previous association. The CRC32c checksum is set by the sender of each SCTP packet to provide additional protection against data corruption in the network. The receiver of an SCTP packet with an invalid CRC32c checksum silently discards the packet.

1.5.7. Path Management

The sending SCTP user is able to manipulate the set of transport addresses used as destinations for SCTP packets through the primitives described in Section 11. The SCTP path management function monitors reachability through heartbeats when other packet traffic is inadequate to provide this information and advises the SCTP user when reachability of any transport address of the peer endpoint changes. The path management function chooses the destination transport address for each outgoing SCTP packet based on the SCTP user's instructions and the currently perceived reachability status of the eligible destination set. The path management function is also responsible for reporting the eligible set of local transport addresses to the peer endpoint during association startup and for reporting the transport addresses returned from the peer endpoint to the SCTP user.

At association startup, a primary path is defined for each SCTP endpoint and is used to send SCTP packets normally.

On the receiving end, the path management is responsible for verifying the existence of a valid SCTP association to which the inbound SCTP packet belongs before passing it for further processing.

Note: Path Management and Packet Validation are done at the same time; although described separately above, in reality, they cannot be performed as separate items.

1.6. Serial Number Arithmetic

It is essential to remember that the actual Transmission Sequence Number space is finite, though very large. This space ranges from 0 to 2^{32} - 1. Since the space is finite, all arithmetic dealing with Transmission Sequence Numbers **MUST** be performed modulo 2^{32}. This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from 2^{32} - 1 to 0 again. There are some subtleties to computer modulo arithmetic, so great care has to be taken in programming the comparison of such values. When referring to TSNs, the symbol "<=" means "less than or equal" (modulo 2^{32}).

Comparisons and arithmetic on TSNs in this document **SHOULD** use Serial Number Arithmetic, as defined in [RFC1982], where SERIAL_BITS = 32.

An endpoint **SHOULD NOT** transmit a DATA chunk with a TSN that is more than 2^{31} - 1 above the beginning TSN of its current send window. Doing so will cause problems in comparing TSNs.

Transmission Sequence Numbers wrap around when they reach 2^{32} - 1. That is, the next TSN a DATA chunk **MUST** use after transmitting TSN = 2^{32} - 1 is TSN = 0.

Any arithmetic done on Stream Sequence Numbers **SHOULD** use Serial Number Arithmetic, as defined in [RFC1982], where SERIAL_BITS = 16. All other arithmetic and comparisons in this document use normal arithmetic.

1.7. Changes from RFC 4960

SCTP was originally defined in [RFC4960], which this document obsoletes. Readers interested in the details of the various changes that this document incorporates are asked to consult [RFC8540].

In addition to these and further editorial changes, the following changes have been incorporated in this document:

- Update references.
- Improve the language related to requirements levels.
- Allow the ASSOCIATE primitive to take multiple remote addresses; also refer to the socket API specification.
- Refer to the Packetization Layer Path MTU Discovery (PLPMTUD) specification for path MTU discovery.
- Move the description of ICMP handling from the Appendix to the main text.
- Remove the Appendix describing Explicit Congestion Notification (ECN) handling from the document.
- Describe the packet size handling more precisely by introducing PMTU, PMDCS, and AMDCS.
- Add the definition of control chunk.
- Improve the description of the handling of INIT and INIT ACK chunks with invalid mandatory parameters.
- Allow using L > 1 for Appropriate Byte Counting (ABC) during slow start.
- Explicitly describe the reinitialization of the congestion controller on route changes.
- Improve the terminology to make it clear that this specification does not describe a full mesh architecture.
- Improve the description of sequence number generation (Transmission Sequence Number and Stream Sequence Number).
- Improve the description of reneging.
- Don't require the change of the Cumulative TSN Ack anymore for increasing the congestion window. This improves the consistency with the handling in congestion avoidance.
- Improve the description of the State Cookie.
- Fix the API for retrieving messages in case of association failures.

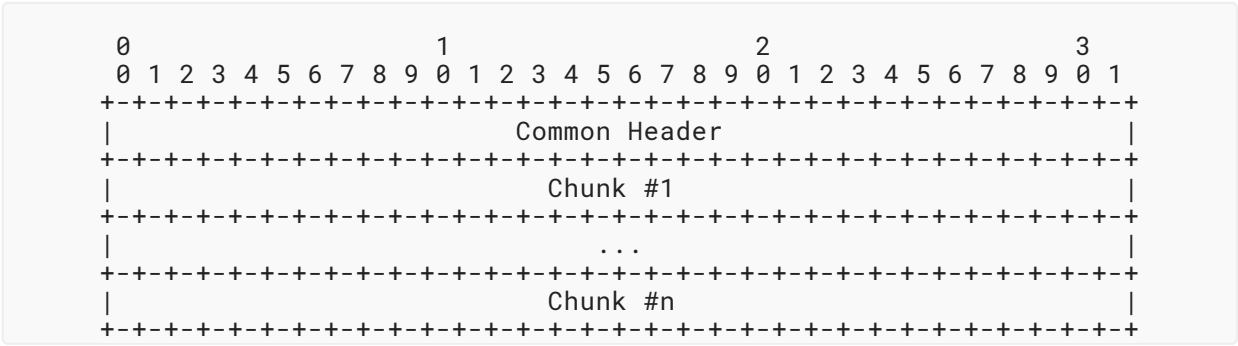
2. Conventions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. SCTP Packet Format

An SCTP packet is composed of a common header and chunks. A chunk contains either control information or user data.

The SCTP packet format is shown below:

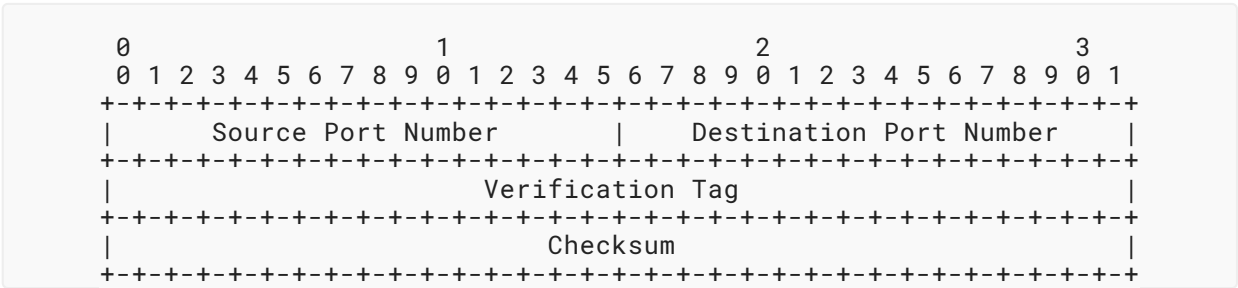


INIT, INIT ACK, and SHUTDOWN COMPLETE chunks **MUST NOT** be bundled with any other chunk into an SCTP packet. All other chunks **MAY** be bundled to form an SCTP packet that does not exceed the PMTU. See Section 6.10 for more details on chunk bundling.

If a user data message does not fit into one SCTP packet, it can be fragmented into multiple chunks using the procedure defined in Section 6.9.

All integer fields in an SCTP packet **MUST** be transmitted in network byte order, unless otherwise stated.

3.1. SCTP Common Header Field Descriptions



Source Port Number: 16 bits (unsigned integer)
: This is the SCTP sender's port number. It can be used by the receiver in combination with the source IP address, the SCTP Destination Port Number, and possibly the destination IP address to identify the association to which this packet belongs. The Source Port Number 0 **MUST NOT** be used.

Destination Port Number: 16 bits (unsigned integer)
: This is the SCTP port number to which this packet is destined. The receiving host will use this port number to de-multiplex the SCTP packet to the correct receiving endpoint/application. The Destination Port Number 0 **MUST NOT** be used.

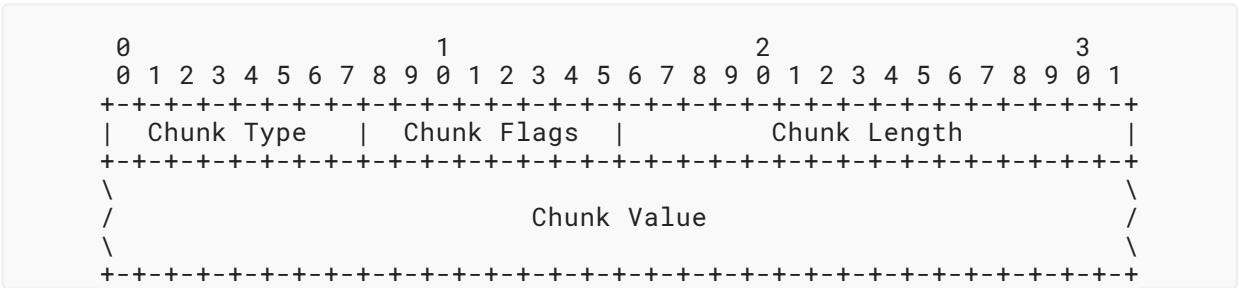
Verification Tag: 32 bits (unsigned integer)
: The receiver of an SCTP packet uses the Verification Tag to validate the sender of this packet. On transmit, the value of the Verification Tag **MUST** be set to the value of the Initiate Tag received from the peer endpoint during the association initialization, with the following exceptions:

- A packet containing an INIT chunk **MUST** have a zero Verification Tag.
- A packet containing a SHUTDOWN COMPLETE chunk with the T bit set **MUST** have the Verification Tag copied from the packet with the SHUTDOWN ACK chunk.
- A packet containing an ABORT chunk **MAY** have the Verification Tag copied from the packet that caused the ABORT chunk to be sent. For details, see Sections 8.4 and 8.5.

Checksum: 32 bits (unsigned integer)
: This field contains the checksum of the SCTP packet. Its calculation is discussed in Section 6.8. SCTP uses the CRC32c algorithm as described in Appendix A for calculating the checksum.

3.2. Chunk Field Descriptions

The figure below illustrates the field format for the chunks to be transmitted in the SCTP packet. Each chunk is formatted with a Chunk Type field, a Chunk Flags field, a Chunk Length field, and a Chunk Value field.



Chunk Type: 8 bits (unsigned integer)
: This field identifies the type of information contained in the Chunk Value field. It takes a value from 0 to 254. The value of 255 is reserved for future use as an extension field.

The values of Chunk Types defined in this document are as follows:

ID Value	Chunk Type
0	Payload Data (DATA)
1	Initiation (INIT)

ID Value	Chunk Type
2	Initiation Acknowledgement (INIT ACK)
3	Selective Acknowledgement (SACK)
4	Heartbeat Request (HEARTBEAT)
5	Heartbeat Acknowledgement (HEARTBEAT ACK)
6	Abort (ABORT)
7	Shutdown (SHUTDOWN)
8	Shutdown Acknowledgement (SHUTDOWN ACK)
9	Operation Error (ERROR)
10	State Cookie (COOKIE ECHO)
11	Cookie Acknowledgement (COOKIE ACK)
12	Reserved for Explicit Congestion Notification Echo (ECNE)
13	Reserved for Congestion Window Reduced (CWR)
14	Shutdown Complete (SHUTDOWN COMPLETE)
15 to 62	Unassigned
63	Reserved for IETF-defined Chunk Extensions
64 to 126	Unassigned
127	Reserved for IETF-defined Chunk Extensions
128 to 190	Unassigned
191	Reserved for IETF-defined Chunk Extensions
192 to 254	Unassigned
255	Reserved for IETF-defined Chunk Extensions

Table 1: Chunk Types

Note: The ECNE and CWR chunk types are reserved for future use of Explicit Congestion Notification (ECN).

Chunk Types are encoded such that the highest-order 2 bits specify the action that is taken if the processing endpoint does not recognize the Chunk Type.

00	Stop processing this SCTP packet and discard the unrecognized chunk and all further chunks.
01	Stop processing this SCTP packet, discard the unrecognized chunk and all further chunks, and report the unrecognized chunk in an ERROR chunk using the 'Unrecognized Chunk Type' error cause.
10	Skip this chunk and continue processing.
11	Skip this chunk and continue processing, but report it in an ERROR chunk using the 'Unrecognized Chunk Type' error cause.

Table 2: Processing of Unknown Chunks

Chunk Flags: 8 bits
: The usage of these bits depends on the Chunk Type, as given by the Chunk Type field. Unless otherwise specified, they are set to 0 on transmit and are ignored on receipt.

Chunk Length: 16 bits (unsigned integer)
: This value represents the size of the chunk in bytes, including the Chunk Type, Chunk Flags, Chunk Length, and Chunk Value fields. Therefore, if the Chunk Value field is zero-length, the Length field will be set to 4. The Chunk Length field does not count any chunk padding. However, it does include any padding of variable-length parameters other than the last parameter in the chunk.

  Note: A robust implementation is expected to accept the chunk whether or not the final padding has been included in the Chunk Length.

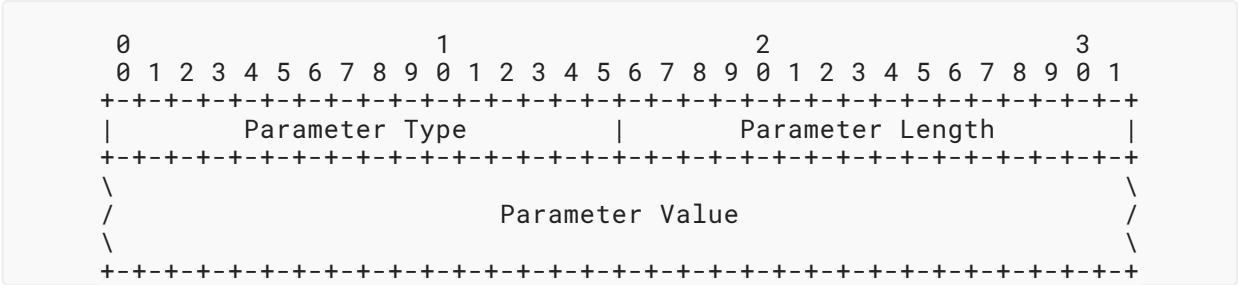
Chunk Value: variable length
: The Chunk Value field contains the actual information to be transferred in the chunk. The usage and format of this field is dependent on the Chunk Type.

The total length of a chunk (including Type, Length, and Value fields) **MUST** be a multiple of 4 bytes. If the length of the chunk is not a multiple of 4 bytes, the sender **MUST** pad the chunk with all zero bytes, and this padding is not included in the Chunk Length field. The sender **MUST NOT** pad with more than 3 bytes. The receiver **MUST** ignore the padding bytes.

SCTP-defined chunks are described in detail in Section 3.3. The guidelines for IETF-defined chunk extensions can be found in Section 15.1 of this document.

3.2.1. Optional/Variable-Length Parameter Format

Chunk values of SCTP control chunks consist of a chunk-type-specific header of required fields, followed by zero or more parameters. The optional and variable-length parameters contained in a chunk are defined in a Type-Length-Value format, as shown below.



Parameter Type: 16 bits (unsigned integer)
: The Type field is a 16-bit identifier of the type of parameter. It takes a value of 0 to 65534.

  The value of 65535 is reserved for IETF-defined extensions. Values other than those defined in specific SCTP chunk descriptions are reserved for use by IETF.

Parameter Length: 16 bits (unsigned integer)
: The Parameter Length field contains the size of the parameter in bytes, including the Parameter Type, Parameter Length, and Parameter Value fields. Thus, a parameter with a zero-length Parameter Value field would have a Parameter Length field of 4. The Parameter Length does not include any padding bytes.

Parameter Value: variable length
: The Parameter Value field contains the actual information to be transferred in the parameter.

The total length of a parameter (including Parameter Type, Parameter Length, and Parameter Value fields) **MUST** be a multiple of 4 bytes. If the length of the parameter is not a multiple of 4 bytes, the sender pads the parameter at the end (i.e., after the Parameter Value field) with all zero bytes. The length of the padding is not included in the Parameter Length field. A sender **MUST NOT** pad with more than 3 bytes. The receiver **MUST** ignore the padding bytes.

The Parameter Types are encoded such that the highest-order 2 bits specify the action that is taken if the processing endpoint does not recognize the Parameter Type.

00	Stop processing this parameter and do not process any further parameters within this chunk.
01	Stop processing this parameter, do not process any further parameters within this chunk, and report the unrecognized parameter, as described in Section 3.2.2.
10	Skip this parameter and continue processing.
11	Skip this parameter and continue processing, but report the unrecognized parameter, as described in Section 3.2.2.

Table 3: Processing of Unknown Parameters

Please note that, when an INIT or INIT ACK chunk is received, in all four cases, an INIT ACK or COOKIE ECHO chunk is sent in response, respectively. In the 00 or 01 case, the processing of the parameters after the unknown parameter is canceled, but no processing already done is rolled back.

The actual SCTP parameters are defined in the specific SCTP chunk sections. The rules for IETF-defined parameter extensions are defined in Section 15.3. Parameter types **MUST** be unique across all chunks. For example, the parameter type '5' is used to represent an IPv4 address (see Section 3.3.2.1.1). The value '5' then is reserved across all chunks to represent an IPv4 address and **MUST NOT** be reused with a different meaning in any other chunk.

3.2.2. Reporting of Unrecognized Parameters

If the receiver of an INIT chunk detects unrecognized parameters and has to report them according to Section 3.2.1, it **MUST** put the "Unrecognized Parameter" parameter(s) in the INIT ACK chunk sent in response to the INIT chunk. Note that, if the receiver of the INIT chunk is not going to establish an association (e.g., due to lack of resources), an "Unrecognized Parameters" error cause would not be included with any ABORT chunk being sent to the sender of the INIT chunk.

If the receiver of any other chunk (e.g., INIT ACK) detects unrecognized parameters and has to report them according to Section 3.2.1, it **SHOULD** bundle the ERROR chunk containing the "Unrecognized Parameters" error cause with the chunk sent in response (e.g., COOKIE ECHO). If the receiver of an INIT ACK chunk cannot bundle the COOKIE ECHO chunk with the ERROR chunk, the ERROR chunk **MAY** be sent separately but not before the COOKIE ACK chunk has been received.

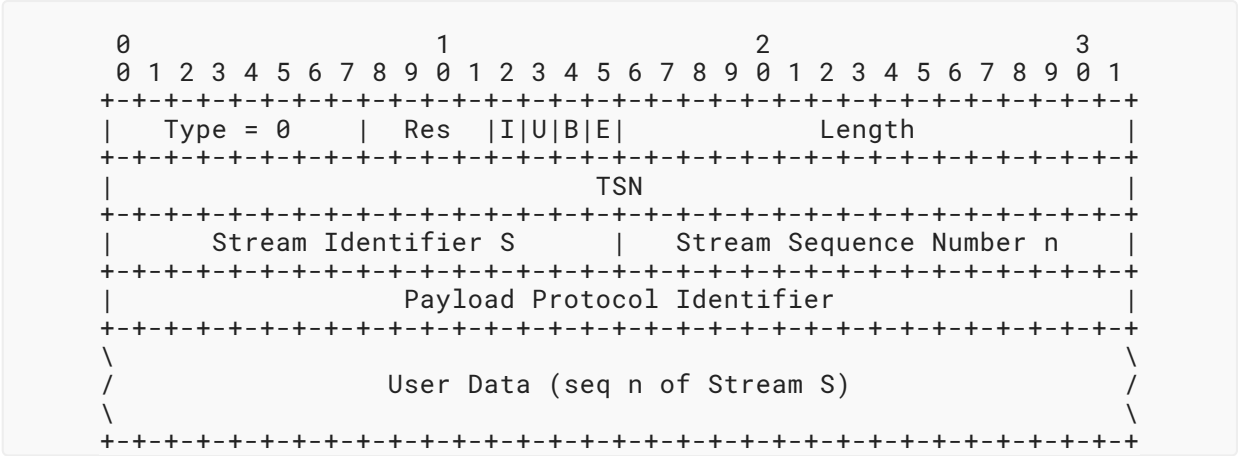
Any time a COOKIE ECHO chunk is sent in a packet, it **MUST** be the first chunk.

3.3. SCTP Chunk Definitions

This section defines the format of the different SCTP chunk types.

3.3.1. Payload Data (DATA) (0)

The following format **MUST** be used for the DATA chunk:



Res: 4 bits
: All set to 0 on transmit and ignored on receipt.

I bit: 1 bit
: The (I)mmediate bit **MAY** be set by the sender whenever the sender of a DATA chunk can benefit from the corresponding SACK chunk being sent back without delay. See Section 4 of [RFC7053] for a discussion of the benefits.

U bit: 1 bit
: The (U)nordered bit, if set to 1, indicates that this is an unordered DATA chunk, and there is no Stream Sequence Number assigned to this DATA chunk. Therefore, the receiver **MUST** ignore the Stream Sequence Number field.

  After reassembly (if necessary), unordered DATA chunks **MUST** be dispatched to the upper layer by the receiver without any attempt to reorder.

  If an unordered user message is fragmented, each fragment of the message **MUST** have its U bit set to 1.

B bit: 1 bit
: The (B)eginning fragment bit, if set, indicates the first fragment of a user message.

E bit: 1 bit
: The (E)nding fragment bit, if set, indicates the last fragment of a user message.

Length: 16 bits (unsigned integer)
: This field indicates the length of the DATA chunk in bytes from the beginning of the type field to the end of the User Data field excluding any padding. A DATA chunk with one byte of user data will have the Length field set to 17 (indicating 17 bytes).

  A DATA chunk with a User Data field of length L will have the Length field set to (16 + L) (indicating 16 + L bytes) where L **MUST** be greater than 0.

TSN: 32 bits (unsigned integer)
: This value represents the TSN for this DATA chunk. The valid range of TSN is from 0 to 4294967295 (2^{32} - 1). TSN wraps back to 0 after reaching 4294967295.

Stream Identifier S: 16 bits (unsigned integer)
: Identifies the stream to which the following user data belongs.

Stream Sequence Number n: 16 bits (unsigned integer)
: This value represents the Stream Sequence Number of the following user data within the stream S. Valid range is 0 to 65535.

  When a user message is fragmented by SCTP for transport, the same Stream Sequence Number **MUST** be carried in each of the fragments of the message.

Payload Protocol Identifier: 32 bits (unsigned integer)
: This value represents an application (or upper layer) specified protocol identifier. This value is passed to SCTP by its upper layer and sent to its peer. This identifier is not used by SCTP but can be used by certain network entities, as well as by the peer application, to identify the type of information being carried in this DATA chunk. This field **MUST** be sent even in fragmented DATA chunks (to make sure it is available for agents in the middle of the network). Note that this field is not touched by an SCTP implementation; the upper layer is responsible for the host to network byte order conversion of this field.

  The value 0 indicates that no application identifier is specified by the upper layer for this payload data.

User Data: variable length
: This is the payload user data. The implementation **MUST** pad the end of the data to a 4-byte boundary with all zero bytes. Any padding **MUST NOT** be included in the Length field. A sender **MUST** never add more than 3 bytes of padding.

An unfragmented user message **MUST** have both the B and E bits set to 1. Setting both B and E bits to 0 indicates a middle fragment of a multi-fragment user message, as summarized in the following table:

B	E	Description
1	0	First piece of a fragmented user message
0	0	Middle piece of a fragmented user message
0	1	Last piece of a fragmented user message
1	1	Unfragmented message

Table 4: Fragment Description Flags

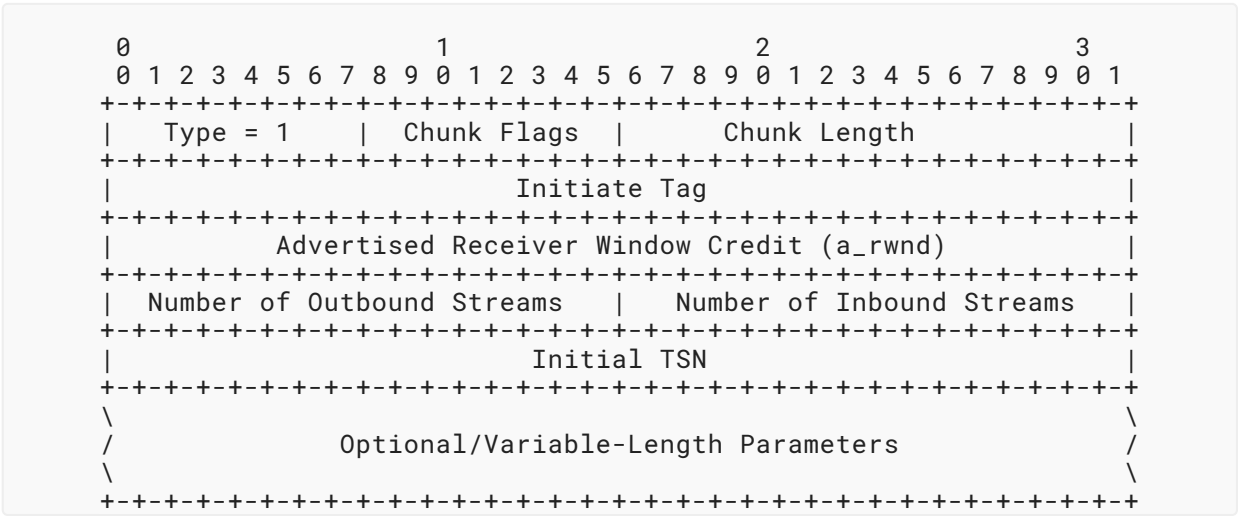
When a user message is fragmented into multiple chunks, the TSNs are used by the receiver to reassemble the message. This means that the TSNs for each fragment of a fragmented user message **MUST** be strictly sequential.

The TSNs of DATA chunks sent **SHOULD** be strictly sequential.

Note: The extension described in [RFC8260] can be used to mitigate the head of line blocking when transferring large user messages.

3.3.2. Initiation (INIT) (1)

This chunk is used to initiate an SCTP association between two endpoints. The format of the INIT chunk is shown below:



The following parameters are specified for the INIT chunk. Unless otherwise noted, each parameter **MUST** only be included once in the INIT chunk.

Fixed-Length Parameter	Status
Initiate Tag	Mandatory
Advertised Receiver Window Credit	Mandatory
Number of Outbound Streams	Mandatory
Number of Inbound Streams	Mandatory
Initial TSN	Mandatory

Table 5: Fixed-Length Parameters of INIT Chunks

Variable-Length Parameter	Status	Type Value
IPv4 Address (Note 1)	Optional	5

Variable-Length Parameter	Status	Type Value
IPv6 Address (Note 1)	Optional	6
Cookie Preservative	Optional	9
Reserved for ECN Capable (Note 2)	Optional	32768 (0x8000)
Host Name Address (Note 3)	Deprecated	11
Supported Address Types (Note 4)	Optional	12

Table 6: Variable-Length Parameters of INIT Chunks

Note 1: The INIT chunks can contain multiple addresses that can be IPv4 and/or IPv6 in any combination.

Note 2: The ECN Capable field is reserved for future use of Explicit Congestion Notification.

Note 3: An INIT chunk **MUST NOT** contain the Host Name Address parameter. The receiver of an INIT chunk containing a Host Name Address parameter **MUST** send an ABORT chunk and **MAY** include an "Unresolvable Address" error cause.

Note 4: This parameter, when present, specifies all the address types the sending endpoint can support. The absence of this parameter indicates that the sending endpoint can support any address type.

If an INIT chunk is received with all mandatory parameters that are specified for the INIT chunk, then the receiver **SHOULD** process the INIT chunk and send back an INIT ACK. The receiver of the INIT chunk **MAY** bundle an ERROR chunk with the COOKIE ACK chunk later. However, restrictive implementations **MAY** send back an ABORT chunk in response to the INIT chunk.

The Chunk Flags field in INIT chunks is reserved, and all bits in it **SHOULD** be set to 0 by the sender and ignored by the receiver.

Initiate Tag: 32 bits (unsigned integer)
: The receiver of the INIT chunk (the responding end) records the value of the Initiate Tag parameter. This value **MUST** be placed into the Verification Tag field of every SCTP packet that the receiver of the INIT chunk transmits within this association.

  The Initiate Tag is allowed to have any value except 0. See Section 5.3.1 for more on the selection of the tag value.

  If the value of the Initiate Tag in a received INIT chunk is found to be 0, the receiver **MUST** silently discard the packet.

Advertised Receiver Window Credit (a_rwnd): 32 bits (unsigned integer)
: This value represents the dedicated buffer space, in number of bytes, the sender of the INIT chunk has reserved in association with this window.

The Advertised Receiver Window Credit **MUST NOT** be smaller than 1500.

A receiver of an INIT chunk with the a_rwnd value set to a value smaller than 1500 **MUST** discard the packet, **SHOULD** send a packet in response containing an ABORT chunk and using the Initiate Tag as the Verification Tag, and **MUST NOT** change the state of any existing association.

During the life of the association, this buffer space **SHOULD NOT** be reduced (i.e., dedicated buffers ought not to be taken away from this association); however, an endpoint **MAY** change the value of a_rwnd it sends in SACK chunks.

Number of Outbound Streams (OS): 16 bits (unsigned integer)
Defines the number of outbound streams the sender of this INIT chunk wishes to create in this association. The value of 0 **MUST NOT** be used.

A receiver of an INIT chunk with the OS value set to 0 **MUST** discard the packet, **SHOULD** send a packet in response containing an ABORT chunk and using the Initiate Tag as the Verification Tag, and **MUST NOT** change the state of any existing association.

Number of Inbound Streams (MIS): 16 bits (unsigned integer)
Defines the maximum number of streams the sender of this INIT chunk allows the peer end to create in this association. The value 0 **MUST NOT** be used.

Note: There is no negotiation of the actual number of streams; instead, the two endpoints will use the min(requested, offered). See Section 5.1.1 for details.

A receiver of an INIT chunk with the MIS value set to 0 **MUST** discard the packet, **SHOULD** send a packet in response containing an ABORT chunk and using the Initiate Tag as the Verification Tag, and **MUST NOT** change the state of any existing association.

Initial TSN (I-TSN): 32 bits (unsigned integer)
Defines the TSN that the sender of the INIT chunk will use initially. The valid range is from 0 to 4294967295 and the Initial TSN **SHOULD** be set to a random value in that range. The methods described in [RFC4086] can be used for the Initial TSN randomization.

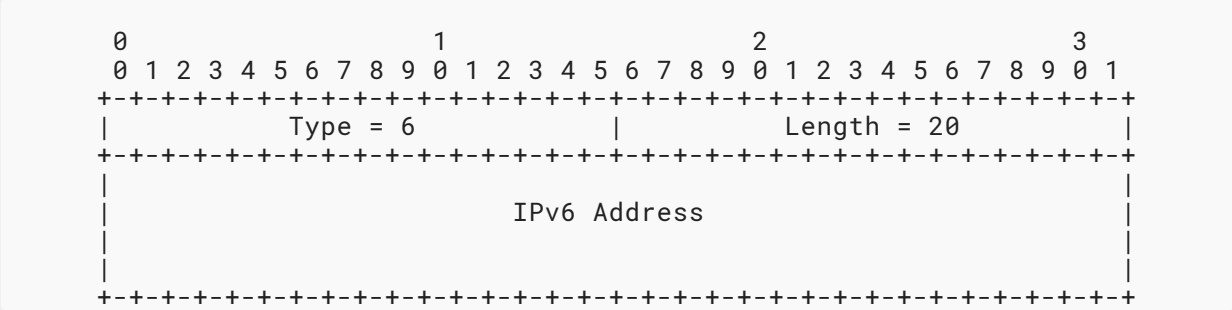
3.3.2.1. Optional or Variable-Length Parameters in INIT chunks

The following parameters follow the Type-Length-Value format as defined in Section 3.2.1. Any Type-Length-Value fields **MUST** be placed after the fixed-length fields. (The fixed-length fields are defined in the previous section.)

3.3.2.1.1. IPv4 Address (5)

IPv4 Address: 32 bits (unsigned integer)
Contains an IPv4 address of the sending endpoint. It is binary encoded.

3.3.2.1.2. IPv6 Address (6)



IPv6 Address: 128 bits (unsigned integer)
Contains an IPv6 [RFC8200] address of the sending endpoint. It is binary encoded.

A sender **MUST NOT** use an IPv4-mapped IPv6 address [RFC4291] but **SHOULD** instead use an IPv4 Address parameter for an IPv4 address.

Combined with the Source Port Number in the SCTP common header, the value passed in an IPv4 or IPv6 Address parameter indicates a transport address the sender of the INIT chunk will support for the association being initiated. That is, during the life time of this association, this IP address can appear in the source address field of an IP datagram sent from the sender of the INIT chunk and can be used as a destination address of an IP datagram sent from the receiver of the INIT chunk.

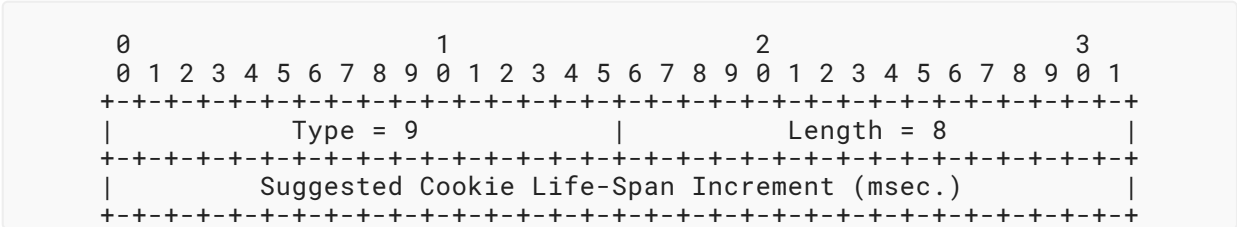
More than one IP Address parameter can be included in an INIT chunk when the sender of the INIT chunk is multi-homed. Moreover, a multi-homed endpoint might have access to different types of network; thus, more than one address type can be present in one INIT chunk, i.e., IPv4 and IPv6 addresses are allowed in the same INIT chunk.

If the INIT chunk contains at least one IP Address parameter, then the source address of the IP datagram containing the INIT chunk and any additional address(es) provided within the INIT can be used as destinations by the endpoint receiving the INIT chunk. If the INIT chunk does not contain any IP Address parameters, the endpoint receiving the INIT chunk **MUST** use the source address associated with the received IP datagram as its sole destination address for the association.

Note that not using any IP Address parameters in the INIT and INIT ACK chunk is a way to make an association more likely to work in combination with Network Address Translation (NAT).

3.3.2.1.3. Cookie Preservative (9)

The sender of the INIT chunk uses this parameter to suggest to the receiver of the INIT chunk a longer life span for the State Cookie.



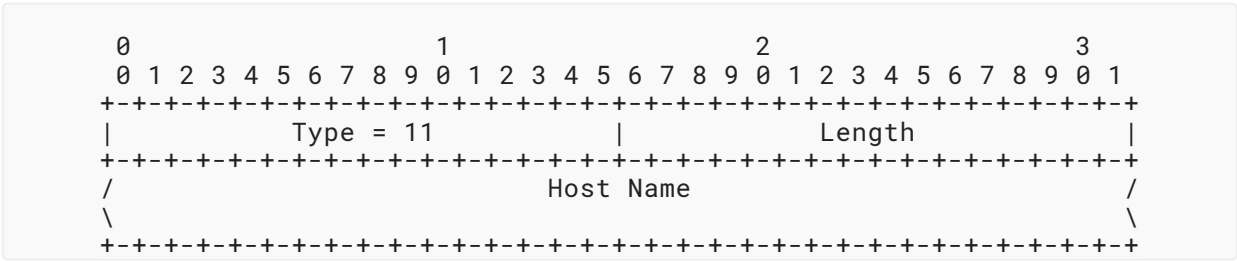
Suggested Cookie Life-Span Increment: 32 bits (unsigned integer)

This parameter indicates to the receiver how much increment in milliseconds the sender wishes the receiver to add to its default cookie life span.

This optional parameter **MAY** be added to the INIT chunk by the sender when it reattempts establishing an association with a peer to which its previous attempt of establishing the association failed due to a stale cookie operation error. The receiver **MAY** choose to ignore the suggested cookie life span increase for its own security reasons.

3.3.2.1.4. Host Name Address (11)

The sender of an INIT chunk or INIT ACK chunk **MUST NOT** include this parameter. The usage of the Host Name Address parameter is deprecated. The receiver of an INIT chunk or an INIT ACK containing a Host Name Address parameter **MUST** send an ABORT chunk and **MAY** include an "Unresolvable Address" error cause.

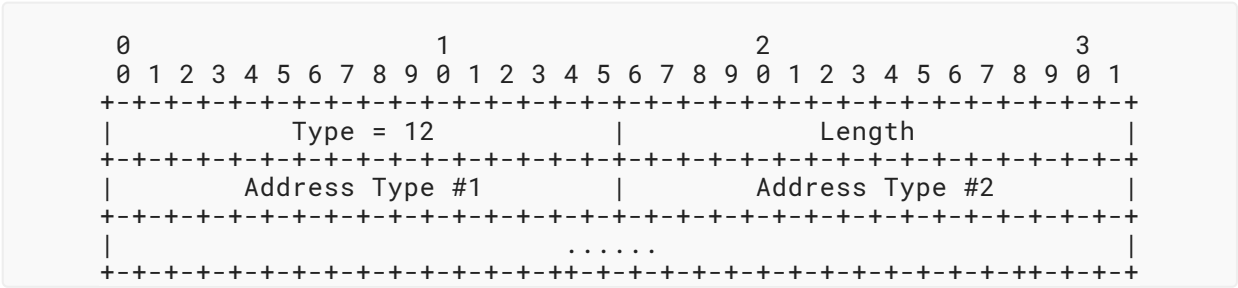


Host Name: variable length
: This field contains a host name in "host name syntax" per Section 2.1 of [RFC1123]. The method for resolving the host name is out of scope of SCTP.

  At least one null terminator is included in the Host Name string and **MUST** be included in the length.

3.3.2.1.5. Supported Address Types (12)

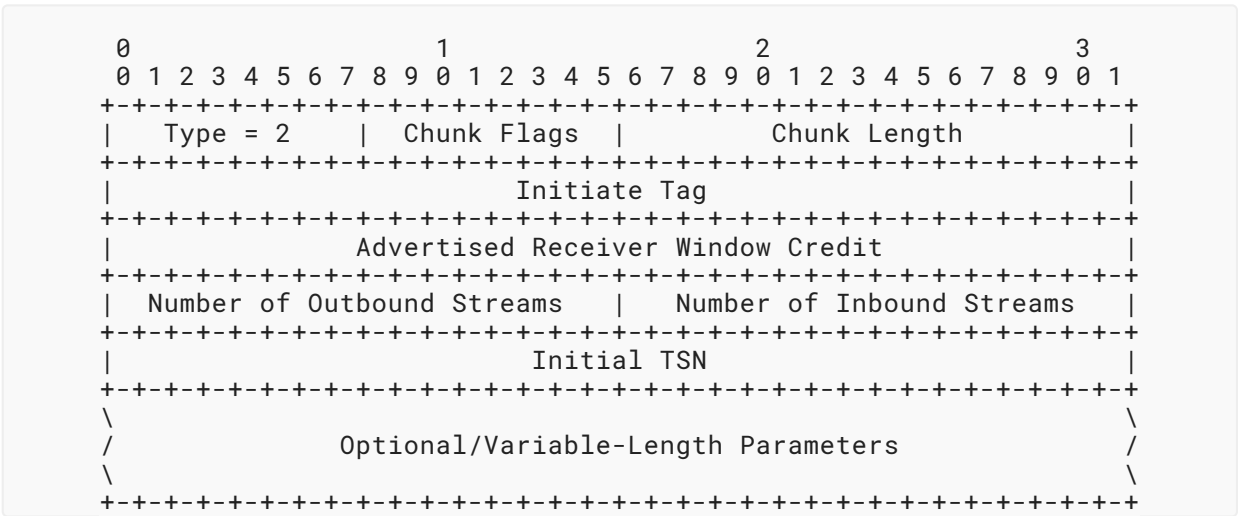
The sender of the INIT chunk uses this parameter to list all the address types it can support.



Address Type: 16 bits (unsigned integer)
: This is filled with the type value of the corresponding address TLV (e.g., 5 for indicating IPv4, and 6 for indicating IPv6). The value indicating the Host Name Address parameter **MUST NOT** be used when sending this parameter and **MUST** be ignored when receiving this parameter.

3.3.3. Initiation Acknowledgement (INIT ACK) (2)

The INIT ACK chunk is used to acknowledge the initiation of an SCTP association. The format of the INIT ACK chunk is shown below:



The parameter part of INIT ACK is formatted similarly to the INIT chunk. The following parameters are specified for the INIT ACK chunk:

Fixed-Length Parameter	Status
Initiate Tag	Mandatory
Advertised Receiver Window Credit	Mandatory
Number of Outbound Streams	Mandatory
Number of Inbound Streams	Mandatory
Initial TSN	Mandatory

Table 7: Fixed-Length Parameters of INIT ACK Chunks

It uses two extra variable parameters: the State Cookie and the Unrecognized Parameter.

Variable-Length Parameter	Status	Type Value
State Cookie	Mandatory	7
IPv4 Address (Note 1)	Optional	5
IPv6 Address (Note 1)	Optional	6
Unrecognized Parameter	Optional	8
Reserved for ECN Capable (Note 2)	Optional	32768 (0x8000)
Host Name Address (Note 3)	Deprecated	11

Table 8: Variable-Length Parameters of INIT ACK Chunks

Note 1: The INIT ACK chunks can contain any number of IP Address parameters that can be IPv4 and/or IPv6 in any combination.

Note 2: The ECN Capable field is reserved for future use of Explicit Congestion Notification.

Note 3: An INIT ACK chunk **MUST NOT** contain the Host Name Address parameter. The receiver of INIT ACK chunks containing a Host Name Address parameter **MUST** send an ABORT chunk and **MAY** include an "Unresolvable Address" error cause.

The Chunk Flags field in INIT ACK chunks is reserved, and all bits in it **SHOULD** be set to 0 by the sender and ignored by the receiver.

Initiate Tag: 32 bits (unsigned integer)
: The receiver of the INIT ACK chunk records the value of the Initiate Tag parameter. This value **MUST** be placed into the Verification Tag field of every SCTP packet that the receiver of the INIT ACK chunk transmits within this association.

The Initiate Tag **MUST NOT** take the value 0. See Section 5.3.1 for more on the selection of the Initiate Tag value.

If an endpoint in the COOKIE-WAIT state receives an INIT ACK chunk with the Initiate Tag set to 0, it **MUST** destroy the TCB and **SHOULD** send an ABORT chunk with the T bit set. If such an INIT ACK chunk is received in any state other than CLOSED or COOKIE-WAIT, it **SHOULD** be discarded silently (see Section 5.2.3).

Advertised Receiver Window Credit (a_rwnd): 32 bits (unsigned integer)
This value represents the dedicated buffer space, in number of bytes, the sender of the INIT ACK chunk has reserved in association with this window.

The Advertised Receiver Window Credit **MUST NOT** be smaller than 1500.

A receiver of an INIT ACK chunk with the a_rwnd value set to a value smaller than 1500 **MUST** discard the packet, **SHOULD** send a packet in response containing an ABORT chunk and using the Initiate Tag as the Verification Tag, and **MUST NOT** change the state of any existing association.

During the life of the association, this buffer space **SHOULD NOT** be reduced (i.e., dedicated buffers ought not to be taken away from this association); however, an endpoint **MAY** change the value of a_rwnd it sends in SACK chunks.

Number of Outbound Streams (OS): 16 bits (unsigned integer)
Defines the number of outbound streams the sender of this INIT ACK chunk wishes to create in this association. The value of 0 **MUST NOT** be used, and the value **MUST NOT** be greater than the MIS value sent in the INIT chunk.

If an endpoint in the COOKIE-WAIT state receives an INIT ACK chunk with the OS value set to 0, it **MUST** destroy the TCB and **SHOULD** send an ABORT chunk. If such an INIT ACK chunk is received in any state other than CLOSED or COOKIE-WAIT, it **SHOULD** be discarded silently (see Section 5.2.3).

Number of Inbound Streams (MIS): 16 bits (unsigned integer)
Defines the maximum number of streams the sender of this INIT ACK chunk allows the peer end to create in this association. The value 0 **MUST NOT** be used.

Note: There is no negotiation of the actual number of streams, but instead the two endpoints will use the min(requested, offered). See Section 5.1.1 for details.

If an endpoint in the COOKIE-WAIT state receives an INIT ACK chunk with the MIS value set to 0, it **MUST** destroy the TCB and **SHOULD** send an ABORT chunk. If such an INIT ACK chunk is received in any state other than CLOSED or COOKIE-WAIT, it **SHOULD** be discarded silently (see Section 5.2.3).

Initial TSN (I-TSN): 32 bits (unsigned integer)
: Defines the TSN that the sender of the INIT ACK chunk will use initially. The valid range is from 0 to 4294967295 and the Initial TSN **SHOULD** be set to a random value in that range. The methods described in [RFC4086] can be used for the Initial TSN randomization.

Implementation Note: An implementation **MUST** be prepared to receive an INIT ACK chunk that is quite large (more than 1500 bytes) due to the variable size of the State Cookie and the variable address list. For example, if a responder to the INIT chunk has 1000 IPv4 addresses it wishes to send, it would need at least 8,000 bytes to encode this in the INIT ACK chunk.

If an INIT ACK chunk is received with all mandatory parameters that are specified for the INIT ACK chunk, then the receiver **SHOULD** process the INIT ACK chunk and send back a COOKIE ECHO chunk. The receiver of the INIT ACK chunk **MAY** bundle an ERROR chunk with the COOKIE ECHO chunk. However, restrictive implementations **MAY** send back an ABORT chunk in response to the INIT ACK chunk.

In combination with the Source Port Number carried in the SCTP common header, each IP Address parameter in the INIT ACK chunk indicates to the receiver of the INIT ACK chunk a valid transport address supported by the sender of the INIT ACK chunk for the life time of the association being initiated.

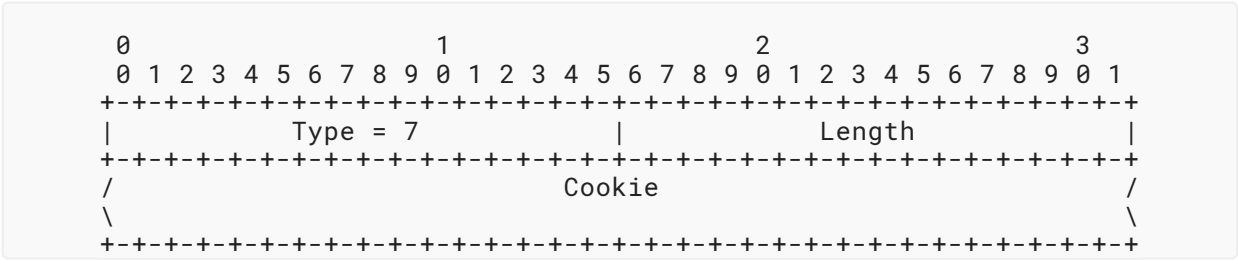
If the INIT ACK chunk contains at least one IP Address parameter, then the source address of the IP datagram containing the INIT ACK chunk and any additional address(es) provided within the INIT ACK chunk **MAY** be used as destinations by the receiver of the INIT ACK chunk. If the INIT ACK chunk does not contain any IP Address parameters, the receiver of the INIT ACK chunk **MUST** use the source address associated with the received IP datagram as its sole destination address for the association.

The State Cookie and Unrecognized Parameters use the Type-Length-Value format as defined in Section 3.2.1 and are described below. The other fields are defined in the same way as their counterparts in the INIT chunk.

3.3.3.1. Optional or Variable-Length Parameters in INIT ACK Chunks

The State Cookie and Unrecognized Parameters use the Type-Length-Value format, as defined in Section 3.2.1, and are described below. The IPv4 Address parameter is described in Section 3.3.2.1.1, and the IPv6 Address parameter is described in Section 3.3.2.1.2. The Host Name Address parameter is described in Section 3.3.2.1.4 and **MUST NOT** be included in an INIT ACK chunk. Any Type-Length-Value fields **MUST** be placed after the fixed-length fields. (The fixed-length fields are defined in the previous section.)

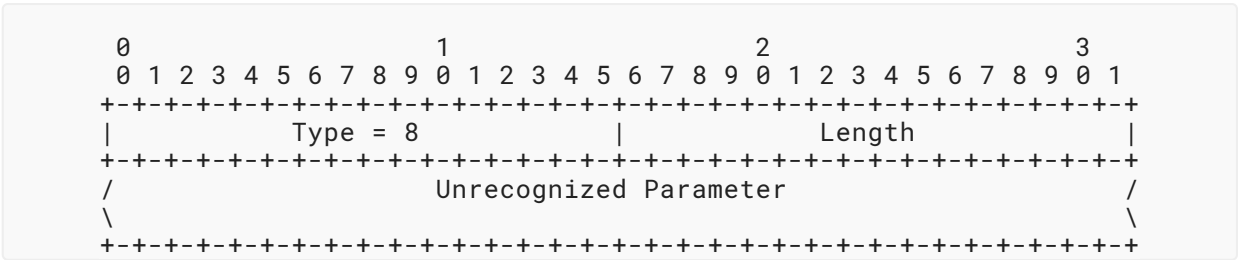
3.3.3.1.1. State Cookie (7)



Cookie: variable length
: This parameter value **MUST** contain all the necessary state and parameter information required for the sender of this INIT ACK chunk to create the association, along with a Message Authentication Code (MAC). See Section 5.1.3 for details on State Cookie definition.

3.3.3.1.2. Unrecognized Parameter (8)

This parameter is returned to the originator of the INIT chunk when the INIT chunk contains an unrecognized parameter that has a type that indicates it **SHOULD** be reported to the sender.



Unrecognized Parameter: variable length
: The Parameter Value field will contain an unrecognized parameter copied from the INIT chunk complete with Parameter Type, Length, and Value fields.

3.3.4. Selective Acknowledgement (SACK) (3)

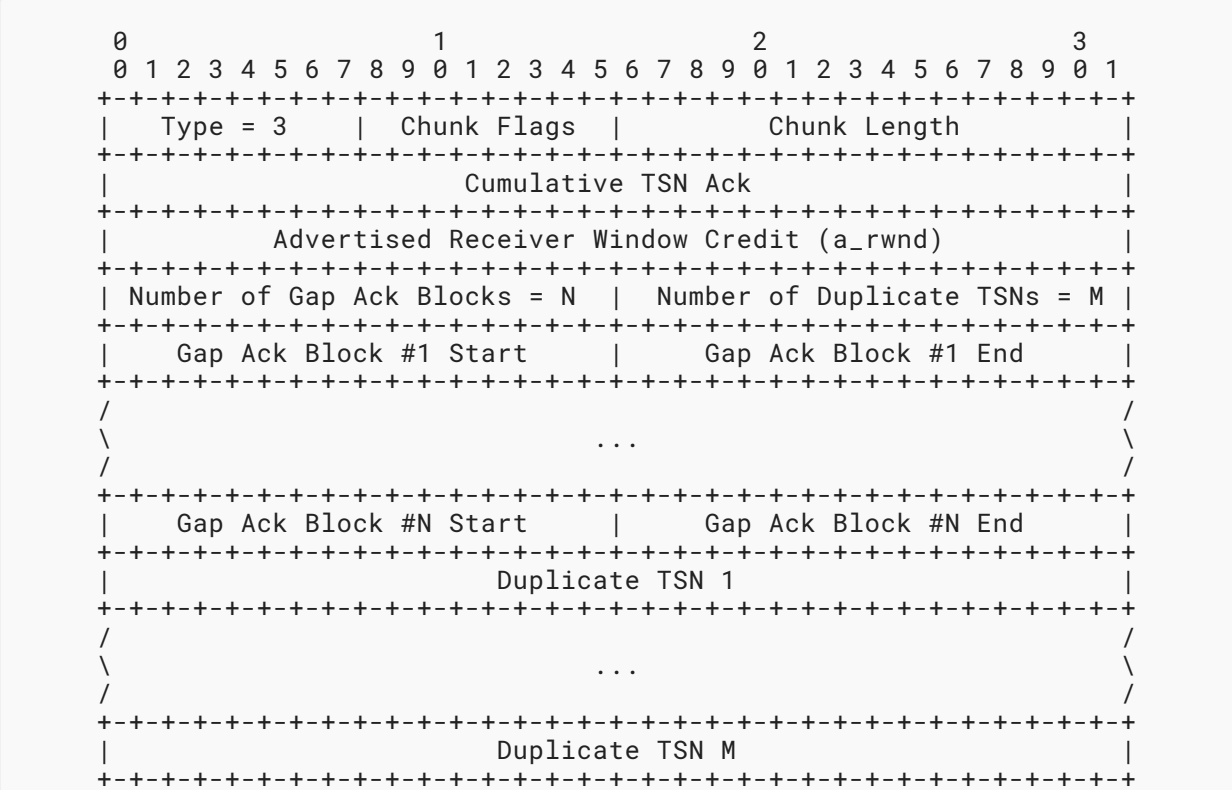
This chunk is sent to the peer endpoint to acknowledge received DATA chunks and to inform the peer endpoint of gaps in the received subsequences of DATA chunks as represented by their TSNs.

The SACK chunk **MUST** contain the Cumulative TSN Ack, Advertised Receiver Window Credit (a_rwnd), Number of Gap Ack Blocks, and Number of Duplicate TSNs fields.

By definition, the value of the Cumulative TSN Ack parameter is the last TSN received before a break in the sequence of received TSNs occurs; the next TSN value following this one has not yet been received at the endpoint sending the SACK chunk. This parameter therefore acknowledges receipt of all TSNs less than or equal to its value.

The handling of a_rwnd by the receiver of the SACK chunk is discussed in detail in Section 6.2.1.

The SACK chunk also contains zero or more Gap Ack Blocks. Each Gap Ack Block acknowledges a subsequence of TSNs received following a break in the sequence of received TSNs. The Gap Ack Blocks **SHOULD** be isolated. This means that the TSN just before each Gap Ack Block and the TSN just after each Gap Ack Block have not been received. By definition, all TSNs acknowledged by Gap Ack Blocks are greater than the value of the Cumulative TSN Ack.



Chunk Flags: 8 bits
: All set to 0 on transmit and ignored on receipt.

Cumulative TSN Ack: 32 bits (unsigned integer)
: The largest TSN, such that all TSNs smaller than or equal to it have been received and the next one has not been received. In the case where no DATA chunk has been received, this value is set to the peer's Initial TSN minus one.

Advertised Receiver Window Credit (a_rwnd): 32 bits (unsigned integer)
: This field indicates the updated receive buffer space in bytes of the sender of this SACK chunk; see Section 6.2.1 for details.

Number of Gap Ack Blocks: 16 bits (unsigned integer)
: Indicates the number of Gap Ack Blocks included in this SACK chunk.

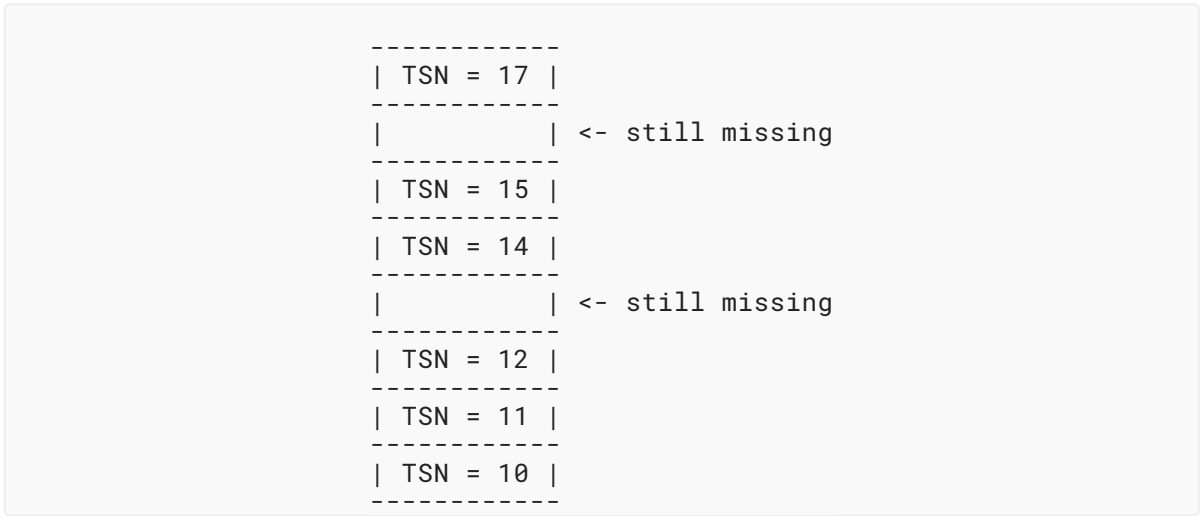
Number of Duplicate TSNs: 16 bit
: This field contains the number of duplicate TSNs the endpoint has received. Each duplicate TSN is listed following the Gap Ack Block list.

Gap Ack Blocks:
: These fields contain the Gap Ack Blocks. They are repeated for each Gap Ack Block up to the number of Gap Ack Blocks defined in the Number of Gap Ack Blocks field. All DATA chunks with TSNs greater than or equal to (Cumulative TSN Ack + Gap Ack Block Start) and less than or equal to (Cumulative TSN Ack + Gap Ack Block End) of each Gap Ack Block are assumed to have been received correctly.

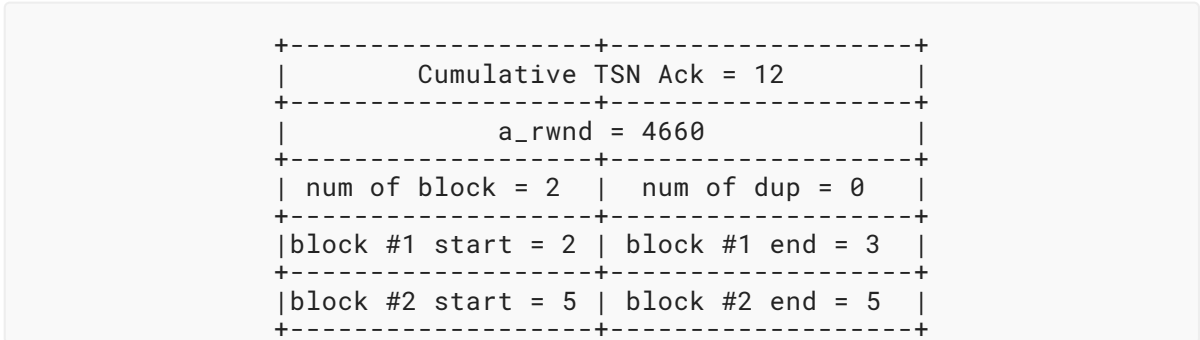
Gap Ack Block Start: 16 bits (unsigned integer)
: Indicates the Start offset TSN for this Gap Ack Block. To calculate the actual TSN number, the Cumulative TSN Ack is added to this offset number. This calculated TSN identifies the lowest TSN in this Gap Ack Block that has been received.

Gap Ack Block End: 16 bits (unsigned integer)
: Indicates the End offset TSN for this Gap Ack Block. To calculate the actual TSN number, the Cumulative TSN Ack is added to this offset number. This calculated TSN identifies the highest TSN in this Gap Ack Block that has been received.

For example, assume that the receiver has the following DATA chunks newly arrived at the time when it decides to send a Selective ACK:



Then, the parameter part of the SACK chunk **MUST** be constructed as follows (assuming the new a_rwnd is set to 4660 by the sender):



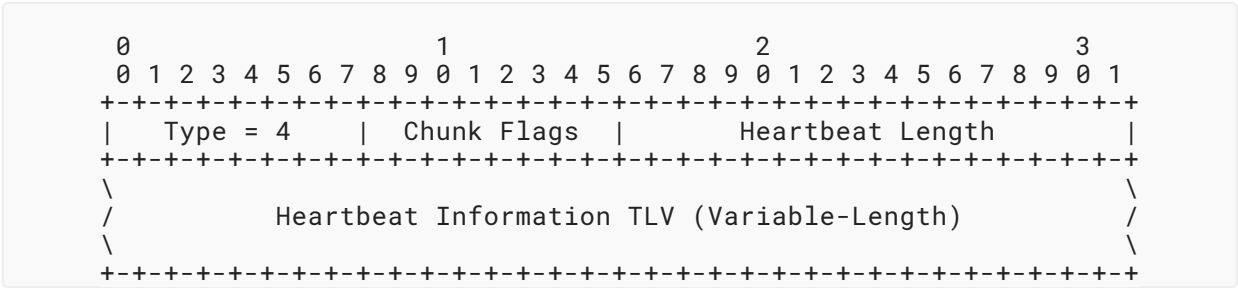
Duplicate TSN: 32 bits (unsigned integer)
: Indicates the number of times a TSN was received in duplicate since the last SACK chunk was sent. Every time a receiver gets a duplicate TSN (before sending the SACK chunk), it adds it to the list of duplicates. The duplicate count is reinitialized to zero after sending each SACK chunk.

  For example, if a receiver were to get the TSN 19 three times, it would list 19 twice in the outbound SACK chunk. After sending the SACK chunk, if it received yet one more TSN 19, it would list 19 as a duplicate once in the next outgoing SACK chunk.

3.3.5. Heartbeat Request (HEARTBEAT) (4)

An endpoint **SHOULD** send a HEARTBEAT (HB) chunk to its peer endpoint to probe the reachability of a particular destination transport address defined in the present association.

The parameter field contains the Heartbeat Information, which is a variable-length opaque data structure understood only by the sender.



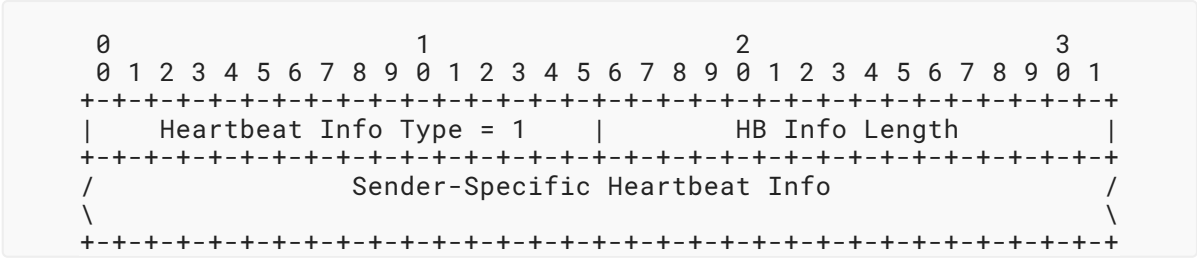
Chunk Flags: 8 bits
: Set to 0 on transmit and ignored on receipt.

Heartbeat Length: 16 bits (unsigned integer)
: Set to the size of the chunk in bytes, including the chunk header and the Heartbeat Information field.

Heartbeat Information: variable length
: Defined as a variable-length parameter using the format described in Section 3.2.1, that is:

Variable Parameters	Status	Type Value
Heartbeat Info	Mandatory	1

Table 9: Variable-Length Parameters of HEARTBEAT Chunks

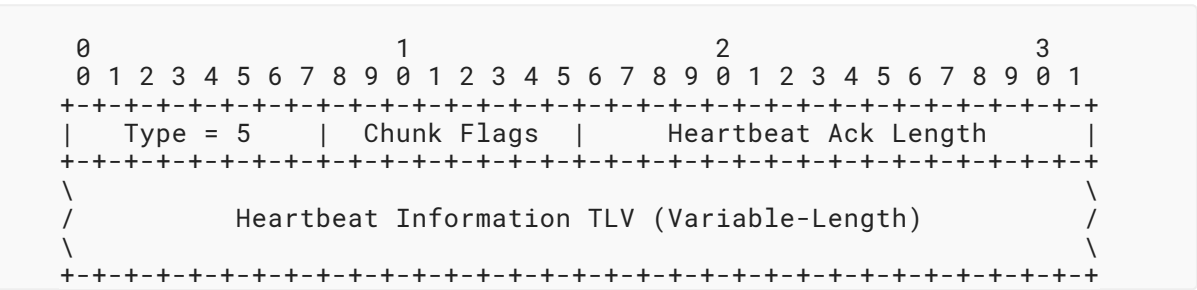


The Sender-Specific Heartbeat Info field **SHOULD** include information about the sender's current time when this HEARTBEAT chunk is sent and the destination transport address to which this HEARTBEAT chunk is sent (see Section 8.3). This information is simply reflected back by the receiver in the HEARTBEAT ACK chunk (see Section 3.3.6). Note also that the HEARTBEAT chunk is both for reachability checking and for path verification (see Section 5.4). When a HEARTBEAT chunk is being used for path verification purposes, it **MUST** include a random nonce of length 64 bits or longer ([RFC4086] provides some information on randomness guidelines).

3.3.6. Heartbeat Acknowledgement (HEARTBEAT ACK) (5)

An endpoint **MUST** send this chunk to its peer endpoint as a response to a HEARTBEAT chunk (see Section 8.3). A packet containing the HEARTBEAT ACK chunk is always sent to the source IP address of the IP datagram containing the HEARTBEAT chunk to which this HEARTBEAT ACK chunk is responding.

The parameter field contains a variable-length opaque data structure.



Chunk Flags: 8 bits
: Set to 0 on transmit and ignored on receipt.

Heartbeat Ack Length: 16 bits (unsigned integer)
: Set to the size of the chunk in bytes, including the chunk header and the Heartbeat Information field.

Heartbeat Information: variable length
: This field **MUST** contain the Heartbeat Info parameter (as defined in Section 3.3.5) of the Heartbeat Request to which this Heartbeat Acknowledgement is responding.

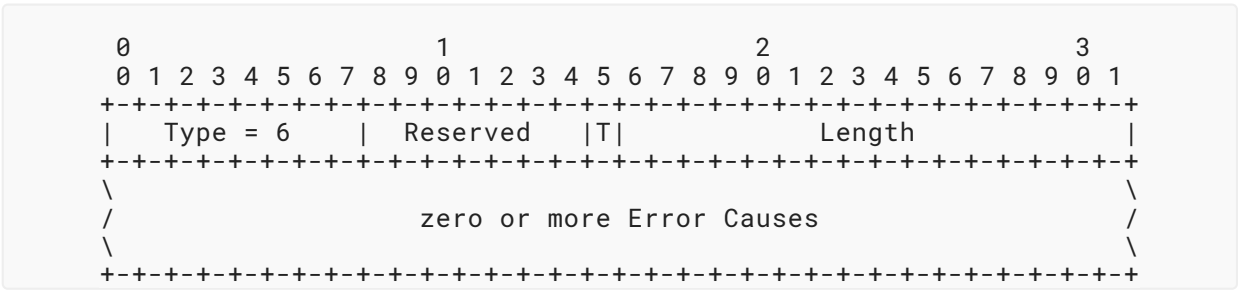
Variable Parameters	Status	Type Value
Heartbeat Info	Mandatory	1

Table 10: Variable-Length Parameters of HEARTBEAT ACK Chunks

3.3.7. Abort Association (ABORT) (6)

The ABORT chunk is sent to the peer of an association to close the association. The ABORT chunk **MAY** contain error causes to inform the receiver about the reason of the abort. DATA chunks **MUST NOT** be bundled with ABORT chunks. Control chunks (except for INIT, INIT ACK, and SHUTDOWN COMPLETE) **MAY** be bundled with an ABORT chunk, but they **MUST** be placed before the ABORT chunk in the SCTP packet; otherwise, they will be ignored by the receiver.

If an endpoint receives an ABORT chunk with a format error or no TCB is found, it **MUST** silently discard it. Moreover, under any circumstances, an endpoint that receives an ABORT chunk **MUST NOT** respond to that ABORT chunk by sending an ABORT chunk of its own.



Chunk Flags: 8 bits

Reserved: 7 bits

Set to 0 on transmit and ignored on receipt.

T bit: 1 bit

The T bit is set to 0 if the sender filled in the Verification Tag expected by the peer. If the Verification Tag is reflected, the T bit **MUST** be set to 1. Reflecting means that the sent Verification Tag is the same as the received one.

Length: 16 bits (unsigned integer)

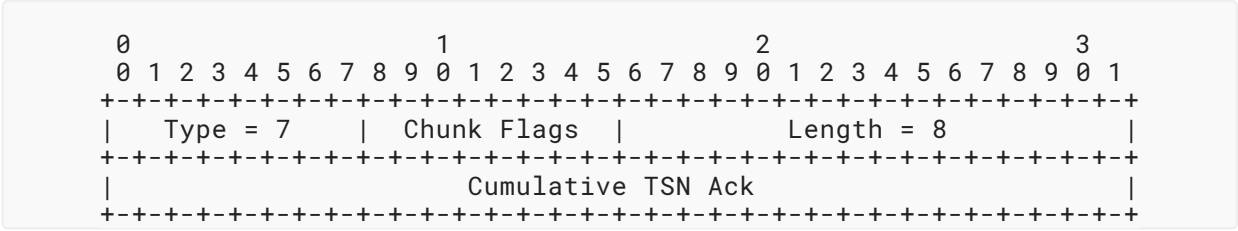
Set to the size of the chunk in bytes, including the chunk header and all the Error Cause fields present.

See Section 3.3.10 for Error Cause definitions.

Note: Special rules apply to this chunk for verification; please see Section 8.5.1 for details.

3.3.8. Shutdown Association (SHUTDOWN) (7)

An endpoint in an association **MUST** use this chunk to initiate a graceful close of the association with its peer. This chunk has the following format.



Chunk Flags: 8 bits
Set to 0 on transmit and ignored on receipt.

Length: 16 bits (unsigned integer)
Indicates the length of the parameter. Set to 8.

Cumulative TSN Ack: 32 bits (unsigned integer)
The largest TSN, such that all TSNs smaller than or equal to it have been received and the next one has not been received.

Note: Since the SHUTDOWN chunk does not contain Gap Ack Blocks, it cannot be used to acknowledge TSNs received out of order. In a SACK chunk, lack of Gap Ack Blocks that were previously included indicates that the data receiver reneged on the associated DATA chunks.

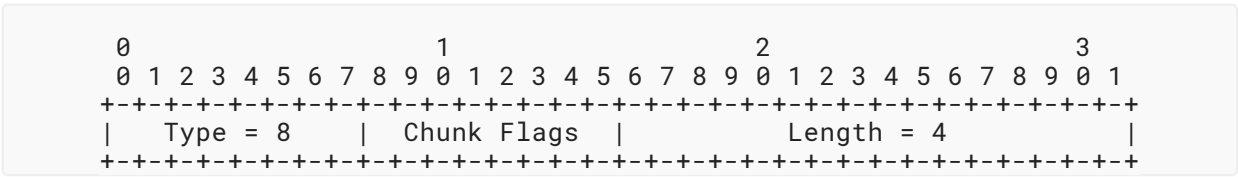
Since the SHUTDOWN chunk does not contain Gap Ack Blocks, the receiver of the SHUTDOWN chunk **MUST NOT** interpret the lack of a Gap Ack Block as a renege. (See Section 6.2 for information on reneging.)

The sender of the SHUTDOWN chunk **MAY** bundle a SACK chunk to indicate any gaps in the received TSNs.

3.3.9. Shutdown Acknowledgement (SHUTDOWN ACK) (8)

This chunk **MUST** be used to acknowledge the receipt of the SHUTDOWN chunk at the completion of the shutdown process; see Section 9.2 for details.

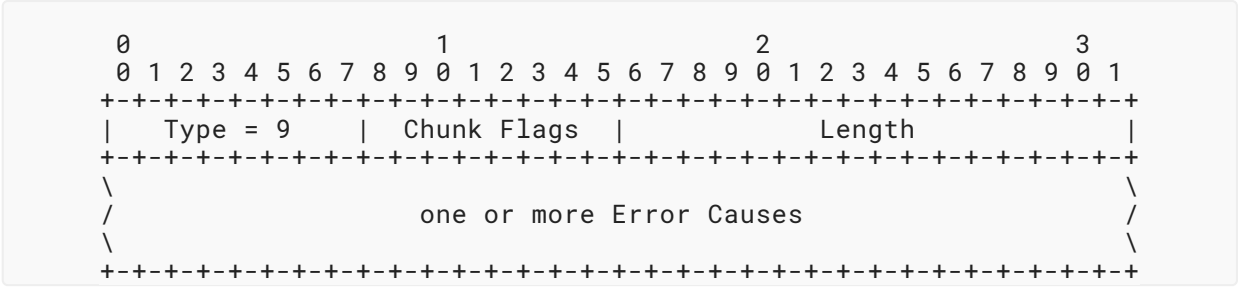
The SHUTDOWN ACK chunk has no parameters.



Chunk Flags: 8 bits
Set to 0 on transmit and ignored on receipt.

3.3.10. Operation Error (ERROR) (9)

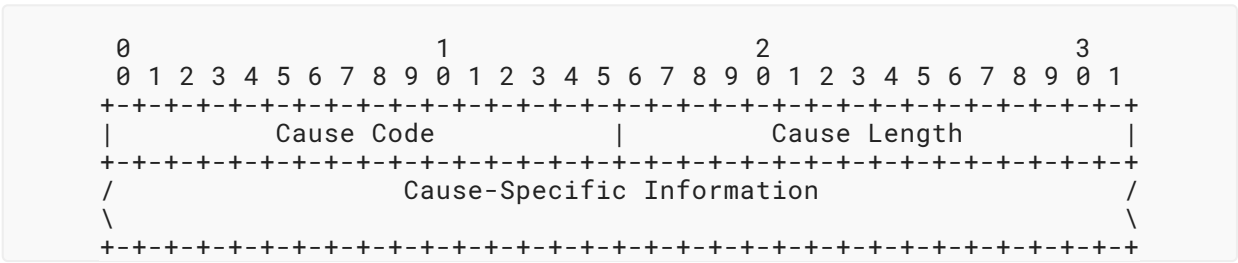
An endpoint sends this chunk to its peer endpoint to notify it of certain error conditions. It contains one or more error causes. An Operation Error is not considered fatal in and of itself, but the corresponding error cause **MAY** be used with an ABORT chunk to report a fatal condition. An ERROR chunk has the following format:



Chunk Flags: 8 bits
: Set to 0 on transmit and ignored on receipt.

Length: 16 bits (unsigned integer)
: Set to the size of the chunk in bytes, including the chunk header and all the Error Cause fields present.

Error causes are defined as variable-length parameters using the format described in Section 3.2.1, that is:



Cause Code: 16 bits (unsigned integer)
: Defines the type of error conditions being reported.

Value	Cause Code
1	Invalid Stream Identifier
2	Missing Mandatory Parameter
3	Stale Cookie
4	Out of Resource
5	Unresolvable Address
6	Unrecognized Chunk Type
7	Invalid Mandatory Parameter
8	Unrecognized Parameters

Value	Cause Code
9	No User Data
10	Cookie Received While Shutting Down
11	Restart of an Association with New Addresses
12	User-Initiated Abort
13	Protocol Violation

Table 11: Cause Code

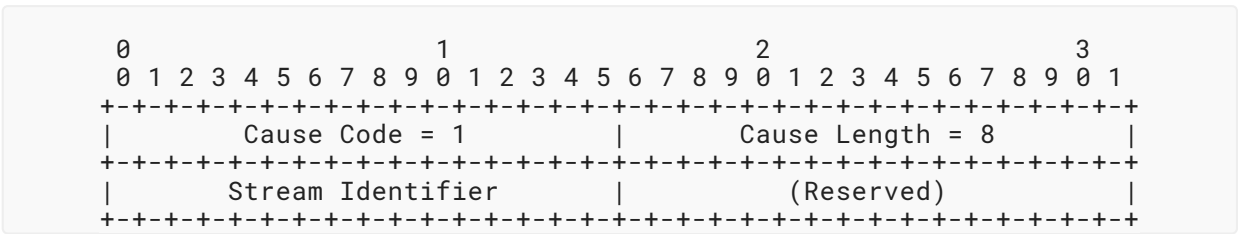
Cause Length: 16 bits (unsigned integer)
: Set to the size of the parameter in bytes, including the Cause Code, Cause Length, and Cause-Specific Information fields.

Cause-Specific Information: variable length
: This field carries the details of the error condition.

Sections 3.3.10.1 - 3.3.10.13 define error causes for SCTP. Guidelines for the IETF to define new error cause values are discussed in Section 15.4.

3.3.10.1. Invalid Stream Identifier (1)

Indicates that the endpoint received a DATA chunk sent using a nonexistent stream.

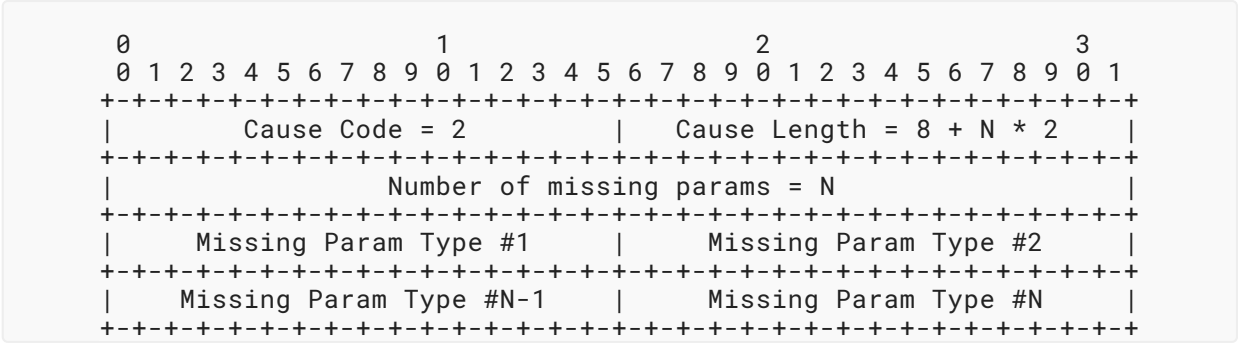


Stream Identifier: 16 bits (unsigned integer)
: Contains the Stream Identifier of the DATA chunk received in error.

Reserved: 16 bits
: This field is reserved. It is set to all 0's on transmit and ignored on receipt.

3.3.10.2. Missing Mandatory Parameter (2)

Indicates that one or more mandatory TLV parameters are missing in a received INIT or INIT ACK chunk.

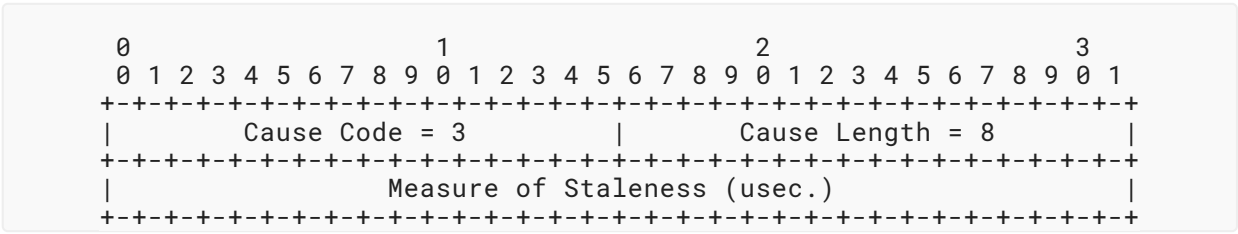


Number of Missing params: 32 bits (unsigned integer)
: This field contains the number of parameters contained in the Cause-Specific Information field.

Missing Param Type: 16 bits (unsigned integer)
: Each field will contain the missing mandatory parameter number.

3.3.10.3. Stale Cookie (3)

Indicates the receipt of a valid State Cookie that has expired.

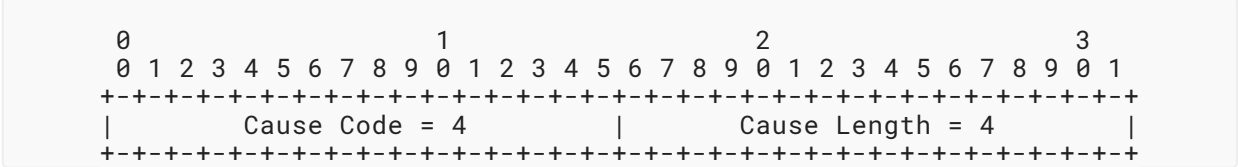


Measure of Staleness: 32 bits (unsigned integer)
: This field contains the difference, rounded up in microseconds, between the current time and the time the State Cookie expired.

  The sender of this error cause **MAY** choose to report how long past expiration the State Cookie is by including a non-zero value in the Measure of Staleness field. If the sender does not wish to provide the Measure of Staleness, it **SHOULD** set this field to the value of zero.

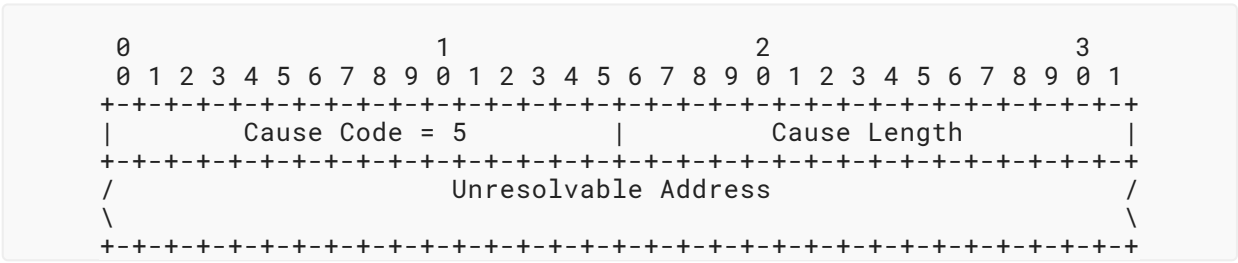
3.3.10.4. Out of Resource (4)

Indicates that the sender is out of resource. This is usually sent in combination with or within an ABORT chunk.



3.3.10.5. Unresolvable Address (5)

Indicates that the sender is not able to resolve the specified address parameter (e.g., type of address is not supported by the sender). This is usually sent in combination with or within an ABORT chunk.

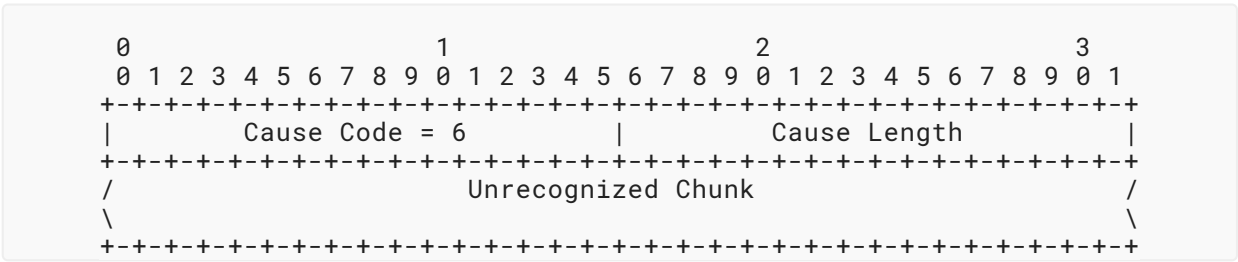


Unresolvable Address: variable length

The Unresolvable Address field contains the complete Type, Length, and Value of the address parameter (or Host Name parameter) that contains the unresolvable address or host name.

3.3.10.6. Unrecognized Chunk Type (6)

This error cause is returned to the originator of the chunk if the receiver does not understand the chunk and the upper bits of the 'Chunk Type' are set to 01 or 11.

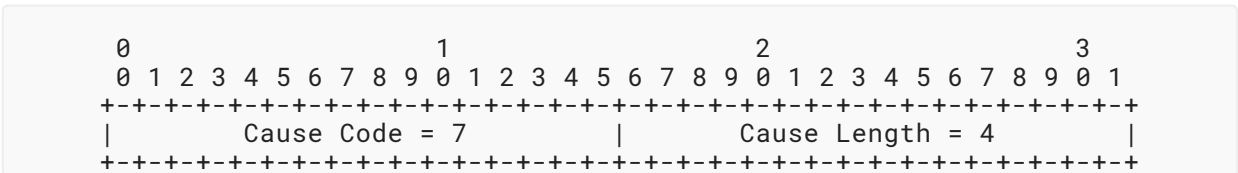


Unrecognized Chunk: variable length

The Unrecognized Chunk field contains the unrecognized chunk from the SCTP packet complete with Chunk Type, Chunk Flags, and Chunk Length.

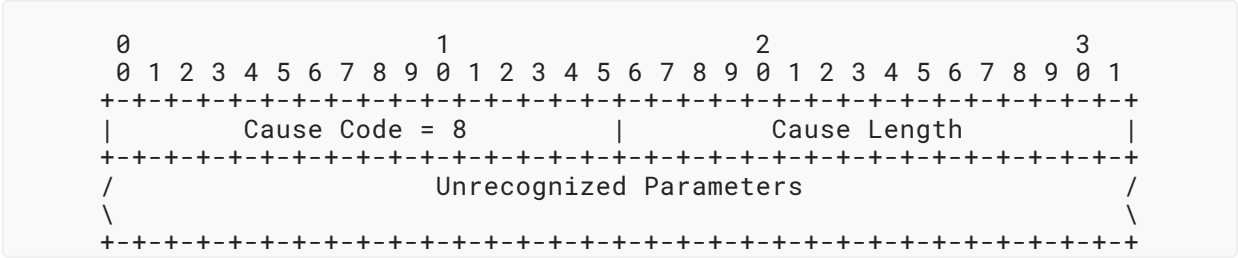
3.3.10.7. Invalid Mandatory Parameter (7)

This error cause is returned to the originator of an INIT or INIT ACK chunk when one of the mandatory parameters is set to an invalid value.



3.3.10.8. Unrecognized Parameters (8)

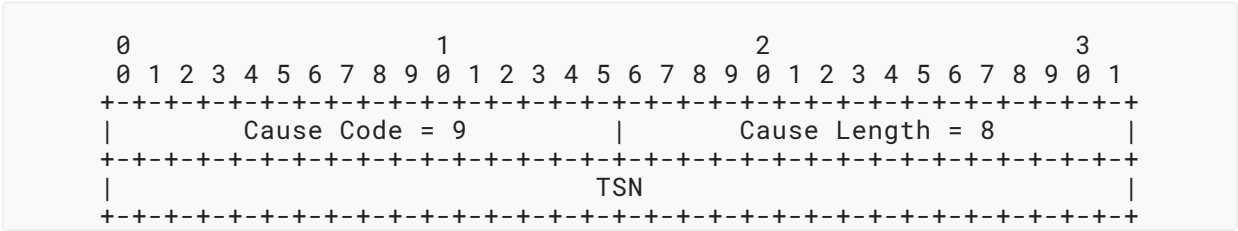
This error cause is returned to the originator of the INIT ACK chunk if the receiver does not recognize one or more Optional TLV parameters in the INIT ACK chunk.



Unrecognized Parameters: variable length
: The Unrecognized Parameters field contains the unrecognized parameters copied from the INIT ACK chunk complete with TLV. This error cause is normally contained in an ERROR chunk bundled with the COOKIE ECHO chunk when responding to the INIT ACK chunk, when the sender of the COOKIE ECHO chunk wishes to report unrecognized parameters.

3.3.10.9. No User Data (9)

This error cause is returned to the originator of a DATA chunk if a received DATA chunk has no user data.

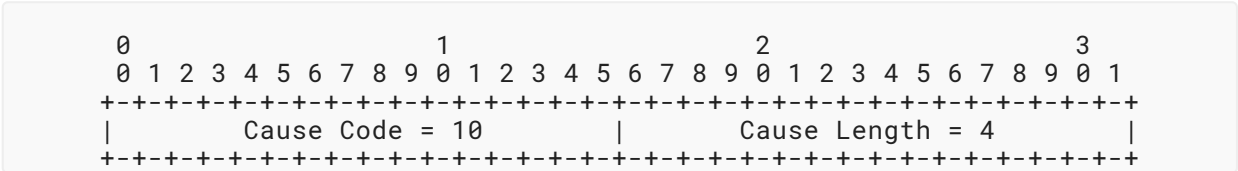


TSN: 32 bits (unsigned integer)
: This parameter contains the TSN of the DATA chunk received with no User Data field.

This cause code is normally returned in an ABORT chunk (see Section 6.2).

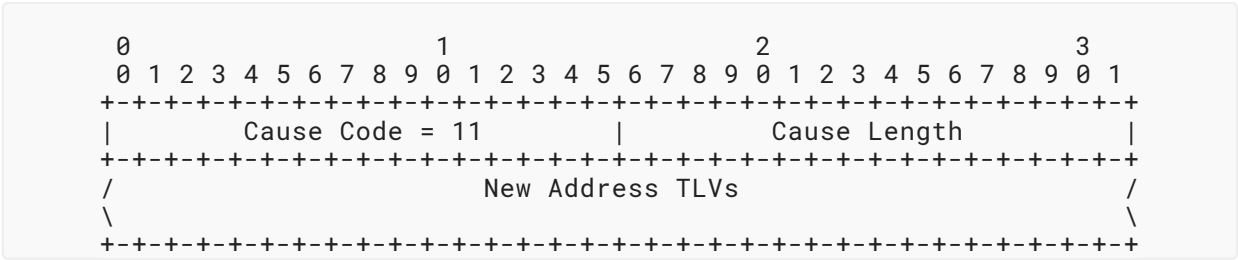
3.3.10.10. Cookie Received While Shutting Down (10)

A COOKIE ECHO chunk was received while the endpoint was in the SHUTDOWN-ACK-SENT state. This error is usually returned in an ERROR chunk bundled with the retransmitted SHUTDOWN ACK chunk.



3.3.10.11. Restart of an Association with New Addresses (11)

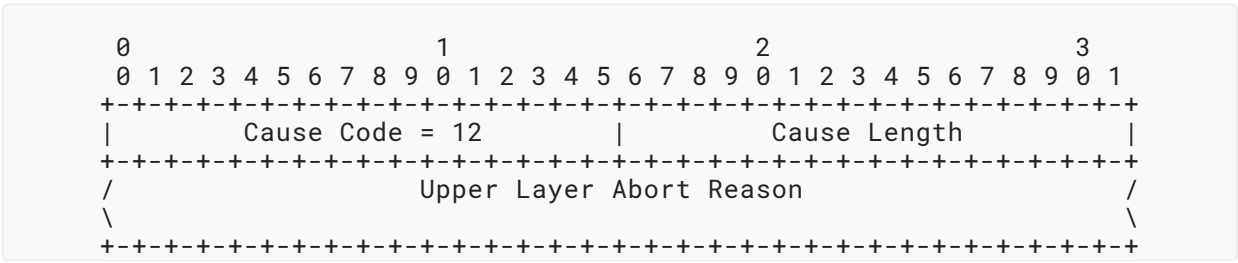
An INIT chunk was received on an existing association. But the INIT chunk added addresses to the association that were previously not part of the association. The new addresses are listed in the error cause. This error cause is normally sent as part of an ABORT chunk refusing the INIT chunk (see Section 5.2).



Note: Each New Address TLV is an exact copy of the TLV that was found in the INIT chunk that was new, including the Parameter Type and the Parameter Length.

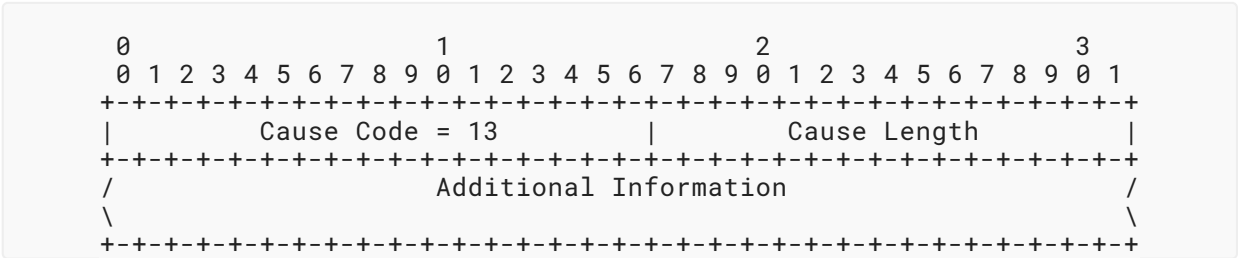
3.3.10.12. User-Initiated Abort (12)

This error cause **MAY** be included in ABORT chunks that are sent because of an upper-layer request. The upper layer can specify an Upper Layer Abort Reason that is transported by SCTP transparently and **MAY** be delivered to the upper-layer protocol at the peer.



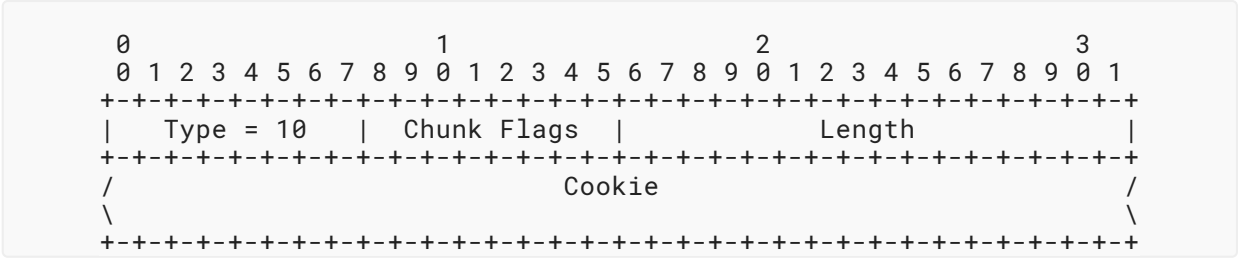
3.3.10.13. Protocol Violation (13)

This error cause **MAY** be included in ABORT chunks that are sent because an SCTP endpoint detects a protocol violation of the peer that is not covered by the error causes described in Sections 3.3.10.1 - 3.3.10.12. An implementation **MAY** provide additional information specifying what kind of protocol violation has been detected.



3.3.11. Cookie Echo (COOKIE ECHO) (10)

This chunk is used only during the initialization of an association. It is sent by the initiator of an association to its peer to complete the initialization process. This chunk **MUST** precede any DATA chunk sent within the association but **MAY** be bundled with one or more DATA chunks in the same packet.



Chunk Flags: 8 bits
Set to 0 on transmit and ignored on receipt.

Length: 16 bits (unsigned integer)
Set to the size of the chunk in bytes, including the 4 bytes of the chunk header and the size of the cookie.

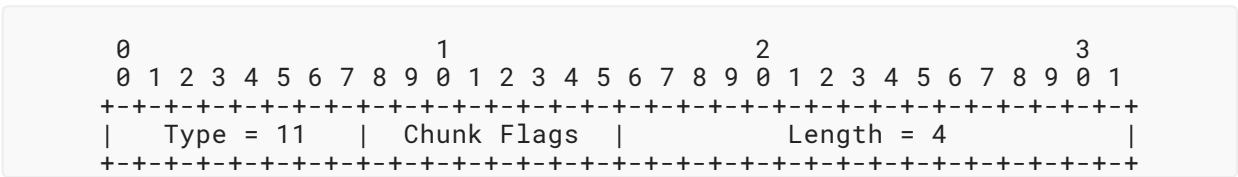
Cookie: variable size
This field **MUST** contain the exact cookie received in the State Cookie parameter from the previous INIT ACK chunk.

An implementation **SHOULD** make the cookie as small as possible to ensure interoperability.

Note: A Cookie Echo does not contain a State Cookie parameter; instead, the data within the State Cookie's Parameter Value becomes the data within the Cookie Echo's Chunk Value. This allows an implementation to change only the first 2 bytes of the State Cookie parameter to become a COOKIE ECHO chunk.

3.3.12. Cookie Acknowledgement (COOKIE ACK) (11)

This chunk is used only during the initialization of an association. It is used to acknowledge the receipt of a COOKIE ECHO chunk. This chunk **MUST** precede any DATA or SACK chunk sent within the association but **MAY** be bundled with one or more DATA chunks or SACK chunk's in the same SCTP packet.

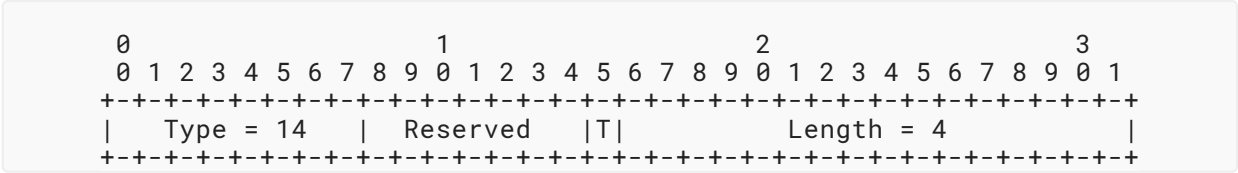


Chunk Flags: 8 bits
Set to 0 on transmit and ignored on receipt.

3.3.13. Shutdown Complete (SHUTDOWN COMPLETE) (14)

This chunk **MUST** be used to acknowledge the receipt of the SHUTDOWN ACK chunk at the completion of the shutdown process; see Section 9.2 for details.

The SHUTDOWN COMPLETE chunk has no parameters.



Chunk Flags: 8 bits

Reserved: 7 bits

Set to 0 on transmit and ignored on receipt.

T bit: 1 bit

The T bit is set to 0 if the sender filled in the Verification Tag expected by the peer. If the Verification Tag is reflected, the T bit **MUST** be set to 1. Reflecting means that the sent Verification Tag is the same as the received one.

Note: Special rules apply to this chunk for verification; please see Section 8.5.1 for details.

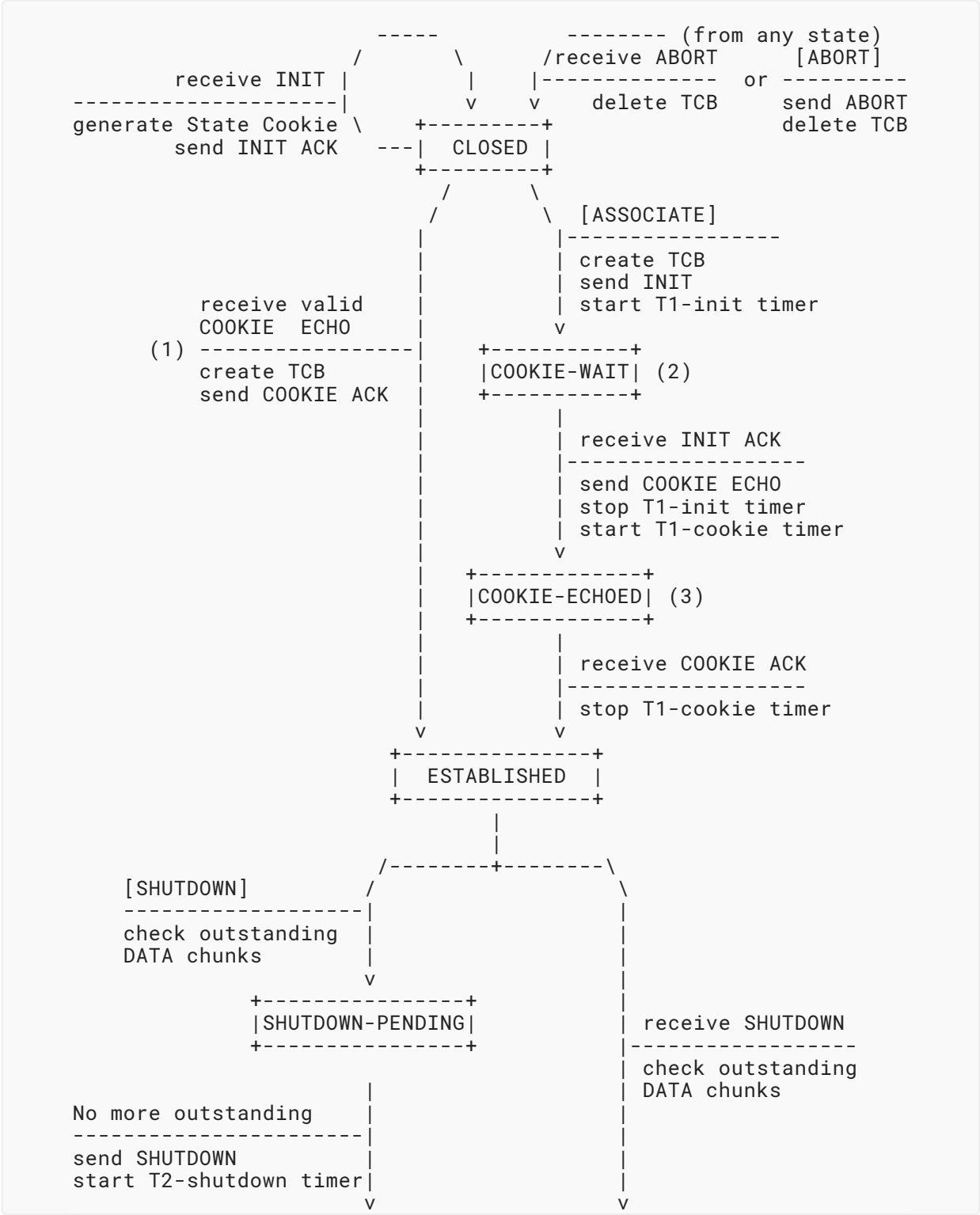
4. SCTP Association State Diagram

During the life time of an SCTP association, the SCTP endpoint's association progresses from one state to another in response to various events. The events that might potentially advance an association's state include:

- SCTP user primitive calls, e.g., [ASSOCIATE], [SHUTDOWN], or [ABORT],
- reception of INIT, COOKIE ECHO, ABORT, SHUTDOWN, etc., and control chunks, or
- some timeout events.

The state diagram in the figures below illustrates state changes, together with the causing events and resulting actions. Note that some of the error conditions are not shown in the state diagram. Full descriptions of all special cases are found in the text.

Note: Chunk names are given in all capital letters, while parameter names have the first letter capitalized, e.g., COOKIE ECHO chunk type vs. State Cookie parameter. If more than one event/message can occur that causes a state transition, it is labeled (A) or (B).


(from any state)
receive ABORT
[ABORT]
or
receive INIT
delete TCB
send ABORT
generate State Cookie
delete TCB
send INIT ACK
CLOSED
[ASSOCIATE]
create TCB
send INIT
start T1-init timer
receive valid
COOKIE ECHO
(1)
create TCB
send COOKIE ACK
COOKIE-WAIT
(2)
receive INIT ACK
send COOKIE ECHO
stop T1-init timer
start T1-cookie timer
COOKIE-ECHOED
(3)
receive COOKIE ACK
stop T1-cookie timer
ESTABLISHED
[SHUTDOWN]
check outstanding
DATA chunks
SHUTDOWN-PENDING
receive SHUTDOWN
check outstanding
DATA chunks
No more outstanding
send SHUTDOWN
start T2-shutdown timer

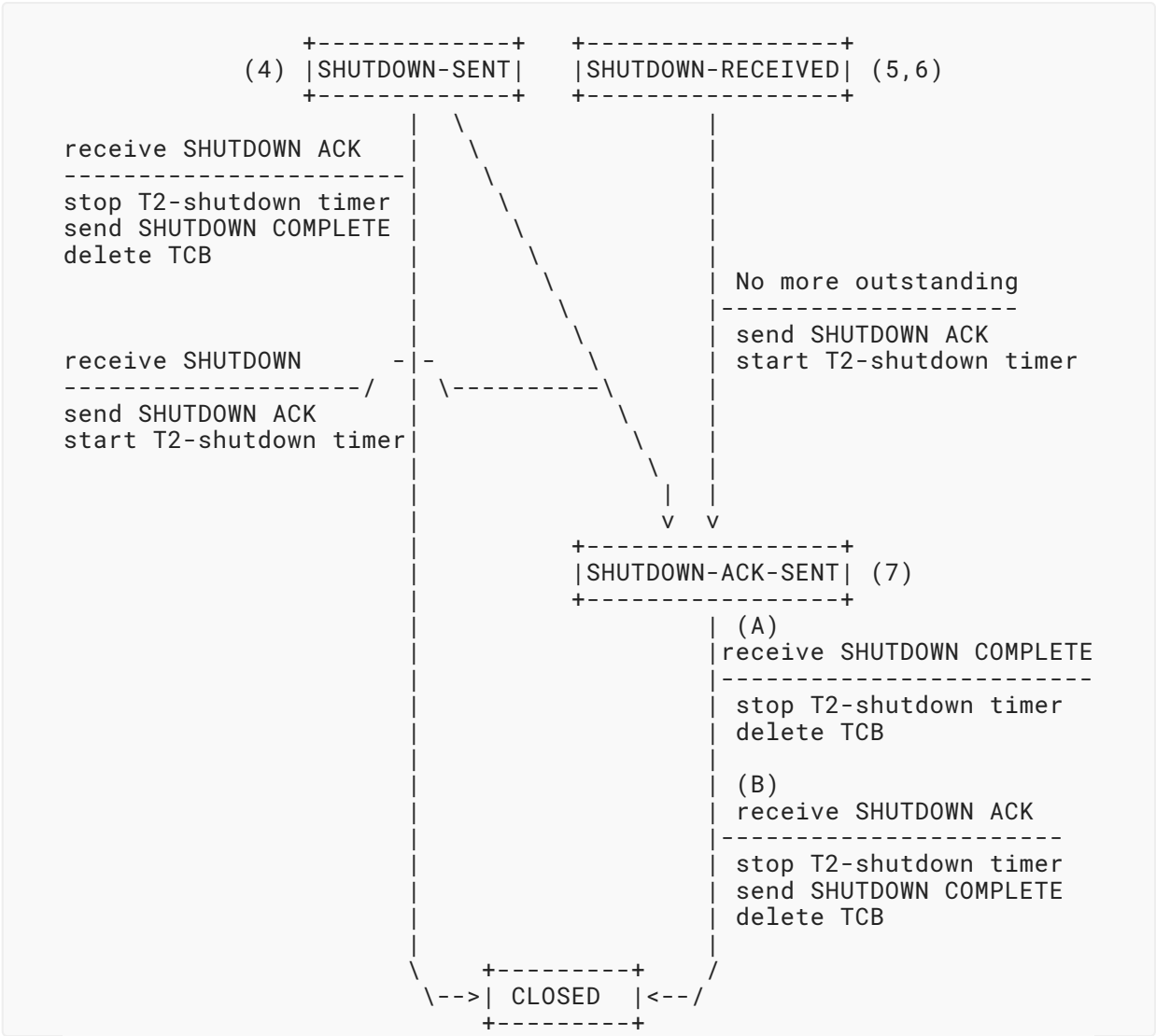


Figure 3: State Transition Diagram of SCTP

The following applies:

1) If the State Cookie in the received COOKIE ECHO chunk is invalid (i.e., failed to pass the integrity check), the receiver **MUST** silently discard the packet. Or, if the received State Cookie is expired (see Section 5.1.5), the receiver **MUST** send back an ERROR chunk. In either case, the receiver stays in the CLOSED state.

2) If the T1-init timer expires, the endpoint **MUST** retransmit the INIT chunk and restart the T1-init timer. The endpoint stays in the COOKIE-WAIT state. This **MUST** be repeated up to 'Max.Init.Retransmits' times. After that, the endpoint **MUST** abort the initialization process and report the error to the SCTP user.

3) If the T1-cookie timer expires, the endpoint **MUST** retransmit COOKIE ECHO chunk and restart the T1-cookie timer. The endpoint stays in the COOKIE-ECHOED state. This **MUST** be repeated up to 'Max.Init.Retransmits' times. After that, the endpoint **MUST** abort the initialization process and report the error to the SCTP user.

4) In the SHUTDOWN-SENT state, the endpoint **MUST** acknowledge any received DATA chunks without delay.

5) In the SHUTDOWN-RECEIVED state, the endpoint **MUST NOT** accept any new send requests from its SCTP user.

6) In the SHUTDOWN-RECEIVED state, the endpoint **MUST** transmit or retransmit data and leave this state when all data in queue is transmitted.

7) In the SHUTDOWN-ACK-SENT state, the endpoint **MUST NOT** accept any new send requests from its SCTP user.

The CLOSED state is used to indicate that an association is not created (i.e., does not exist).

5. Association Initialization

Before the first data transmission can take place from one SCTP endpoint ("A") to another SCTP endpoint ("Z"), the two endpoints **MUST** complete an initialization process in order to set up an SCTP association between them.

The SCTP user at an endpoint can use the ASSOCIATE primitive to initialize an SCTP association to another SCTP endpoint.

Implementation Note: From an SCTP user's point of view, an association might be implicitly opened, without an ASSOCIATE primitive (see Section 11.1.2) being invoked, by the initiating endpoint's sending of the first user data to the destination endpoint. The initiating SCTP will assume default values for all mandatory and optional parameters for the INIT/INIT ACK chunk.

Once the association is established, unidirectional streams are open for data transfer on both ends (see Section 5.1.1).

5.1. Normal Establishment of an Association

The initialization process consists of the following steps (assuming that SCTP endpoint "A" tries to set up an association with SCTP endpoint "Z" and "Z" accepts the new association):

A) "A" first builds a TCB and sends an INIT chunk to "Z". In the INIT chunk, "A" **MUST** provide its Verification Tag (Tag_A) in the Initiate Tag field. Tag_A **SHOULD** be a random number in the range of 1 to 4294967295 (see Section 5.3.1 for Tag value selection). After sending the INIT chunk, "A" starts the T1-init timer and enters the COOKIE-WAIT state.

B) "Z" responds immediately with an INIT ACK chunk. The destination IP address of the INIT ACK chunk **MUST** be set to the source IP address of the INIT chunk to which this INIT ACK chunk is responding. In the response, besides filling in other parameters, "Z" **MUST** set the Verification Tag field to Tag_A and also provide its own Verification Tag (Tag_Z) in the Initiate Tag field.

Moreover, "Z" **MUST** generate and send along with the INIT ACK chunk a State Cookie. See Section 5.1.3 for State Cookie generation.

After sending an INIT ACK chunk with the State Cookie parameter, "Z" **MUST NOT** allocate any resources or keep any states for the new association. Otherwise, "Z" will be vulnerable to resource attacks.

C) Upon reception of the INIT ACK chunk from "Z", "A" stops the T1-init timer and leaves the COOKIE-WAIT state. "A" then sends the State Cookie received in the INIT ACK chunk in a COOKIE ECHO chunk, starts the T1-cookie timer, and enters the COOKIE-ECHOED state.

The COOKIE ECHO chunk **MAY** be bundled with any pending outbound DATA chunks, but it **MUST** be the first chunk in the packet and, until the COOKIE ACK chunk is returned, the sender **MUST NOT** send any other packets to the peer.

D) Upon reception of the COOKIE ECHO chunk, endpoint "Z" replies with a COOKIE ACK chunk after building a TCB and moving to the ESTABLISHED state. A COOKIE ACK chunk **MAY** be bundled with any pending DATA chunks (and/or SACK chunks), but the COOKIE ACK chunk **MUST** be the first chunk in the packet.

Implementation Note: An implementation can choose to send the COMMUNICATION UP notification to the SCTP user upon reception of a valid COOKIE ECHO chunk.

E) Upon reception of the COOKIE ACK chunk, endpoint "A" moves from the COOKIE-ECHOED state to the ESTABLISHED state, stopping the T1-cookie timer. It can also notify its ULP about the successful establishment of the association with a COMMUNICATION UP notification (see Section 11).

An INIT or INIT ACK chunk **MUST NOT** be bundled with any other chunk. They **MUST** be the only chunks present in the SCTP packets that carry them.

An endpoint **MUST** send the INIT ACK chunk to the IP address from which it received the INIT chunk.

The T1-init timer and T1-cookie timer **SHOULD** follow the same rules given in Section 6.3. If the application provided multiple IP addresses of the peer, there **SHOULD** be a T1-init and T1-cookie timer for each address of the peer. Retransmissions of INIT chunks and COOKIE ECHO chunks **SHOULD** use all addresses of the peer similar to retransmissions of DATA chunks.

If an endpoint receives an INIT, INIT ACK, or COOKIE ECHO chunk but decides not to establish the new association due to missing mandatory parameters in the received INIT or INIT ACK chunk, invalid parameter values, or lack of local resources, it **SHOULD** respond with an ABORT chunk. It

SHOULD also specify the cause of abort, such as the type of the missing mandatory parameters, etc., by including an error cause in the ABORT chunk. The Verification Tag field in the common header of the outbound SCTP packet containing the ABORT chunk **MUST** be set to the Initiate Tag value of the received INIT or INIT ACK chunk this ABORT chunk is responding to.

Note that a COOKIE ECHO chunk that does not pass the integrity check is not considered an 'invalid mandatory parameter' and requires special handling; see Section 5.1.5.

After the reception of the first DATA chunk in an association, the endpoint **MUST** immediately respond with a SACK chunk to acknowledge the DATA chunk. Subsequent acknowledgements **SHOULD** be done as described in Section 6.2.

When the TCB is created, each endpoint **MUST** set its internal Cumulative TSN Ack Point to the value of its transmitted Initial TSN minus one.

Implementation Note: The IP addresses and SCTP port are generally used as the key to find the TCB within an SCTP instance.

5.1.1. Handle Stream Parameters

In the INIT and INIT ACK chunks, the sender of the chunk **MUST** indicate the number of outbound streams (OS) it wishes to have in the association, as well as the maximum inbound streams (MIS) it will accept from the other endpoint.

After receiving the stream configuration information from the other side, each endpoint **MUST** perform the following check: If the peer's MIS is less than the endpoint's OS, meaning that the peer is incapable of supporting all the outbound streams the endpoint wants to configure, the endpoint **MUST** use MIS outbound streams and **MAY** report any shortage to the upper layer. The upper layer can then choose to abort the association if the resource shortage is unacceptable.

After the association is initialized, the valid outbound stream identifier range for either endpoint **MUST** be 0 to min(local OS, remote MIS) - 1.

5.1.2. Handle Address Parameters

During the association initialization, an endpoint uses the following rules to discover and collect the destination transport address(es) of its peer.

A) If there are no address parameters present in the received INIT or INIT ACK chunk, the endpoint **MUST** take the source IP address from which the chunk arrives and record it, in combination with the SCTP Source Port Number, as the only destination transport address for this peer.

B) If there is a Host Name Address parameter present in the received INIT or INIT ACK chunk, the endpoint **MUST** immediately send an ABORT chunk and **MAY** include an "Unresolvable Address" error cause to its peer. The ABORT chunk **SHOULD** be sent to the source IP address from which the last peer packet was received.

C) If there are only IPv4/IPv6 addresses present in the received INIT or INIT ACK chunk, the receiver **MUST** derive and record all the transport addresses from the received chunk AND the source IP address that sent the INIT or INIT ACK chunk. The transport addresses are derived by the combination of SCTP Source Port Number (from the common header) and the IP Address parameter(s) carried in the INIT or INIT ACK chunk and the source IP address of the IP datagram. The receiver **SHOULD** use only these transport addresses as destination transport addresses when sending subsequent packets to its peer.

D) An INIT or INIT ACK chunk **MUST** be treated as belonging to an already established association (or one in the process of being established) if the use of any of the valid address parameters contained within the chunk would identify an existing TCB.

Implementation Note: In some cases (e.g., when the implementation does not control the source IP address that is used for transmitting), an endpoint might need to include in its INIT or INIT ACK chunk all possible IP addresses from which packets to the peer could be transmitted.

After all transport addresses are derived from the INIT or INIT ACK chunk using the above rules, the endpoint selects one of the transport addresses as the initial primary path.

The packet containing the INIT ACK chunk **MUST** be sent to the source address of the packet containing the INIT chunk.

The sender of INIT chunks **MAY** include a 'Supported Address Types' parameter in the INIT chunk to indicate what types of addresses are acceptable.

Implementation Note: In the case that the receiver of an INIT ACK chunk fails to resolve the address parameter due to an unsupported type, it can abort the initiation process and then attempt a reinitiation by using a 'Supported Address Types' parameter in the new INIT chunk to indicate what types of address it prefers.

If an SCTP endpoint that only supports either IPv4 or IPv6 receives IPv4 and IPv6 addresses in an INIT or INIT ACK chunk from its peer, it **MUST** use all the addresses belonging to the supported address family. The other addresses **MAY** be ignored. The endpoint **SHOULD NOT** respond with any kind of error indication.

If an SCTP endpoint lists in the 'Supported Address Types' parameter either IPv4 or IPv6 but uses the other family for sending the packet containing the INIT chunk, or if it also lists addresses of the other family in the INIT chunk, then the address family that is not listed in the 'Supported Address Types' parameter **SHOULD** also be considered as supported by the receiver of the INIT chunk. The receiver of the INIT chunk **SHOULD NOT** respond with any kind of error indication.

5.1.3. Generating State Cookie

When sending an INIT ACK chunk as a response to an INIT chunk, the sender of the INIT ACK chunk creates a State Cookie and sends it in the State Cookie parameter of the INIT ACK chunk. Inside this State Cookie, the sender **MUST** include a MAC (see [RFC2104] for an example) to provide integrity protection on the State Cookie. The State Cookie **SHOULD** also contain a timestamp on

when the State Cookie is created and the lifespan of the State Cookie, along with all the information necessary for it to establish the association, including the port numbers and the Verification Tags.

The method used to generate the MAC is strictly a private matter for the receiver of the INIT chunk. The use of a MAC is mandatory to prevent denial-of-service attacks. MAC algorithms can have different performances depending on the platform. Choosing a high-performance MAC algorithm increases the resistance against cookie flooding attacks. A MAC with acceptable security properties **SHOULD** be used. The secret key **SHOULD** be random ([RFC4086] provides some information on randomness guidelines). The secret keys need to have an appropriate size. The secret key **SHOULD** be changed reasonably frequently (e.g., hourly), and the timestamp in the State Cookie **MAY** be used to determine which key is used to verify the MAC.

If the State Cookie is not encrypted, it **MUST NOT** contain information that is not being envisioned to be shared.

An implementation **SHOULD** make the cookie as small as possible to ensure interoperability.

5.1.4. State Cookie Processing

When an endpoint (in the COOKIE-WAIT state) receives an INIT ACK chunk with a State Cookie parameter, it **MUST** immediately send a COOKIE ECHO chunk to its peer with the received State Cookie. The sender **MAY** also add any pending DATA chunks to the packet after the COOKIE ECHO chunk.

The endpoint **MUST** also start the T1-cookie timer after sending the COOKIE ECHO chunk. If the timer expires, the endpoint **MUST** retransmit the COOKIE ECHO chunk and restart the T1-cookie timer. This is repeated until either a COOKIE ACK chunk is received or 'Max.Init.Retransmits' (see Section 16) is reached, causing the peer endpoint to be marked unreachable (and thus the association enters the CLOSED state).

5.1.5. State Cookie Authentication

When an endpoint receives a COOKIE ECHO chunk from another endpoint with which it has no association, it takes the following actions:

1) Compute a MAC using the information carried in the State Cookie and the secret key. The timestamp in the State Cookie **MAY** be used to determine which secret key to use. If secrets are kept only for a limited amount of time and the secret key to use is not available anymore, the packet containing the COOKIE ECHO chunk **MUST** be silently discarded. [RFC2104] can be used as a guideline for generating the MAC.

2) Authenticate the State Cookie as one that it previously generated by comparing the computed MAC against the one carried in the State Cookie. If this comparison fails, the SCTP packet, including the COOKIE ECHO chunk and any DATA chunks, **MUST** be silently discarded.

3) Compare the port numbers and the Verification Tag contained within the COOKIE ECHO chunk to the actual port numbers and the Verification Tag within the SCTP common header of the received packet. If these values do not match, the packet **MUST** be silently discarded.

4) Compare the creation timestamp in the State Cookie to the current local time. If the elapsed time is longer than the lifespan carried in the State Cookie, then the packet, including the COOKIE ECHO chunk and any attached DATA chunks, **SHOULD** be discarded, and the endpoint **MUST** transmit an ERROR chunk with a "Stale Cookie" error cause to the peer endpoint.

5) If the State Cookie is valid, create an association to the sender of the COOKIE ECHO chunk with the information in the State Cookie carried in the COOKIE ECHO chunk and enter the ESTABLISHED state.

6) Send a COOKIE ACK chunk to the peer acknowledging receipt of the COOKIE ECHO chunk. The COOKIE ACK chunk **MAY** be bundled with an outbound DATA chunk or SACK chunk; however, the COOKIE ACK chunk **MUST** be the first chunk in the SCTP packet.

7) Immediately acknowledge any DATA chunk bundled with the COOKIE ECHO chunk with a SACK chunk (subsequent DATA chunk acknowledgement **SHOULD** follow the rules defined in Section 6.2). As mentioned in step 6, if the SACK chunk is bundled with the COOKIE ACK chunk, the COOKIE ACK chunk **MUST** appear first in the SCTP packet.

If a COOKIE ECHO chunk is received from an endpoint with which the receiver of the COOKIE ECHO chunk has an existing association, the procedures in Section 5.2 **SHOULD** be followed.

5.1.6. An Example of Normal Association Establishment

In the following example, "A" initiates the association and then sends a user message to "Z"; then, "Z" sends two user messages to "A" later (assuming no bundling or fragmentation occurs):

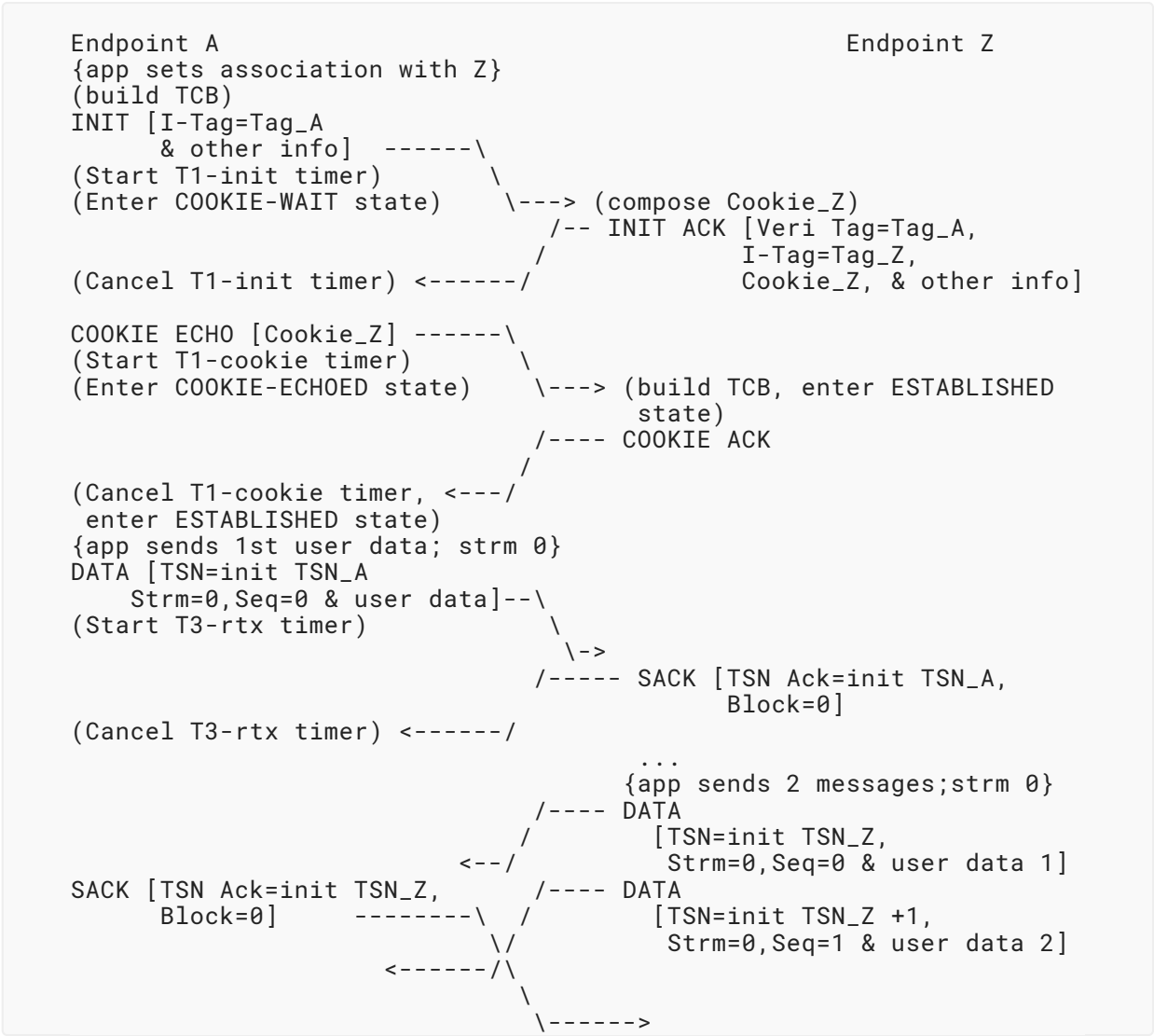


Figure 4: A Setup Example

If the T1-init timer expires at "A" after the INIT or COOKIE ECHO chunks are sent, the same INIT or COOKIE ECHO chunk with the same Initiate Tag (i.e., Tag_A) or State Cookie is retransmitted and the timer is restarted. This is repeated 'Max.Init.Retransmits' times before "A" considers "Z" unreachable and reports the failure to its upper layer (and thus the association enters the CLOSED state).

When retransmitting the INIT chunk, the endpoint **MUST** follow the rules defined in Section 6.3 to determine the proper timer value.

5.2. Handle Duplicate or Unexpected INIT, INIT ACK, COOKIE ECHO, and COOKIE ACK Chunks

During the life time of an association (in one of the possible states), an endpoint can receive from its peer endpoint one of the setup chunks (INIT, INIT ACK, COOKIE ECHO, or COOKIE ACK). The receiver treats such a setup chunk as a duplicate and process it as described in this section.

Note: An endpoint will not receive the chunk unless the chunk was sent to an SCTP transport address and is from an SCTP transport address associated with this endpoint. Therefore, the endpoint processes such a chunk as part of its current association.

The following scenarios can cause duplicated or unexpected chunks:

A) the peer has crashed without being detected, restarted itself, and sent a new INIT chunk trying to restore the association,

B) both sides are trying to initialize the association at about the same time,

C) the chunk is from a stale packet that was used to establish the present association or a past association that is no longer in existence,

D) the chunk is a false packet generated by an attacker, or

E) the peer never received the COOKIE ACK chunk and is retransmitting its COOKIE ECHO chunk.

The rules in the following sections are applied in order to identify and correctly handle these cases.

5.2.1. INIT Chunk Received in COOKIE-WAIT or COOKIE-ECHOED State (Item B)

This usually indicates an initialization collision, i.e., each endpoint is attempting, at about the same time, to establish an association with the other endpoint.

Upon receipt of an INIT chunk in the COOKIE-WAIT state, an endpoint **MUST** respond with an INIT ACK chunk using the same parameters it sent in its original INIT chunk (including its Initiate Tag, unchanged). When responding, the following rules **MUST** be applied:

1) The packet containing the INIT ACK chunk **MUST** only be sent to an address passed by the upper layer in the request to initialize the association.

2) The packet containing the INIT ACK chunk **MUST** only be sent to an address reported in the incoming INIT chunk.

3) The packet containing the INIT ACK chunk **SHOULD** be sent to the source address of the received packet containing the INIT chunk.

Upon receipt of an INIT chunk in the COOKIE-ECHOED state, an endpoint **MUST** respond with an INIT ACK chunk using the same parameters it sent in its original INIT chunk (including its Initiate Tag, unchanged), provided that no new address has been added to the forming association. If the INIT chunk indicates that a new address has been added to the association, then the entire INIT chunk **MUST** be discarded, and the state of the existing association **SHOULD NOT** be changed. An ABORT chunk **SHOULD** be sent in a response that **MAY** include the "Restart of an Association with New Addresses" error cause. The error **SHOULD** list the addresses that were added to the restarting association.

When responding in either state (COOKIE-WAIT or COOKIE-ECHOED) with an INIT ACK chunk, the original parameters are combined with those from the newly received INIT chunk. The endpoint **MUST** also generate a State Cookie with the INIT ACK chunk. The endpoint uses the parameters sent in its INIT chunk to calculate the State Cookie.

After that, the endpoint **MUST NOT** change its state, the T1-init timer **MUST** be left running, and the corresponding TCB **MUST NOT** be destroyed. The normal procedures for handling State Cookies when a TCB exists will resolve the duplicate INIT chunks to a single association.

For an endpoint that is in the COOKIE-ECHOED state, it **MUST** populate its Tie-Tags within both the association TCB and inside the State Cookie (see Section 5.2.2 for a description of the Tie-Tags).

5.2.2. Unexpected INIT Chunk in States Other than CLOSED, COOKIE-ECHOED, COOKIE-WAIT, and SHUTDOWN-ACK-SENT

Unless otherwise stated, upon receipt of an unexpected INIT chunk for this association, the endpoint **MUST** generate an INIT ACK chunk with a State Cookie. Before responding, the endpoint **MUST** check to see if the unexpected INIT chunk adds new addresses to the association. If new addresses are added to the association, the endpoint **MUST** respond with an ABORT chunk, copying the 'Initiate Tag' of the unexpected INIT chunk into the 'Verification Tag' of the outbound packet carrying the ABORT chunk. In the ABORT chunk, the error cause **MAY** be set to "Restart of an Association with New Addresses". The error **SHOULD** list the addresses that were added to the restarting association. If no new addresses are added, when responding to the INIT chunk in the outbound INIT ACK chunk, the endpoint **MUST** copy its current Tie-Tags to a reserved place within the State Cookie and the association's TCB. We refer to these locations inside the cookie as the Peer's-Tie-Tag and the Local-Tie-Tag. We will refer to the copy within an association's TCB as the Local Tag and Peer's Tag. The outbound SCTP packet containing this INIT ACK chunk **MUST** carry a Verification Tag value equal to the Initiate Tag found in the unexpected INIT chunk. And the INIT ACK chunk **MUST** contain a new Initiate Tag (randomly generated; see Section 5.3.1). Other parameters for the endpoint **SHOULD** be copied from the existing parameters of the association (e.g., number of outbound streams) into the INIT ACK chunk and cookie.

After sending the INIT ACK or ABORT chunk, the endpoint **MUST** take no further actions, i.e., the existing association, including its current state, and the corresponding TCB **MUST NOT** be changed.

Only when a TCB exists and the association is not in a COOKIE-WAIT or SHUTDOWN-ACK-SENT state are the Tie-Tags populated with a random value other than 0. For a normal association INIT chunk (i.e., the endpoint is in the CLOSED state), the Tie-Tags **MUST** be set to 0 (indicating that no previous TCB existed).

5.2.3. Unexpected INIT ACK Chunk

If an INIT ACK chunk is received by an endpoint in any state other than the COOKIE-WAIT or CLOSED state, the endpoint **SHOULD** discard the INIT ACK chunk. An unexpected INIT ACK chunk usually indicates the processing of an old or duplicated INIT chunk.

5.2.4. Handle a COOKIE ECHO Chunk When a TCB Exists

When a COOKIE ECHO chunk is received by an endpoint in any state for an existing association (i.e., not in the CLOSED state), the following rules are applied:

1) Compute a MAC as described in step 1 of Section 5.1.5.

2) Authenticate the State Cookie as described in step 2 of Section 5.1.5 (this is case C or D above).

3) Compare the timestamp in the State Cookie to the current time. If the State Cookie is older than the lifespan carried in the State Cookie and the Verification Tags contained in the State Cookie do not match the current association's Verification Tags, the packet, including the COOKIE ECHO chunk and any DATA chunks, **SHOULD** be discarded. The endpoint also **MUST** transmit an ERROR chunk with a "Stale Cookie" error cause to the peer endpoint (this is case C or D in Section 5.2).

   If both Verification Tags in the State Cookie match the Verification Tags of the current association, consider the State Cookie valid (this is case E in Section 5.2), even if the lifespan is exceeded.

4) If the State Cookie proves to be valid, unpack the TCB into a temporary TCB.

5) Refer to Table 12 to determine the correct action to be taken.

Local Tag	Peer's Tag	Local-Tie-Tag	Peer's-Tie-Tag	Action
X	X	M	M	(A)
M	X	A	A	(B)
M	0	A	A	(B)
X	M	0	0	(C)
M	M	A	A	(D)

Table 12: Handling of a COOKIE ECHO Chunk When a TCB Exists

Legend:

X - Tag does not match the existing TCB.
M - Tag matches the existing TCB.
0 - Tag unknown (Peer's Tag not known yet / No Tie-Tag in cookie).
A - All cases, i.e., M, X, or 0.

For any case not shown in Table 12, the cookie **SHOULD** be silently discarded.

Action:

A) In this case, the peer might have restarted. When the endpoint recognizes this potential 'restart', the existing session is treated the same as if it received an ABORT chunk followed by a new COOKIE ECHO chunk with the following exceptions:

- Any SCTP DATA chunks **MAY** be retained (this is an implementation-specific option).
- A RESTART notification **SHOULD** be sent to the ULP instead of a COMMUNICATION LOST notification.

All the congestion control parameters (e.g., cwnd, ssthresh) related to this peer **MUST** be reset to their initial values (see Section 6.2.1).

After this, the endpoint enters the ESTABLISHED state.

If the endpoint is in the SHUTDOWN-ACK-SENT state and recognizes that the peer has restarted (Action A), it **MUST NOT** set up a new association but instead resend the SHUTDOWN ACK chunk and send an ERROR chunk with a "Cookie Received While Shutting Down" error cause to its peer.

B) In this case, both sides might be attempting to start an association at about the same time, but the peer endpoint sent its INIT chunk after responding to the local endpoint's INIT chunk. Thus, it might have picked a new Verification Tag, not being aware of the previous tag it had sent this endpoint. The endpoint **SHOULD** stay in or enter the ESTABLISHED state, but it **MUST** update its peer's Verification Tag from the State Cookie, stop any T1-init or T1-cookie timers that might be running, and send a COOKIE ACK chunk.

C) In this case, the local endpoint's cookie has arrived late. Before it arrived, the local endpoint sent an INIT chunk and received an INIT ACK chunk and finally sent a COOKIE ECHO chunk with the peer's same tag but a new tag of its own. The cookie **SHOULD** be silently discarded. The endpoint **SHOULD NOT** change states and **SHOULD** leave any timers running.

D) When both local and remote tags match, the endpoint **SHOULD** enter the ESTABLISHED state if it is in the COOKIE-ECHOED state. It **SHOULD** stop any T1-cookie timer that is running and send a COOKIE ACK chunk.

Note: The "peer's Verification Tag" is the tag received in the Initiate Tag field of the INIT or INIT ACK chunk.

5.2.4.1. An Example of an Association Restart

In the following example, "A" initiates the association after a restart has occurred. Endpoint "Z" had no knowledge of the restart until the exchange (i.e., Heartbeats had not yet detected the failure of "A") (assuming no bundling or fragmentation occurs):

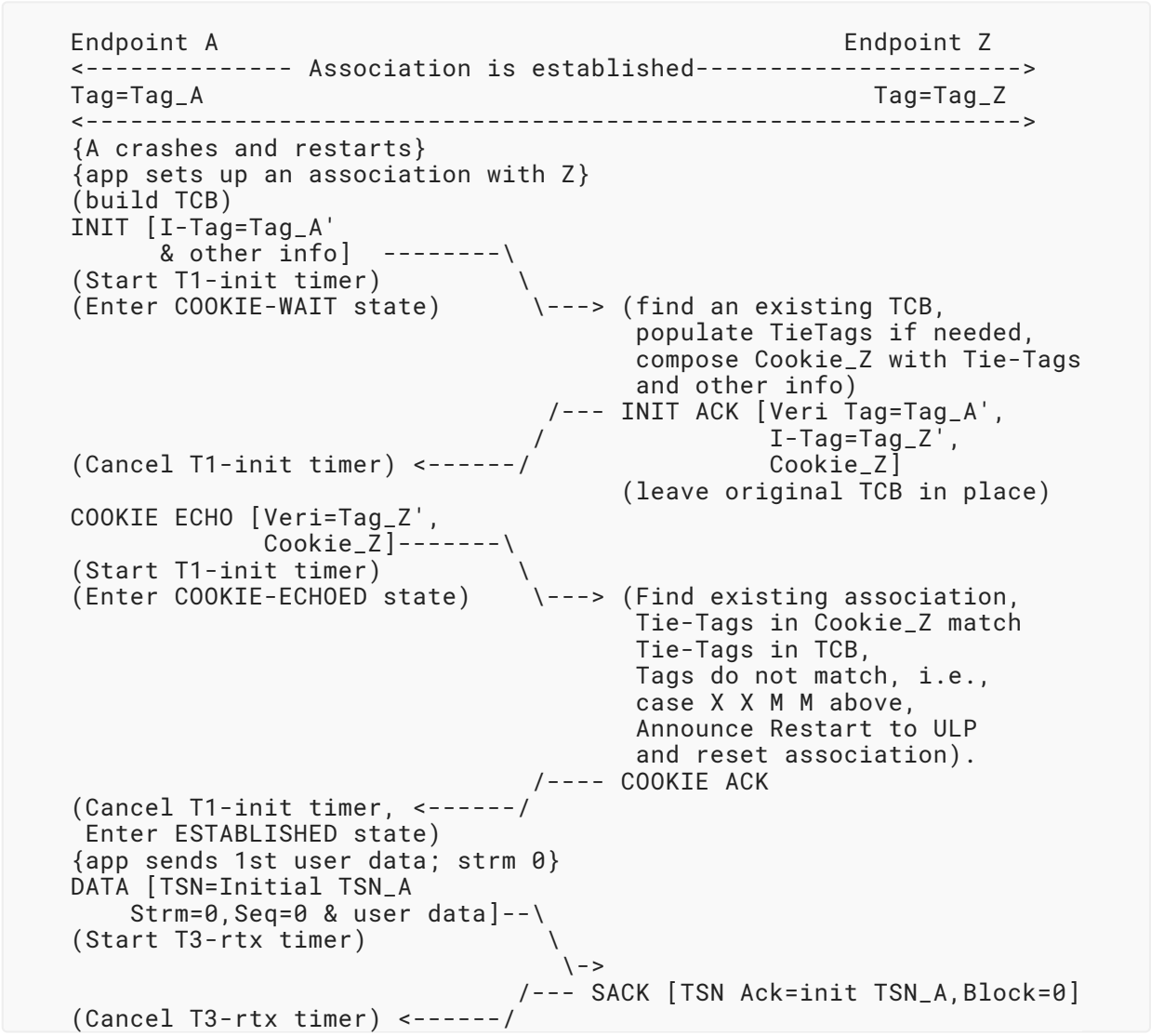


Figure 5: A Restart Example

5.2.5. Handle Duplicate COOKIE ACK Chunk

At any state other than COOKIE-ECHOED, an endpoint **SHOULD** silently discard a received COOKIE ACK chunk.

5.2.6. Handle Stale Cookie Error

Receipt of an ERROR chunk with a "Stale Cookie" error cause indicates one of a number of possible events:

A) The association failed to completely set up before the State Cookie issued by the sender was processed.

B) An old State Cookie was processed after setup completed.

C) An old State Cookie is received from someone that the receiver is not interested in having an association with and the ABORT chunk was lost.

When processing an ERROR chunk with a "Stale Cookie" error cause, an endpoint **SHOULD** first examine if an association is in the process of being set up, i.e., the association is in the COOKIE-ECHOED state. In all cases, if the association is not in the COOKIE-ECHOED state, the ERROR chunk **SHOULD** be silently discarded.

If the association is in the COOKIE-ECHOED state, the endpoint **MAY** elect one of the following three alternatives.

1) Send a new INIT chunk to the endpoint to generate a new State Cookie and reattempt the setup procedure.

2) Discard the TCB and report to the upper layer the inability to set up the association.

3) Send a new INIT chunk to the endpoint, adding a Cookie Preservative parameter requesting an extension to the life time of the State Cookie. When calculating the time extension, an implementation **SHOULD** use the RTT information measured based on the previous COOKIE ECHO/ERROR chunk exchange and **SHOULD** add no more than 1 second beyond the measured RTT, due to long State Cookie life times making the endpoint more subject to a replay attack.

5.3. Other Initialization Issues

5.3.1. Selection of Tag Value

Initiate Tag values **SHOULD** be selected from the range of 1 to 2^{32} - 1. It is very important that the Initiate Tag value be randomized to help protect against off-path attacks. The methods described in [RFC4086] can be used for the Initiate Tag randomization. Careful selection of Initiate Tags is also necessary to prevent old duplicate packets from previous associations being mistakenly processed as belonging to the current association.

Moreover, the Verification Tag value used by either endpoint in a given association **MUST NOT** change during the life time of an association. A new Verification Tag value **MUST** be used each time the endpoint tears down and then reestablishes an association to the same peer.

5.4. Path Verification

During association establishment, the two peers exchange a list of addresses. In the predominant case, these lists accurately represent the addresses owned by each peer. However, a misbehaving peer might supply addresses that it does not own. To prevent this, the following rules are applied to all addresses of the new association:

1) Any addresses passed to the sender of the INIT chunk by its upper layer in the request to initialize an association are automatically considered to be CONFIRMED.

2) For the receiver of the COOKIE ECHO chunk, the only CONFIRMED address is the address to which the packet containing the INIT ACK chunk was sent.

3) All other addresses not covered by rules 1 and 2 are considered UNCONFIRMED and are subject to probing for verification.

To probe an address for verification, an endpoint will send HEARTBEAT chunks including a 64-bit random nonce and a path indicator (to identify the address that the HEARTBEAT chunk is sent to) within the Heartbeat Info parameter.

Upon receipt of the HEARTBEAT ACK chunk, a verification is made that the nonce included in the Heartbeat Info parameter is the one sent to the address indicated inside the Heartbeat Info parameter. When this match occurs, the address that the original HEARTBEAT was sent to is now considered CONFIRMED and available for normal data transfer.

These probing procedures are started when an association moves to the ESTABLISHED state and are ended when all paths are confirmed.

In each RTO, a probe **MAY** be sent on an active UNCONFIRMED path in an attempt to move it to the CONFIRMED state. If during this probing the path becomes inactive, this rate is lowered to the normal HEARTBEAT rate. At the expiration of the RTO timer, the error counter of any path that was probed but not CONFIRMED is incremented by one and subjected to path failure detection, as defined in Section 8.2. When probing UNCONFIRMED addresses, however, the association overall error count is not incremented.

The number of packets containing HEARTBEAT chunks sent at each RTO **SHOULD** be limited by the 'HB.Max.Burst' parameter. It is an implementation decision as to how to distribute packets containing HEARTBEAT chunks to the peer's addresses for path verification.

Whenever a path is confirmed, an indication **MAY** be given to the upper layer.

An endpoint **MUST NOT** send any chunks to an UNCONFIRMED address, with the following exceptions:

- A HEARTBEAT chunk including a nonce **MAY** be sent to an UNCONFIRMED address.
- A HEARTBEAT ACK chunk **MAY** be sent to an UNCONFIRMED address.

- A COOKIE ACK chunk **MAY** be sent to an UNCONFIRMED address, but it **MUST** be bundled with a HEARTBEAT chunk including a nonce. An implementation that does not support bundling **MUST NOT** send a COOKIE ACK chunk to an UNCONFIRMED address.
- A COOKIE ECHO chunk **MAY** be sent to an UNCONFIRMED address, but it **MUST** be bundled with a HEARTBEAT chunk including a nonce, and the size of the SCTP packet **MUST NOT** exceed the PMTU. If the implementation does not support bundling or if the bundled COOKIE ECHO chunk plus HEARTBEAT chunk (including nonce) would result in an SCTP packet larger than the PMTU, then the implementation **MUST NOT** send a COOKIE ECHO chunk to an UNCONFIRMED address.

6. User Data Transfer

Data transmission **MUST** only happen in the ESTABLISHED, SHUTDOWN-PENDING, and SHUTDOWN-RECEIVED states. The only exception to this is that DATA chunks are allowed to be bundled with an outbound COOKIE ECHO chunk when in the COOKIE-WAIT state.

DATA chunks **MUST** only be received according to the rules below in ESTABLISHED, SHUTDOWN-PENDING, and SHUTDOWN-SENT states. A DATA chunk received in CLOSED is out of the blue and **SHOULD** be handled per Section 8.4. A DATA chunk received in any other state **SHOULD** be discarded.

A SACK chunk **MUST** be processed in ESTABLISHED, SHUTDOWN-PENDING, and SHUTDOWN-RECEIVED states. An incoming SACK chunk **MAY** be processed in COOKIE-ECHOED. A SACK chunk in the CLOSED state is out of the blue and **SHOULD** be processed according to the rules in Section 8.4. A SACK chunk received in any other state **SHOULD** be discarded.

For transmission efficiency, SCTP defines mechanisms for bundling of small user messages and fragmentation of large user messages. The following diagram depicts the flow of user messages through SCTP.

In this section, the term "data sender" refers to the endpoint that transmits a DATA chunk, and the term "data receiver" refers to the endpoint that receives a DATA chunk. A data receiver will transmit SACK chunks.

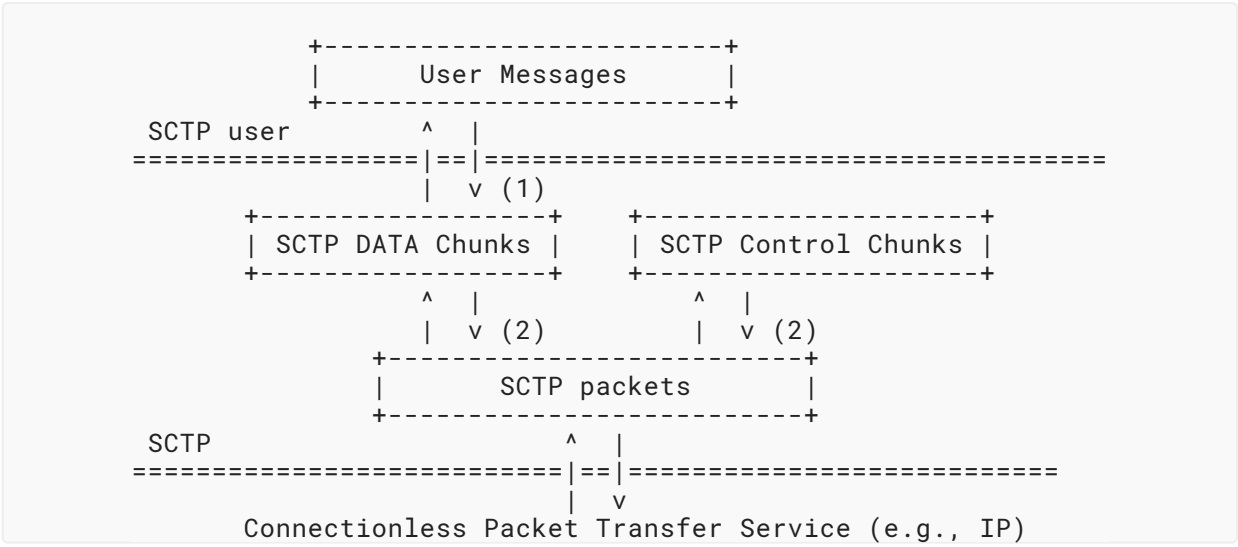


Figure 6: Illustration of User Data Transfer

The following applies:

1) When converting user messages into DATA chunks, an endpoint **MUST** fragment large user messages into multiple DATA chunks. The size of each DATA chunk **SHOULD** be smaller than or equal to the Association Maximum DATA Chunk Size (AMDCS). The data receiver will normally reassemble the fragmented message from DATA chunks before delivery to the user (see Section 6.9 for details).

2) Multiple DATA and control chunks **MAY** be bundled by the sender into a single SCTP packet for transmission, as long as the final size of the SCTP packet does not exceed the current PMTU. The receiver will unbundle the packet back into the original chunks. Control chunks **MUST** come before DATA chunks in the packet.

The fragmentation and bundling mechanisms, as detailed in Sections 6.9 and 6.10, are **OPTIONAL** to implement by the data sender, but they **MUST** be implemented by the data receiver, i.e., an endpoint **MUST** properly receive and process bundled or fragmented data.

6.1. Transmission of DATA Chunks

This section specifies the rules for sending DATA chunks. In particular, it defines zero window probing, which is required to avoid the indefinite stalling of an association in case of a loss of packets containing SACK chunks performing window updates.

This document is specified as if there is a single retransmission timer per destination transport address, but implementations **MAY** have a retransmission timer for each DATA chunk.

The following general rules **MUST** be applied by the data sender for transmission and/or retransmission of outbound DATA chunks:

A) At any given time, the data sender **MUST NOT** transmit new data to any destination transport address if its peer's rwnd indicates that the peer has no buffer space (i.e., rwnd is smaller than the size of the next DATA chunk; see Section 6.2.1), except for zero window probes.

A zero window probe is a DATA chunk sent when the receiver has no buffer space. This rule allows the sender to probe for a change in rwnd that the sender missed due to the SACK chunks having been lost in transit from the data receiver to the data sender. A zero window probe **MUST** only be sent when the cwnd allows (see rule B below). A zero window probe **SHOULD** only be sent when all outstanding DATA chunks have been cumulatively acknowledged and no DATA chunks are in flight. Senders **MUST** support zero window probing.

If the sender continues to receive SACK chunks from the peer while doing zero window probing, the unacknowledged window probes **SHOULD NOT** increment the error counter for the association or any destination transport address. This is because the receiver could keep its window closed for an indefinite time. Section 6.2 describes the receiver behavior when it advertises a zero window. The sender **SHOULD** send the first zero window probe after 1 RTO when it detects that the receiver has closed its window and **SHOULD** increase the probe interval exponentially afterwards. Also note that the cwnd **SHOULD** be adjusted according to Section 7.2.1. Zero window probing does not affect the calculation of cwnd.

The sender **MUST** also have an algorithm for sending new DATA chunks to avoid silly window syndrome (SWS) as described in [RFC1122]. The algorithm can be similar to the one described in Section 4.2.3.4 of [RFC1122].

B) At any given time, the sender **MUST NOT** transmit new data to a given transport address if it has cwnd + (PMDCS - 1) or more bytes of data outstanding to that transport address. If data is available, the sender **SHOULD** exceed cwnd by up to (PMDCS - 1) bytes on a new data transmission if the flightsize does not currently reach cwnd. The breach of cwnd **MUST** constitute one packet only.

C) When the time comes for the sender to transmit, before sending new DATA chunks, the sender **MUST** first transmit any DATA chunks that are marked for retransmission (limited by the current cwnd).

D) When the time comes for the sender to transmit new DATA chunks, the protocol parameter 'Max.Burst' **SHOULD** be used to limit the number of packets sent. The limit **MAY** be applied by adjusting cwnd temporarily, as follows:

```
if ((flightsize + Max.Burst * PMDCS) < cwnd)
    cwnd = flightsize + Max.Burst * PMDCS
```

Or, it **MAY** be applied by strictly limiting the number of packets emitted by the output routine. When calculating the number of packets to transmit, and particularly when using the formula above, cwnd **SHOULD NOT** be changed permanently.

E) Then, the sender can send as many new DATA chunks as rule A and rule B allow.

Multiple DATA chunks committed for transmission **MAY** be bundled in a single packet. Furthermore, DATA chunks being retransmitted **MAY** be bundled with new DATA chunks, as long as the resulting SCTP packet size does not exceed the PMTU. A ULP can request that no bundling is performed, but this only turns off any delays that an SCTP implementation might be using to increase bundling efficiency. It does not in itself stop all bundling from occurring (i.e., in case of congestion or retransmission).

Before an endpoint transmits a DATA chunk, if any received DATA chunks have not been acknowledged (e.g., due to delayed ack), the sender **SHOULD** create a SACK chunk and bundle it with the outbound DATA chunk, as long as the size of the final SCTP packet does not exceed the current PMTU. See Section 6.2.

When the window is full (i.e., transmission is disallowed by rule A and/or rule B), the sender **MAY** still accept send requests from its upper layer but **MUST** transmit no more DATA chunks until some or all of the outstanding DATA chunks are acknowledged and transmission is allowed by rule A and rule B again.

Whenever a transmission or retransmission is made to any address, if the T3-rtx timer of that address is not currently running, the sender **MUST** start that timer. If the timer for that address is already running, the sender **MUST** restart the timer if the earliest (i.e., lowest TSN) outstanding DATA chunk sent to that address is being retransmitted. Otherwise, the data sender **MUST NOT** restart the timer.

When starting or restarting the T3-rtx timer, the timer value **SHOULD** be adjusted according to the timer rules defined in Sections 6.3.2 and 6.3.3.

The data sender **MUST NOT** use a TSN that is more than $2^{31} - 1$ above the beginning TSN of the current send window.

For each stream, the data sender **MUST NOT** have more than $2^{16} - 1$ ordered user messages in the current send window.

Whenever the sender of a DATA chunk can benefit from the corresponding SACK chunk being sent back without delay, the sender **MAY** set the I bit in the DATA chunk header. Please note that why the sender has set the I bit is irrelevant to the receiver.

Reasons for setting the I bit include, but are not limited to, the following (see Section 4 of [RFC7053] for a discussion of the benefits):

- The application requests that the I bit of the last DATA chunk of a user message be set when providing the user message to the SCTP implementation (see Section 11.1).
- The sender is in the SHUTDOWN-PENDING state.
- The sending of a DATA chunk fills the congestion or receiver window.

6.2. Acknowledgement on Reception of DATA Chunks

The SCTP endpoint **MUST** always acknowledge the reception of each valid DATA chunk when the DATA chunk received is inside its receive window.

When the receiver's advertised window is 0, the receiver **MUST** drop any new incoming DATA chunk with a TSN larger than the largest TSN received so far. Also, if the new incoming DATA chunk holds a TSN value less than the largest TSN received so far, then the receiver **SHOULD** drop the largest TSN held for reordering and accept the new incoming DATA chunk. In either case, if such a DATA chunk is dropped, the receiver **MUST** immediately send back a SACK chunk with the current receive window showing only DATA chunks received and accepted so far. The dropped DATA chunk(s) **MUST NOT** be included in the SACK chunk, as they were not accepted. The receiver **MUST** also have an algorithm for advertising its receive window to avoid receiver silly window syndrome (SWS), as described in [RFC1122]. The algorithm can be similar to the one described in Section 4.2.3.3 of [RFC1122].

The guidelines on the delayed acknowledgement algorithm specified in Section 4.2 of [RFC5681] **SHOULD** be followed. Specifically, an acknowledgement **SHOULD** be generated for at least every second packet (not every second DATA chunk) received and **SHOULD** be generated within 200 ms of the arrival of any unacknowledged DATA chunk. In some situations, it might be beneficial for an SCTP transmitter to be more conservative than the algorithms detailed in this document allow. However, an SCTP transmitter **MUST NOT** be more aggressive in sending SACK chunks than the following algorithms allow.

An SCTP receiver **MUST NOT** generate more than one SACK chunk for every incoming packet, other than to update the offered window as the receiving application consumes new data. When the window opens up, an SCTP receiver **SHOULD** send additional SACK chunks to update the window even if no new data is received. The receiver **MUST** avoid sending a large number of window updates -- in particular, large bursts of them. One way to achieve this is to send a window update only if the window can be increased by at least a quarter of the receive buffer size of the association.

Implementation Note: The maximum delay for generating an acknowledgement **MAY** be configured by the SCTP administrator, either statically or dynamically, in order to meet the specific timing requirement of the protocol being carried.

An implementation **MUST NOT** allow the maximum delay (protocol parameter 'SACK.Delay') to be configured to be more than 500 ms. In other words, an implementation **MAY** lower the value of 'SACK.Delay' below 500 ms but **MUST NOT** raise it above 500 ms.

Acknowledgements **MUST** be sent in SACK chunks unless shutdown was requested by the ULP, in which case an endpoint **MAY** send an acknowledgement in the SHUTDOWN chunk. A SACK chunk can acknowledge the reception of multiple DATA chunks. See Section 3.3.4 for SACK chunk format. In particular, the SCTP endpoint **MUST** fill in the Cumulative TSN Ack field to indicate the latest sequential TSN (of a valid DATA chunk) it has received. Any received DATA chunks with TSN

greater than the value in the Cumulative TSN Ack field are reported in the Gap Ack Block fields. The SCTP endpoint **MUST** report as many Gap Ack Blocks as can fit in a single SACK chunk such that the size of the SCTP packet does not exceed the current PMTU.

The SHUTDOWN chunk does not contain Gap Ack Block fields. Therefore, the endpoint **SHOULD** use a SACK chunk instead of the SHUTDOWN chunk to acknowledge DATA chunks received out of order.

Upon receipt of an SCTP packet containing a DATA chunk with the I bit set, the receiver **SHOULD NOT** delay the sending of the corresponding SACK chunk, i.e., the receiver **SHOULD** immediately respond with the corresponding SACK chunk.

When a packet arrives with duplicate DATA chunk(s) and with no new DATA chunk(s), the endpoint **MUST** immediately send a SACK chunk with no delay. If a packet arrives with duplicate DATA chunk(s) bundled with new DATA chunks, the endpoint **MAY** immediately send a SACK chunk. Normally, receipt of duplicate DATA chunks will occur when the original SACK chunk was lost and the peer's RTO has expired. The duplicate TSN number(s) **SHOULD** be reported in the SACK chunk as duplicate.

When an endpoint receives a SACK chunk, it **MAY** use the duplicate TSN information to determine if SACK chunk loss is occurring. Further use of this data is for future study.

The data receiver is responsible for maintaining its receive buffers. The data receiver **SHOULD** notify the data sender in a timely manner of changes in its ability to receive data. How an implementation manages its receive buffers is dependent on many factors (e.g., operating system, memory management system, amount of memory, etc.). However, the data sender strategy defined in Section 6.2.1 is based on the assumption of receiver operation similar to the following:

A) At initialization of the association, the endpoint tells the peer how much receive buffer space it has allocated to the association in the INIT or INIT ACK chunk. The endpoint sets a_rwnd to this value.

B) As DATA chunks are received and buffered, decrement a_rwnd by the number of bytes received and buffered. This is, in effect, closing rwnd at the data sender and restricting the amount of data it can transmit.

C) As DATA chunks are delivered to the ULP and released from the receive buffers, increment a_rwnd by the number of bytes delivered to the upper layer. This is, in effect, opening up rwnd on the data sender and allowing it to send more data. The data receiver **SHOULD NOT** increment a_rwnd unless it has released bytes from its receive buffer. For example, if the receiver is holding fragmented DATA chunks in a reassembly queue, it **SHOULD NOT** increment a_rwnd.

D) When sending a SACK chunk, the data receiver **SHOULD** place the current value of a_rwnd into the a_rwnd field. The data receiver **SHOULD** take into account that the data sender will not retransmit DATA chunks that are acked via the Cumulative TSN Ack (i.e., will drop from its retransmit queue).

Under certain circumstances, the data receiver **MAY** drop DATA chunks that it has received but has not released from its receive buffers (i.e., delivered to the ULP). These DATA chunks might have been acked in Gap Ack Blocks. For example, the data receiver might be holding data in its receive buffers while reassembling a fragmented user message from its peer when it runs out of receive buffer space. It **MAY** drop these DATA chunks even though it has acknowledged them in Gap Ack Blocks. If a data receiver drops DATA chunks, it **MUST NOT** include them in Gap Ack Blocks in subsequent SACK chunks until they are received again via retransmission. In addition, the endpoint **SHOULD** take into account the dropped data when calculating its a_rwnd.

An endpoint **SHOULD NOT** revoke a SACK chunk and discard data. Only in extreme circumstances might an endpoint use this procedure (such as out of buffer space). The data receiver **SHOULD** take into account that dropping data that has been acked in Gap Ack Blocks can result in suboptimal retransmission strategies in the data sender and thus in suboptimal performance.

The following example illustrates the use of delayed acknowledgements:

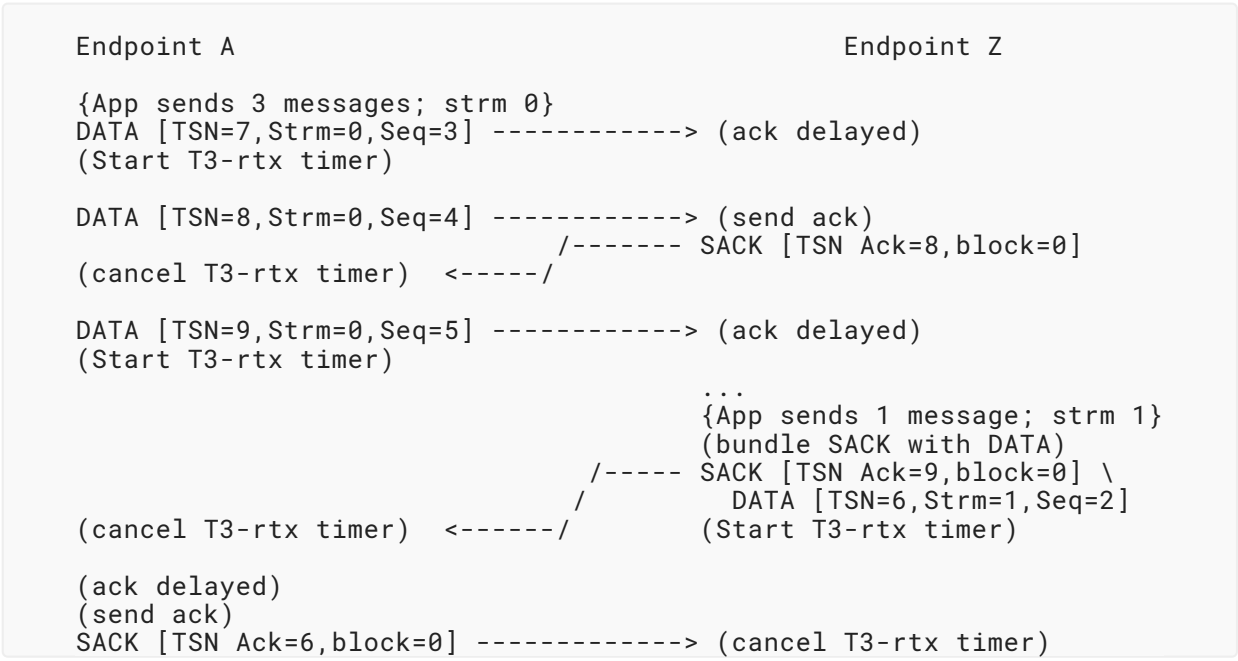


Figure 7: Delayed Acknowledgement Example

If an endpoint receives a DATA chunk with no user data (i.e., the Length field is set to 16), it **SHOULD** send an ABORT chunk with a "No User Data" error cause.

An endpoint **SHOULD NOT** send a DATA chunk with no user data part. This avoids the need to be able to return a zero-length user message in the API, especially in the socket API as specified in [RFC6458] for details.

6.2.1. Processing a Received SACK Chunk

Each SACK chunk an endpoint receives contains an a_rwnd value. This value represents the amount of buffer space the data receiver, at the time of transmitting the SACK chunk, has left of its total receive buffer space (as specified in the INIT/INIT ACK chunk). Using a_rwnd, Cumulative TSN Ack, and Gap Ack Blocks, the data sender can develop a representation of the peer's receive buffer space.

One of the problems the data sender takes into account when processing a SACK chunk is that a SACK chunk can be received out of order. That is, a SACK chunk sent by the data receiver can pass an earlier SACK chunk and be received first by the data sender. If a SACK chunk is received out of order, the data sender can develop an incorrect view of the peer's receive buffer space.

Since there is no explicit identifier that can be used to detect out-of-order SACK chunks, the data sender uses heuristics to determine if a SACK chunk is new.

An endpoint **SHOULD** use the following rules to calculate the rwnd, using the a_rwnd value, the Cumulative TSN Ack, and Gap Ack Blocks in a received SACK chunk.

A) At the establishment of the association, the endpoint initializes the rwnd to the Advertised Receiver Window Credit (a_rwnd) the peer specified in the INIT or INIT ACK chunk.

B) Any time a DATA chunk is transmitted (or retransmitted) to a peer, the endpoint subtracts the data size of the chunk from the rwnd of that peer.

C) Any time a DATA chunk is marked for retransmission, either via T3-rtx timer expiration (Section 6.3.3) or via Fast Retransmit (Section 7.2.4), add the data size of those chunks to the rwnd.

D) Any time a SACK chunk arrives, the endpoint performs the following:

   i) If Cumulative TSN Ack is less than the Cumulative TSN Ack Point, then drop the SACK chunk. Since Cumulative TSN Ack is monotonically increasing, a SACK chunk whose Cumulative TSN Ack is less than the Cumulative TSN Ack Point indicates an out-of-order SACK chunk.

   ii) Set rwnd equal to the newly received a_rwnd minus the number of bytes still outstanding after processing the Cumulative TSN Ack and the Gap Ack Blocks.

iii) If the SACK chunk is missing a TSN that was previously acknowledged via a Gap Ack Block (e.g., the data receiver reneged on the data), then consider the corresponding DATA that might be possibly missing: Count one miss indication towards Fast Retransmit as described in Section 7.2.4, and if no retransmit timer is running for the destination address to which the DATA chunk was originally transmitted, then T3-rtx is started for that destination address.

iv) If the Cumulative TSN Ack matches or exceeds the Fast Recovery exit point (Section 7.2.4), Fast Recovery is exited.

6.3. Management of Retransmission Timer

An SCTP endpoint uses a retransmission timer T3-rtx to ensure data delivery in the absence of any feedback from its peer. The duration of this timer is referred to as RTO (retransmission timeout).

When an endpoint's peer is multi-homed, the endpoint will calculate a separate RTO for each different destination transport address of its peer endpoint.

The computation and management of RTO in SCTP follow closely how TCP manages its retransmission timer. To compute the current RTO, an endpoint maintains two state variables per destination transport address: SRTT (smoothed round-trip time) and RTTVAR (round-trip time variation).

6.3.1. RTO Calculation

The rules governing the computation of SRTT, RTTVAR, and RTO are as follows:

C1) Until an RTT measurement has been made for a packet sent to the given destination transport address, set RTO to the protocol parameter 'RTO.Initial'.

C2) When the first RTT measurement R is made, perform:

```
SRTT = R
RTTVAR = R/2
RTO = SRTT + 4 * RTTVAR
```

C3) When a new RTT measurement R' is made, perform:

```
RTTVAR = (1 - RTO.Beta) * RTTVAR + RTO.Beta * |SRTT - R'|
SRTT = (1 - RTO.Alpha) * SRTT + RTO.Alpha * R'
```

Note: The value of SRTT used in the update to RTTVAR is its value before updating SRTT itself using the second assignment.

After the computation, update:

```
RTO = SRTT + 4 * RTTVAR
```

C4) When data is in flight and when allowed by rule C5 below, a new RTT measurement **MUST** be made each round trip. Furthermore, new RTT measurements **SHOULD** be made no more than once per round trip for a given destination transport address. There are two reasons for this recommendation: First, it appears that measuring more frequently often does not in practice yield any significant benefit [ALLMAN99]; second, if measurements are made more often, then the values of 'RTO.Alpha' and 'RTO.Beta' in rule C3 above **SHOULD** be adjusted so that SRTT and RTTVAR still adjust to changes at roughly the same rate (in terms of how many round trips it takes them to reflect new values) as they would if making only one measurement per round trip and using 'RTO.Alpha' and 'RTO.Beta' as given in rule C3. However, the exact nature of these adjustments remains a research issue.

C5) Karn's algorithm: RTT measurements **MUST NOT** be made using chunks that were retransmitted (and thus for which it is ambiguous whether the reply was for the first instance of the chunk or for a later instance).

RTT measurements **SHOULD** only be made using a DATA chunk with TSN r if no DATA chunk with TSN less than or equal to r was retransmitted since the DATA chunk with TSN r was sent first.

C6) Whenever RTO is computed, if it is less than 'RTO.Min' seconds, then it is rounded up to 'RTO.Min' seconds. The reason for this rule is that RTOs that do not have a high minimum value are susceptible to unnecessary timeouts [ALLMAN99].

C7) A maximum value **MAY** be placed on RTO, provided it is at least 'RTO.Max' seconds.

There is no requirement for the clock granularity G used for computing RTT measurements and the different state variables, other than:

G1) Whenever RTTVAR is computed, if RTTVAR == 0, then adjust RTTVAR = G.

Experience [ALLMAN99] has shown that finer clock granularities (less than 100 msec) perform somewhat better than more coarse granularities.

See Section 16 for suggested parameter values.

6.3.2. Retransmission Timer Rules

The rules for managing the retransmission timer are as follows:

R1) Every time a DATA chunk is sent to any address (including a retransmission), if the T3-rtx timer of that address is not running, start it running so that it will expire after the RTO of that address. The RTO used here is that obtained after any doubling due to previous T3-rtx timer expirations on the corresponding destination address as discussed in rule E2 below.

R2) Whenever all outstanding data sent to an address have been acknowledged, turn off the T3-rtx timer of that address.

R3) Whenever a SACK chunk is received that acknowledges the DATA chunk with the earliest outstanding TSN for that address, restart the T3-rtx timer for that address with its current RTO (if there is still outstanding data on that address).

R4) Whenever a SACK chunk is received missing a TSN that was previously acknowledged via a Gap Ack Block, start the T3-rtx for the destination address to which the DATA chunk was originally transmitted if it is not already running.

The following example shows the use of various timer rules (assuming that the receiver uses delayed acks).

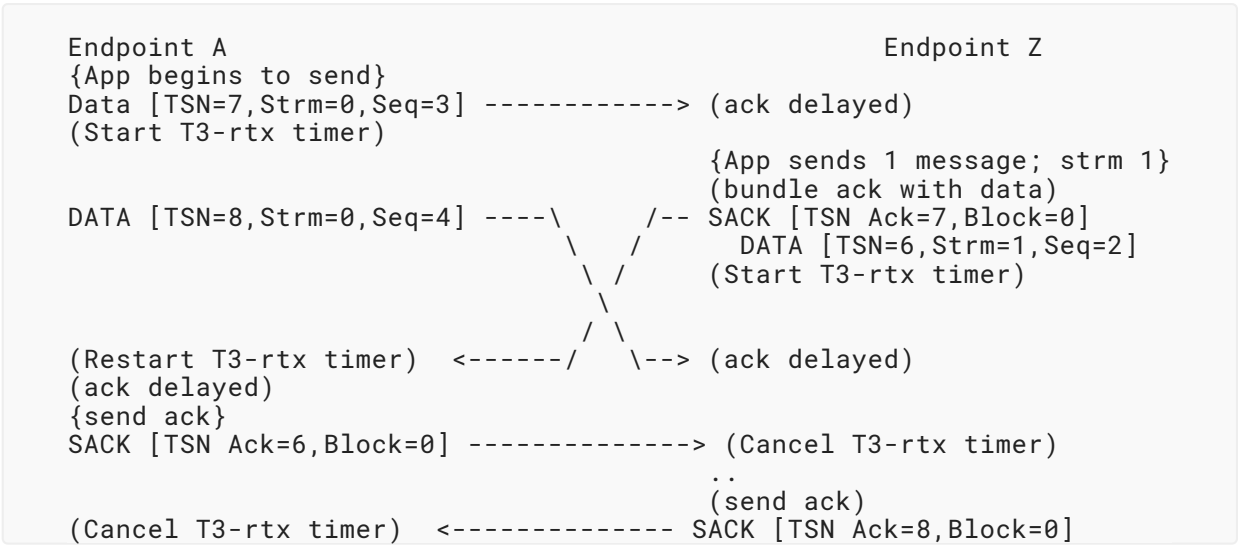


Figure 8: Timer Rule Examples

6.3.3. Handle T3-rtx Expiration

Whenever the retransmission timer T3-rtx expires for a destination address, do the following:

E1) For the destination address for which the timer expires, adjust its ssthresh with rules defined in Section 7.2.3 and set cwnd = PMDCS.

E2) For the destination address for which the timer expires, set RTO = RTO * 2 ("back off the timer"). The maximum value discussed in rule C7 above ('RTO.Max') **MAY** be used to provide an upper bound to this doubling operation.

E3) Determine how many of the earliest (i.e., lowest TSN) outstanding DATA chunks for the address for which the T3-rtx has expired will fit into a single SCTP packet, subject to the PMTU corresponding to the destination transport address to which the retransmission is being sent (this might be different from the address for which the timer expires; see Section 6.4). Call this value K. Bundle and retransmit those K DATA chunks in a single packet to the destination endpoint.

E4) Start the retransmission timer T3-rtx on the destination address to which the retransmission is sent if rule R1 above indicates to do so. The RTO to be used for starting T3-rtx **SHOULD** be the one for the destination address to which the retransmission is sent, which, when the receiver is multi-homed, might be different from the destination address for which the timer expired (see Section 6.4 below).

After retransmitting, once a new RTT measurement is obtained (which can happen only when new data has been sent and acknowledged, per rule C5, or for a measurement made from a HEARTBEAT chunk; see Section 8.3), the computation in rule C3 is performed, including the computation of RTO, which might result in "collapsing" RTO back down after it has been subject to doubling (rule E2).

Any DATA chunks that were sent to the address for which the T3-rtx timer expired but did not fit in an SCTP packet of size smaller than or equal to the PMTU (rule E3 above) **SHOULD** be marked for retransmission and sent as soon as cwnd allows (normally, when a SACK chunk arrives).

The final rule for managing the retransmission timer concerns failover (see Section 6.4.1):

F1) Whenever an endpoint switches from the current destination transport address to a different one, the current retransmission timers are left running. As soon as the endpoint transmits a packet containing DATA chunk(s) to the new transport address, start the timer on that transport address, using the RTO value of the destination address to which the data is being sent, if rule R1 indicates to do so.

6.4. Multi-Homed SCTP Endpoints

An SCTP endpoint is considered multi-homed if there is more than one transport address that can be used as a destination address to reach that endpoint.

Moreover, the ULP of an endpoint selects one of the multiple destination addresses of a multi-homed peer endpoint as the primary path (see Sections 5.1.2 and 11.1 for details).

By default, an endpoint **SHOULD** always transmit to the primary path, unless the SCTP user explicitly specifies the destination transport address (and possibly source transport address) to use.

An endpoint **SHOULD** transmit reply chunks (e.g., INIT ACK, COOKIE ACK, and HEARTBEAT ACK) in response to control chunks to the same destination transport address from which it received the control chunk to which it is replying.

The selection of the destination transport address for packets containing SACK chunks is implementation dependent. However, an endpoint **SHOULD NOT** vary the destination transport address of a SACK chunk when it receives DATA chunks coming from the same source address.

When acknowledging multiple DATA chunks received in packets from different source addresses in a single SACK chunk, the SACK chunk **MAY** be transmitted to one of the destination transport addresses from which the DATA or control chunks being acknowledged were received.

When a receiver of a duplicate DATA chunk sends a SACK chunk to a multi-homed endpoint, it **MAY** be beneficial to vary the destination address and not use the source address of the DATA chunk. The reason is that receiving a duplicate from a multi-homed endpoint might indicate that the return path (as specified in the source address of the DATA chunk) for the SACK chunk is broken.

Furthermore, when its peer is multi-homed, an endpoint **SHOULD** try to retransmit a chunk that timed out to an active destination transport address that is different from the last destination address to which the chunk was sent.

When its peer is multi-homed, an endpoint **SHOULD** send fast retransmissions to the same destination transport address to which the original data was sent. If the primary path has been changed and the original data was sent to the old primary path before the Fast Retransmit, the implementation **MAY** send it to the new primary path.

Retransmissions do not affect the total outstanding data count. However, if the DATA chunk is retransmitted onto a different destination address, both the outstanding data counts on the new destination address and the old destination address to which the data chunk was last sent is adjusted accordingly.

6.4.1. Failover from an Inactive Destination Address

Some of the transport addresses of a multi-homed SCTP endpoint might become inactive due to either the occurrence of certain error conditions (see Section 8.2) or adjustments from the SCTP user.

When there is outbound data to send and the primary path becomes inactive (e.g., due to failures) or where the SCTP user explicitly requests to send data to an inactive destination transport address before reporting an error to its ULP, the SCTP endpoint **SHOULD** try to send the data to an alternate active destination transport address if one exists.

When retransmitting data that timed out, if the endpoint is multi-homed, it needs to consider each source-destination address pair in its retransmission selection policy. When retransmitting timed-out data, the endpoint **SHOULD** attempt to pick the most divergent source-destination pair from the original source-destination pair to which the packet was transmitted.

Note: Rules for picking the most divergent source-destination pair are an implementation decision and are not specified within this document.

6.5. Stream Identifier and Stream Sequence Number

Every DATA chunk **MUST** carry a valid stream identifier. If an endpoint receives a DATA chunk with an invalid stream identifier, it **SHOULD** acknowledge the reception of the DATA chunk following the normal procedure, immediately send an ERROR chunk with cause set to "Invalid Stream Identifier" (see Section 3.3.10), and discard the DATA chunk. The endpoint **MAY** bundle the ERROR chunk and the SACK chunk in the same packet.

The Stream Sequence Number in all the outgoing streams **MUST** start from 0 when the association is established. The Stream Sequence Number of an outgoing stream **MUST** be incremented by 1 for each ordered user message sent on that outgoing stream. In particular, when the Stream Sequence Number reaches the value 65535, the next Stream Sequence Number **MUST** be set to 0. For unordered user messages, the Stream Sequence Number **MUST NOT** be changed.

6.6. Ordered and Unordered Delivery

Within a stream, an endpoint **MUST** deliver DATA chunks received with the U flag set to 0 to the upper layer according to the order of their Stream Sequence Number. If DATA chunks arrive out of order of their Stream Sequence Number, the endpoint **MUST** hold the received DATA chunks from delivery to the ULP until they are reordered.

However, an SCTP endpoint can indicate that no ordered delivery is required for a particular DATA chunk transmitted within the stream by setting the U flag of the DATA chunk to 1.

When an endpoint receives a DATA chunk with the U flag set to 1, it bypasses the ordering mechanism and immediately deliver the data to the upper layer (after reassembly if the user data is fragmented by the data sender).

This provides an effective way of transmitting "out-of-band" data in a given stream. Also, a stream can be used as an "unordered" stream by simply setting the U flag to 1 in all DATA chunks sent through that stream.

Implementation Note: When sending an unordered DATA chunk, an implementation **MAY** choose to place the DATA chunk in an outbound packet that is at the head of the outbound transmission queue if possible.

The 'Stream Sequence Number' field in a DATA chunk with U flag set to 1 has no significance. The sender can fill the 'Stream Sequence Number' with arbitrary value, but the receiver **MUST** ignore the field.

Note: When transmitting ordered and unordered data, an endpoint does not increment its Stream Sequence Number when transmitting a DATA chunk with U flag set to 1.

6.7. Report Gaps in Received DATA TSNs

Upon the reception of a new DATA chunk, an endpoint examines the continuity of the TSNs received. If the endpoint detects a gap in the received DATA chunk sequence, it **SHOULD** send a SACK chunk with Gap Ack Blocks immediately. The data receiver continues sending a SACK chunk after receipt of each SCTP packet that does not fill the gap.

Based on the Gap Ack Block from the received SACK chunk, the endpoint can calculate the missing DATA chunks and make decisions on whether to retransmit them (see Section 6.2.1 for details).

Multiple gaps can be reported in one single SACK chunk (see Section 3.3.4).

When its peer is multi-homed, the SCTP endpoint **SHOULD** always try to send the SACK chunk to the same destination address from which the last DATA chunk was received.

Upon the reception of a SACK chunk, the endpoint **MUST** remove all DATA chunks that have been acknowledged by the SACK chunk's Cumulative TSN Ack from its transmit queue. All DATA chunks with TSNs not included in the Gap Ack Blocks that are smaller than the highest-acknowledged TSN reported in the SACK chunk **MUST** be treated as "missing" by the sending endpoint. The number of "missing" reports for each outstanding DATA chunk **MUST** be recorded by the data sender to make retransmission decisions. See Section 7.2.4 for details.

The following example shows the use of SACK chunk to report a gap.

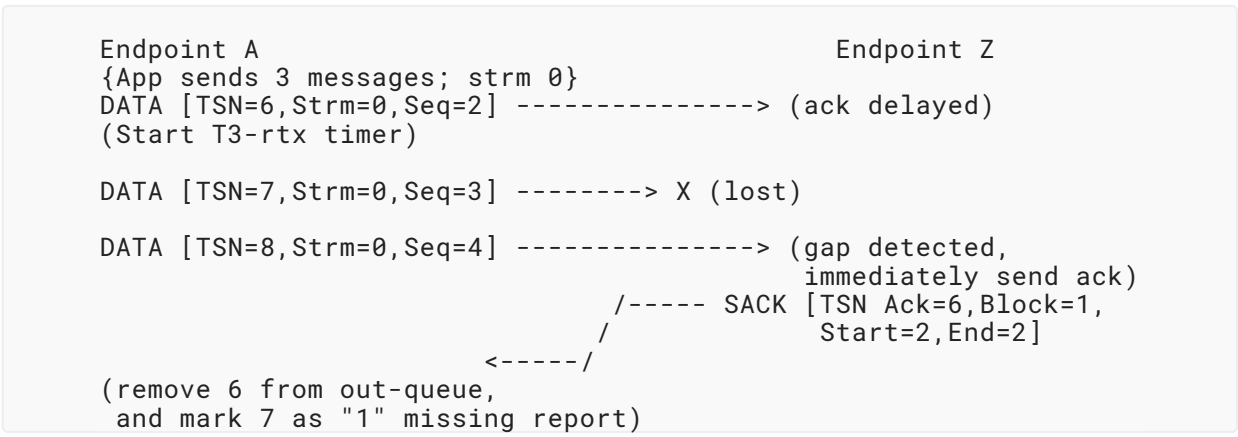


Figure 9: Reporting a Gap Using SACK Chunk

The maximum number of Gap Ack Blocks that can be reported within a single SACK chunk is limited by the current PMTU. When a single SACK chunk cannot cover all the Gap Ack Blocks needed to be reported due to the PMTU limitation, the endpoint **MUST** send only one SACK chunk. This single SACK chunk **MUST** report the Gap Ack Blocks from the lowest to highest TSNs, within the size limit set by the PMTU, and leave the remaining highest TSN numbers unacknowledged.

6.8. CRC32c Checksum Calculation

When sending an SCTP packet, the endpoint **MUST** strengthen the data integrity of the transmission by including the CRC32c checksum value calculated on the packet, as described below.

After the packet is constructed (containing the SCTP common header and one or more control or DATA chunks), the transmitter **MUST**:

1) fill in the proper Verification Tag in the SCTP common header and initialize the checksum field to 0,

2) calculate the CRC32c checksum of the whole packet, including the SCTP common header and all the chunks (refer to Appendix A for details of the CRC32c algorithm), and

3) put the resultant value into the checksum field in the common header and leave the rest of the bits unchanged.

When an SCTP packet is received, the receiver **MUST** first check the CRC32c checksum as follows:

1) Store the received CRC32c checksum value aside.

2) Replace the 32 bits of the checksum field in the received SCTP packet with 0 and calculate a CRC32c checksum value of the whole received packet.

3) Verify that the calculated CRC32c checksum is the same as the received CRC32c checksum. If it is not, the receiver **MUST** treat the packet as an invalid SCTP packet.

The default procedure for handling invalid SCTP packets is to silently discard them.

Any hardware implementation **SHOULD** permit alternative verification of the CRC in software.

6.9. Fragmentation and Reassembly

An endpoint **MAY** support fragmentation when sending DATA chunks, but it **MUST** support reassembly when receiving DATA chunks. If an endpoint supports fragmentation, it **MUST** fragment a user message if the size of the user message to be sent causes the outbound SCTP packet size to exceed the current PMTU. An endpoint that does not support fragmentation and is requested to send a user message such that the outbound SCTP packet size would exceed the current PMTU **MUST** return an error to its upper layer and **MUST NOT** attempt to send the user message.

An SCTP implementation **MAY** provide a mechanism to the upper layer that disables fragmentation when sending DATA chunks. When fragmentation of DATA chunks is disabled, the SCTP implementation **MUST** behave in the same way an implementation that does not support fragmentation, i.e., it rejects calls that would result in sending SCTP packets that exceed the current PMTU.

Implementation Note: In this error case, the SEND primitive discussed in Section 11.1.5 would need to return an error to the upper layer.

If its peer is multi-homed, the endpoint **SHOULD** choose a DATA chunk size smaller than or equal to the AMDCS.

Once a user message is fragmented, it cannot be re-fragmented. Instead, if the PMTU has been reduced, then IP fragmentation **MUST** be used. Therefore, an SCTP association can fail if IP fragmentation is not working on any path. Please see Section 7.3 for details of PMTU discovery.

When determining when to fragment, the SCTP implementation **MUST** take into account the SCTP packet header as well as the DATA chunk header(s). The implementation **MUST** also take into account the space required for a SACK chunk if bundling a SACK chunk with the DATA chunk.

Fragmentation takes the following steps:

1) The data sender **MUST** break the user message into a series of DATA chunks. The sender **SHOULD** choose a size of DATA chunks that is smaller than or equal to the AMDCS.

2) The transmitter **MUST** then assign, in sequence, a separate TSN to each of the DATA chunks in the series. The transmitter assigns the same Stream Sequence Number to each of the DATA chunks. If the user indicates that the user message is to be delivered using unordered delivery, then the U flag of each DATA chunk of the user message **MUST** be set to 1.

3) The transmitter **MUST** also set the B/E bits of the first DATA chunk in the series to 10, the B/E bits of the last DATA chunk in the series to 01, and the B/E bits of all other DATA chunks in the series to 00.

An endpoint **MUST** recognize fragmented DATA chunks by examining the B/E bits in each of the received DATA chunks and queue the fragmented DATA chunks for reassembly. Once the user message is reassembled, SCTP passes the reassembled user message to the specific stream for possible reordering and final dispatching.

If the data receiver runs out of buffer space while still waiting for more fragments to complete the reassembly of the message, it **SHOULD** dispatch part of its inbound message through a partial delivery API (see Section 11), freeing some of its receive buffer space so that the rest of the message can be received.

6.10. Bundling

An endpoint bundles chunks by simply including multiple chunks in one outbound SCTP packet. The total size of the resultant SCTP packet **MUST** be less that or equal to the current PMTU.

If its peer endpoint is multi-homed, the sending endpoint **SHOULD** choose a size no larger than the PMTU of the current primary path.

When bundling control chunks with DATA chunks, an endpoint **MUST** place control chunks first in the outbound SCTP packet. The transmitter **MUST** transmit DATA chunks within an SCTP packet in increasing order of TSN.

Note: Since control chunks are placed first in a packet and since DATA chunks are transmitted before SHUTDOWN or SHUTDOWN ACK chunks, DATA chunks cannot be bundled with SHUTDOWN or SHUTDOWN ACK chunks.

Partial chunks **MUST NOT** be placed in an SCTP packet. A partial chunk is a chunk that is not completely contained in the SCTP packet; i.e., the SCTP packet is too short to contain all the bytes of the chunk as indicated by the chunk length.

An endpoint **MUST** process received chunks in their order in the packet. The receiver uses the Chunk Length field to determine the end of a chunk and beginning of the next chunk, taking account of the fact that all chunks end on a 4-byte boundary. If the receiver detects a partial chunk, it **MUST** drop the chunk.

An endpoint **MUST NOT** bundle INIT, INIT ACK, or SHUTDOWN COMPLETE chunks with any other chunks.

7. Congestion Control

Congestion control is one of the basic functions in SCTP. To manage congestion, the mechanisms and algorithms in this section are to be employed.

Implementation Note: As far as its specific performance requirements are met, an implementation is always allowed to adopt a more conservative congestion control algorithm than the one defined below.

The congestion control algorithms used by SCTP are based on [RFC5681]. This section describes how the algorithms defined in [RFC5681] are adapted for use in SCTP. We first list differences in protocol designs between TCP and SCTP and then describe SCTP's congestion control scheme. The description will use the same terminology as in TCP congestion control whenever appropriate.

SCTP congestion control is always applied to the entire association and not to individual streams.

7.1. SCTP Differences from TCP Congestion Control

Gap Ack Blocks in the SCTP SACK chunk carry the same semantic meaning as the TCP SACK. TCP considers the information carried in the SACK as advisory information only. SCTP considers the information carried in the Gap Ack Blocks in the SACK chunk as advisory. In SCTP, any DATA chunk that has been acknowledged by a SACK chunk, including DATA that arrived at the receiving end out of order, is not considered fully delivered until the Cumulative TSN Ack Point passes the TSN of the DATA chunk (i.e., the DATA chunk has been acknowledged by the Cumulative TSN Ack field in the SACK chunk). Consequently, the value of cwnd controls the amount of outstanding

data, rather than (as in the case of non-SACK TCP) the upper bound between the highest acknowledged sequence number and the latest DATA chunk that can be sent within the congestion window. SCTP SACK leads to different implementations of Fast Retransmit and Fast Recovery than non-SACK TCP. As an example, see [FALL96].

The biggest difference between SCTP and TCP, however, is multi-homing. SCTP is designed to establish robust communication associations between two endpoints, each of which might be reachable by more than one transport address. Potentially different addresses might lead to different data paths between the two endpoints; thus, ideally, one needs a separate set of congestion control parameters for each of the paths. The treatment here of congestion control for multi-homed receivers is new with SCTP and might require refinement in the future. The current algorithms make the following assumptions:

- The sender usually uses the same destination address until being instructed by the upper layer to do otherwise; however, SCTP **MAY** change to an alternate destination in the event an address is marked inactive (see Section 8.2). Also, SCTP **MAY** retransmit to a different transport address than the original transmission.
- The sender keeps a separate congestion control parameter set for each of the destination addresses it can send to (not each source-destination pair but for each destination). The parameters **SHOULD** decay if the address is not used for a long enough time period. [RFC5681] specifies this period of time as a retransmission timeout.
- For each of the destination addresses, an endpoint does slow start upon the first transmission to that address.

Note: TCP guarantees in-sequence delivery of data to its upper-layer protocol within a single TCP session. This means that when TCP notices a gap in the received sequence number, it waits until the gap is filled before delivering the data that was received with sequence numbers higher than that of the missing data. On the other hand, SCTP can deliver data to its upper-layer protocol, even if there is a gap in TSN if the Stream Sequence Numbers are in sequence for a particular stream (i.e., the missing DATA chunks are for a different stream) or if unordered delivery is indicated. Although this does not affect cwnd, it might affect rwnd calculation.

7.2. SCTP Slow-Start and Congestion Avoidance

The slow-start and congestion avoidance algorithms **MUST** be used by an endpoint to control the amount of data being injected into the network. The congestion control in SCTP is employed in regard to the association, not to an individual stream. In some situations, it might be beneficial for an SCTP sender to be more conservative than the algorithms allow; however, an SCTP sender **MUST NOT** be more aggressive than the following algorithms allow.

Like TCP, an SCTP endpoint uses the following three control variables to regulate its transmission rate.

- Receiver advertised window size (rwnd, in bytes), which is set by the receiver based on its available buffer space for incoming packets.

  Note: This variable is kept on the entire association.

- Congestion control window (cwnd, in bytes), which is adjusted by the sender based on observed network conditions.

  Note: This variable is maintained on a per-destination-address basis.
- Slow-start threshold (ssthresh, in bytes), which is used by the sender to distinguish slow-start and congestion avoidance phases.

  Note: This variable is maintained on a per-destination-address basis.

SCTP also requires one additional control variable, partial_bytes_acked, which is used during the congestion avoidance phase to facilitate cwnd adjustment.

Unlike TCP, an SCTP sender **MUST** keep a set of the control variables cwnd, ssthresh, and partial_bytes_acked for EACH destination address of its peer (when its peer is multi-homed). When calculating one of these variables, the length of the DATA chunk, including the padding, **SHOULD** be used.

Only one rwnd is kept for the whole association (no matter if the peer is multi-homed or has a single address).

7.2.1. Slow-Start

Beginning data transmission into a network with unknown conditions or after a sufficiently long idle period requires SCTP to probe the network to determine the available capacity. The slow-start algorithm is used for this purpose at the beginning of a transfer or after repairing loss detected by the retransmission timer.

- The initial cwnd before data transmission **MUST** be set to min(4 * PMDCS, max(2 * PMDCS, 4404)) bytes if the peer address is an IPv4 address and to min(4 * PMDCS, max(2 * PMDCS, 4344)) bytes if the peer address is an IPv6 address.
- The initial cwnd after a retransmission timeout **MUST** be no more than PMDCS, and only one packet is allowed to be in flight until successful acknowledgement.
- The initial value of ssthresh **SHOULD** be arbitrarily high (e.g., the size of the largest-possible advertised window).
- Whenever cwnd is greater than zero, the endpoint is allowed to have cwnd bytes of data outstanding on that transport address. A limited overbooking as described in rule B in Section 6.1 **SHOULD** be supported.
- When cwnd is less than or equal to ssthresh, an SCTP endpoint **MUST** use the slow-start algorithm to increase cwnd only if the current congestion window is being fully utilized and the data sender is not in Fast Recovery. Only when these two conditions are met can the cwnd be increased; otherwise, the cwnd **MUST NOT** be increased. If these conditions are met, then cwnd **MUST** be increased by, at most, the lesser of

  1. the total size of the previously outstanding DATA chunk(s) acknowledged and
  2. L times the destination's PMDCS.

The first upper bound protects against the ACK-Splitting attack outlined in [SAVAGE99]. The positive integer L **SHOULD** be 1 and **MAY** be larger than 1. See [RFC3465] for details of choosing L.

In instances where its peer endpoint is multi-homed, if an endpoint receives a SACK chunk that results in updating the cwnd, then it **SHOULD** update its cwnd (or cwnds) apportioned to the destination addresses to which it transmitted the acknowledged data.

- While the endpoint does not transmit data on a given transport address, the cwnd of the transport address **SHOULD** be adjusted to max(cwnd / 2, 4 * PMDCS) once per RTO. Before the first cwnd adjustment, the ssthresh of the transport address **SHOULD** be set to the cwnd.

7.2.2. Congestion Avoidance

When cwnd is greater than ssthresh, cwnd **SHOULD** be incremented by PMDCS per RTT if the sender has cwnd or more bytes of data outstanding for the corresponding transport address. The basic recommendations for incrementing cwnd during congestion avoidance are as follows:

- SCTP **MAY** increment cwnd by PMDCS.
- SCTP **SHOULD** increment cwnd by PMDCS once per RTT when the sender has cwnd or more bytes of data outstanding for the corresponding transport address.
- SCTP **MUST NOT** increment cwnd by more than PMDCS per RTT.

In practice, an implementation can achieve this goal in the following way:

- partial_bytes_acked is initialized to 0.
- Whenever cwnd is greater than ssthresh, upon each SACK chunk arrival, increase partial_bytes_acked by the total number of bytes (including the chunk header and the padding) of all new DATA chunks acknowledged in that SACK chunk, including chunks acknowledged by the new Cumulative TSN Ack, by Gap Ack Blocks, and by the number of bytes of duplicated chunks reported in Duplicate TSNs.
- When (1) partial_bytes_acked is greater than cwnd and (2) before the arrival of the SACK chunk the sender had less than cwnd bytes of data outstanding (i.e., before the arrival of the SACK chunk, flightsize was less than cwnd), reset partial_bytes_acked to cwnd.
- When (1) partial_bytes_acked is equal to or greater than cwnd and (2) before the arrival of the SACK chunk the sender had cwnd or more bytes of data outstanding (i.e., before the arrival of the SACK chunk, flightsize was greater than or equal to cwnd), partial_bytes_acked is reset to (partial_bytes_acked - cwnd). Next, cwnd is increased by PMDCS.
- Same as in the slow start, when the sender does not transmit DATA chunks on a given transport address, the cwnd of the transport address **SHOULD** be adjusted to max(cwnd / 2, 4 * PMDCS) per RTO.
- When all of the data transmitted by the sender has been acknowledged by the receiver, partial_bytes_acked is initialized to 0.

7.2.3. Congestion Control

Upon detection of packet losses from SACK chunks (see Section 7.2.4), an endpoint **SHOULD** do the following:

```
ssthresh = max(cwnd / 2, 4 * PMDCS)
cwnd = ssthresh
partial_bytes_acked = 0
```

Basically, a packet loss causes cwnd to be cut in half.

When the T3-rtx timer expires on an address, SCTP **SHOULD** perform slow start by:

```
ssthresh = max(cwnd / 2, 4 * PMDCS)
cwnd = PMDCS
partial_bytes_acked = 0
```

and ensure that no more than one SCTP packet will be in flight for that address until the endpoint receives acknowledgement for successful delivery of data to that address.

7.2.4. Fast Retransmit on Gap Reports

In the absence of data loss, an endpoint performs delayed acknowledgement. However, whenever an endpoint notices a hole in the arriving TSN sequence, it **SHOULD** start sending a SACK chunk back every time a packet arrives carrying data until the hole is filled.

Whenever an endpoint receives a SACK chunk that indicates that some TSNs are missing, it **SHOULD** wait for two further miss indications (via subsequent SACK chunks for a total of three missing reports) on the same TSNs before taking action with regard to Fast Retransmit.

Miss indications **SHOULD** follow the Highest TSN Newly Acknowledged (HTNA) algorithm. For each incoming SACK chunk, miss indications are incremented only for missing TSNs prior to the HTNA in the SACK chunk. A newly acknowledged DATA chunk is one not previously acknowledged in a SACK chunk. If an endpoint is in Fast Recovery and a SACK chunks arrives that advances the Cumulative TSN Ack Point, the miss indications are incremented for all TSNs reported missing in the SACK chunk.

When the third consecutive miss indication is received for one or more TSNs, the data sender does the following:

1) Mark the DATA chunk(s) with three miss indications for retransmission.

2) If not in Fast Recovery, adjust the ssthresh and cwnd of the destination address(es) to which the missing DATA chunks were last sent, according to the formula described in Section 7.2.3.

3) If not in Fast Recovery, determine how many of the earliest (i.e., lowest TSN) DATA chunks marked for retransmission will fit into a single packet, subject to constraint of the PMTU of the destination transport address to which the packet is being sent. Call this value K. Retransmit those K DATA chunks in a single packet. When a Fast Retransmit is being performed, the sender **SHOULD** ignore the value of cwnd and **SHOULD NOT** delay retransmission for this single packet.

4) Restart the T3-rtx timer only if the last SACK chunk acknowledged the lowest outstanding TSN number sent to that address or the endpoint is retransmitting the first outstanding DATA chunk sent to that address.

5) Mark the DATA chunk(s) as being fast retransmitted and thus ineligible for a subsequent Fast Retransmit. Those TSNs marked for retransmission due to the Fast-Retransmit algorithm that did not fit in the sent datagram carrying K other TSNs are also marked as ineligible for a subsequent Fast Retransmit. However, as they are marked for retransmission, they will be retransmitted later on as soon as cwnd allows.

6) If not in Fast Recovery, enter Fast Recovery and mark the highest outstanding TSN as the Fast Recovery exit point. When a SACK chunk acknowledges all TSNs up to and including this exit point, Fast Recovery is exited. While in Fast Recovery, the ssthresh and cwnd **SHOULD NOT** change for any destinations due to a subsequent Fast Recovery event (i.e., one **SHOULD NOT** reduce the cwnd further due to a subsequent Fast Retransmit).

Note: Before the above adjustments, if the received SACK chunk also acknowledges new DATA chunks and advances the Cumulative TSN Ack Point, the cwnd adjustment rules defined in Sections 7.2.1 and 7.2.2 **MUST** be applied first.

7.2.5. Reinitialization

During the lifetime of an SCTP association, events can happen that result in using the network under unknown new conditions. When detected by an SCTP implementation, the congestion control **MUST** be reinitialized.

7.2.5.1. Change of Differentiated Services Code Points

SCTP implementations **MAY** allow an application to configure the Differentiated Services Code Point (DSCP) used for sending packets. If a DSCP change might result in outgoing packets being queued in different queues, the congestion control parameters for all affected destination addresses **MUST** be reset to their initial values.

7.2.5.2. Change of Routes

SCTP implementations **MAY** be aware of routing changes affecting packets sent to a destination address. In particular, this includes the selection of a different source address used for sending packets to a destination address. If such a routing change happens, the congestion control parameters for the affected destination addresses **MUST** be reset to their initial values.

7.3. PMTU Discovery

[RFC8899], [RFC8201], and [RFC1191] specify "Packetization Layer Path MTU Discovery", whereby an endpoint maintains an estimate of PMTU along a given Internet path and refrains from sending packets along that path that exceed the PMTU, other than occasional attempts to probe for a change in the PMTU. [RFC8899] is thorough in its discussion of the PMTU discovery mechanism and strategies for determining the current end-to-end PMTU setting as well as detecting changes in this value.

An endpoint **SHOULD** apply these techniques and **SHOULD** do so on a per-destination-address basis.

There are two important SCTP-specific points regarding PMTU discovery:

1) SCTP associations can span multiple addresses. An endpoint **MUST** maintain separate PMTU estimates for each destination address of its peer.

2) The sender **SHOULD** track an AMDCS that will be the smallest PMDCS discovered for all of the peer's destination addresses. When fragmenting messages into multiple parts, this AMDCS **SHOULD** be used to calculate the size of each DATA chunk. This will allow retransmissions to be seamlessly sent to an alternate address without encountering IP fragmentation.

8. Fault Management

8.1. Endpoint Failure Detection

An endpoint **SHOULD** keep a counter on the total number of consecutive retransmissions to its peer (this includes data retransmissions to all the destination transport addresses of the peer if it is multi-homed), including the number of unacknowledged HEARTBEAT chunks observed on the path that is currently used for data transfer. Unacknowledged HEARTBEAT chunks observed on paths different from the path currently used for data transfer **SHOULD NOT** increment the association error counter, as this could lead to association closure even if the path that is currently used for data transfer is available (but idle). If the value of this counter exceeds the limit indicated in the protocol parameter 'Association.Max.Retrans', the endpoint **SHOULD** consider the peer endpoint unreachable and **SHALL** stop transmitting any more data to it (and thus the association enters the CLOSED state). In addition, the endpoint **SHOULD** report the failure to the upper layer and optionally report back all outstanding user data remaining in its outbound queue. The association is automatically closed when the peer endpoint becomes unreachable.

The counter used for endpoint failure detection **MUST** be reset each time a DATA chunk sent to that peer endpoint is acknowledged (by the reception of a SACK chunk). When a HEARTBEAT ACK chunk is received from the peer endpoint, the counter **SHOULD** also be reset. The receiver of the HEARTBEAT ACK chunk **MAY** choose not to clear the counter if there is outstanding data on the association. This allows for handling the possible difference in reachability based on DATA chunks and HEARTBEAT chunks.

8.2. Path Failure Detection

When its peer endpoint is multi-homed, an endpoint **SHOULD** keep an error counter for each of the destination transport addresses of the peer endpoint.

Each time the T3-rtx timer expires on any address, or when a HEARTBEAT chunk sent to an idle address is not acknowledged within an RTO, the error counter of that destination address will be incremented. When the value in the error counter exceeds the protocol parameter 'Path.Max.Retrans' of that destination address, the endpoint **SHOULD** mark the destination transport address as inactive, and a notification **SHOULD** be sent to the upper layer.

When an outstanding TSN is acknowledged or a HEARTBEAT chunk sent to that address is acknowledged with a HEARTBEAT ACK chunk, the endpoint **SHOULD** clear the error counter of the destination transport address to which the DATA chunk was last sent (or HEARTBEAT chunk was sent) and **SHOULD** also report to the upper layer when an inactive destination address is marked as active. When the peer endpoint is multi-homed and the last chunk sent to it was a retransmission to an alternate address, there exists an ambiguity as to whether or not the acknowledgement could be credited to the address of the last chunk sent. However, this ambiguity does not seem to have significant consequences for SCTP behavior. If this ambiguity is undesirable, the transmitter **MAY** choose not to clear the error counter if the last chunk sent was a retransmission.

Note: When configuring the SCTP endpoint, the user ought to avoid having the value of 'Association.Max.Retrans' larger than the summation of the 'Path.Max.Retrans' of all the destination addresses for the remote endpoint. Otherwise, all the destination addresses might become inactive while the endpoint still considers the peer endpoint reachable. When this condition occurs, how SCTP chooses to function is implementation specific.

When the primary path is marked inactive (due to excessive retransmissions, for instance), the sender **MAY** automatically transmit new packets to an alternate destination address if one exists and is active. If more than one alternate address is active when the primary path is marked inactive, only ONE transport address **SHOULD** be chosen and used as the new destination transport address.

8.3. Path Heartbeat

By default, an SCTP endpoint **SHOULD** monitor the reachability of the idle destination transport address(es) of its peer by sending a HEARTBEAT chunk periodically to the destination transport address(es). The sending of HEARTBEAT chunks **MAY** begin upon reaching the ESTABLISHED state and is discontinued after sending either a SHUTDOWN chunk or SHUTDOWN ACK chunk. A receiver of a HEARTBEAT chunk **MUST** respond to a HEARTBEAT chunk with a HEARTBEAT ACK chunk after entering the COOKIE-ECHOED state (sender of the INIT chunk) or the ESTABLISHED state (receiver of the INIT chunk), up until reaching the SHUTDOWN-SENT state (sender of the SHUTDOWN chunk) or the SHUTDOWN-ACK-SENT state (receiver of the SHUTDOWN chunk).

A destination transport address is considered "idle" if no new chunk that can be used for updating path RTT (usually including first transmission DATA, INIT, COOKIE ECHO, or HEARTBEAT chunks, etc.) and no HEARTBEAT chunk has been sent to it within the current heartbeat period of that address. This applies to both active and inactive destination addresses.

The upper layer can optionally initiate the following functions:

A) Disable heartbeat on a specific destination transport address of a given association,

B) Change the 'HB.interval',

C) Re-enable heartbeat on a specific destination transport address of a given association, and

D) Request the sending of an on-demand HEARTBEAT chunk on a specific destination transport address of a given association.

The endpoint **SHOULD** increment the respective error counter of the destination transport address each time a HEARTBEAT chunk is sent to that address and not acknowledged within one RTO.

When the value of this counter exceeds the protocol parameter 'Path.Max.Retrans', the endpoint **SHOULD** mark the corresponding destination address as inactive if it is not so marked and **SHOULD** also report to the upper layer the change in reachability of this destination address. After this, the endpoint **SHOULD** continue sending HEARTBEAT chunks on this destination address but **SHOULD** stop increasing the counter.

The sender of the HEARTBEAT chunk **SHOULD** include in the Heartbeat Information field of the chunk the current time when the packet is sent and the destination address to which the packet is sent.

Implementation Note: An alternative implementation of the heartbeat mechanism that can be used is to increment the error counter variable every time a HEARTBEAT chunk is sent to a destination. Whenever a HEARTBEAT ACK chunk arrives, the sender **SHOULD** clear the error counter of the destination that the HEARTBEAT chunk was sent to. This, in effect, would clear the previously stroked error (and any other error counts as well).

The receiver of the HEARTBEAT chunk **SHOULD** immediately respond with a HEARTBEAT ACK chunk that contains the Heartbeat Information TLV, together with any other received TLVs, copied unchanged from the received HEARTBEAT chunk.

Upon the receipt of the HEARTBEAT ACK chunk, the sender of the HEARTBEAT chunk **SHOULD** clear the error counter of the destination transport address to which the HEARTBEAT chunk was sent and mark the destination transport address as active if it is not so marked. The endpoint **SHOULD** report to the upper layer when an inactive destination address is marked as active due to the reception of the latest HEARTBEAT ACK chunk. The receiver of the HEARTBEAT ACK chunk **SHOULD** also clear the association overall error count (as defined in Section 8.1).

The receiver of the HEARTBEAT ACK chunk **SHOULD** also perform an RTT measurement for that destination transport address using the time value carried in the HEARTBEAT ACK chunk.

On an idle destination address that is allowed to heartbeat, it is **RECOMMENDED** that a HEARTBEAT chunk is sent once per RTO of that destination address plus the protocol parameter 'HB.interval', with jittering of +/- 50% of the RTO value and exponential backoff of the RTO if the previous HEARTBEAT chunk is unanswered.

A primitive is provided for the SCTP user to change the 'HB.interval' and turn on or off the heartbeat on a given destination address. The 'HB.interval' set by the SCTP user is added to the RTO of that destination (including any exponential backoff). Only one heartbeat **SHOULD** be sent each time the heartbeat timer expires (if multiple destinations are idle). It is an implementation decision on how to choose which of the candidate idle destinations to heartbeat to (if more than one destination is idle).

When tuning the 'HB.interval', there is a side effect that **SHOULD** be taken into account. When this value is increased, i.e., the time between the sending of HEARTBEAT chunks is longer, the detection of lost ABORT chunks takes longer as well. If a peer endpoint sends an ABORT chunk for any reason and the ABORT chunk is lost, the local endpoint will only discover the lost ABORT chunk by sending a DATA chunk or HEARTBEAT chunk (thus causing the peer to send another ABORT chunk). This is to be considered when tuning the heartbeat timer. If the sending of HEARTBEAT chunks is disabled, only sending DATA chunks to the association will discover a lost ABORT chunk from the peer.

8.4. Handle "Out of the Blue" Packets

An SCTP packet is called an "Out of the Blue" (OOTB) packet if it is correctly formed (i.e., passed the receiver's CRC32c check; see Section 6.8), but the receiver is not able to identify the association to which this packet belongs.

The receiver of an OOTB packet does the following:

1) If the OOTB packet is to or from a non-unicast address, a receiver **SHOULD** silently discard the packet. Otherwise,

2) If the OOTB packet contains an ABORT chunk, the receiver **MUST** silently discard the OOTB packet and take no further action. Otherwise,

3) If the packet contains an INIT chunk with a Verification Tag set to 0, it **SHOULD** be processed as described in Section 5.1. If, for whatever reason, the INIT chunk cannot be processed normally and an ABORT chunk has to be sent in response, the Verification Tag of the packet containing the ABORT chunk **MUST** be the Initiate Tag of the received INIT chunk, and the T bit of the ABORT chunk has to be set to 0, indicating that the Verification Tag is not reflected. Otherwise,

4) If the packet contains a COOKIE ECHO chunk as the first chunk, it **MUST** be processed as described in Section 5.1. Otherwise,

5) If the packet contains a SHUTDOWN ACK chunk, the receiver **SHOULD** respond to the sender of the OOTB packet with a SHUTDOWN COMPLETE chunk. When sending the SHUTDOWN COMPLETE chunk, the receiver of the OOTB packet **MUST** fill in the Verification Tag field of the outbound packet with the Verification Tag received in the SHUTDOWN ACK chunk and set the T bit in the Chunk Flags to indicate that the Verification Tag is reflected. Otherwise,

6) If the packet contains a SHUTDOWN COMPLETE chunk, the receiver **SHOULD** silently discard the packet and take no further action. Otherwise,

7) If the packet contains an ERROR chunk with the "Stale Cookie" error cause or a COOKIE ACK chunk, the SCTP packet **SHOULD** be silently discarded. Otherwise,

8) The receiver **SHOULD** respond to the sender of the OOTB packet with an ABORT chunk. When sending the ABORT chunk, the receiver of the OOTB packet **MUST** fill in the Verification Tag field of the outbound packet with the value found in the Verification Tag field of the OOTB packet and set the T bit in the Chunk Flags to indicate that the Verification Tag is reflected. After sending this ABORT chunk, the receiver of the OOTB packet **MUST** discard the OOTB packet and **MUST NOT** take any further action.

8.5. Verification Tag

The Verification Tag rules defined in this section apply when sending or receiving SCTP packets that do not contain an INIT, SHUTDOWN COMPLETE, COOKIE ECHO (see Section 5.1), ABORT, or SHUTDOWN ACK chunk. The rules for sending and receiving SCTP packets containing one of these chunk types are discussed separately in Section 8.5.1.

When sending an SCTP packet, the endpoint **MUST** fill in the Verification Tag field of the outbound packet with the tag value in the Initiate Tag parameter of the INIT or INIT ACK chunk received from its peer.

When receiving an SCTP packet, the endpoint **MUST** ensure that the value in the Verification Tag field of the received SCTP packet matches its own tag. If the received Verification Tag value does not match the receiver's own tag value, the receiver **MUST** silently discard the packet and **MUST NOT** process it any further, except for those cases listed in Section 8.5.1 below.

8.5.1. Exceptions in Verification Tag Rules

A) Rules for packets carrying an INIT chunk:

- The sender **MUST** set the Verification Tag of the packet to 0.
- When an endpoint receives an SCTP packet with the Verification Tag set to 0, it **SHOULD** verify that the packet contains only an INIT chunk. Otherwise, the receiver **MUST** silently discard the packet.

B) Rules for packets carrying an ABORT chunk:

- The endpoint **MUST** always fill in the Verification Tag field of the outbound packet with the destination endpoint's tag value if it is known.
- If the ABORT chunk is sent in response to an OOTB packet, the endpoint **MUST** follow the procedure described in Section 8.4.
- The receiver of an ABORT chunk **MUST** accept the packet if the Verification Tag field of the packet matches its own tag and the T bit is not set OR if it is set to its Peer's Tag and the T bit is set in the Chunk Flags. Otherwise, the receiver **MUST** silently discard the packet and take no further action.

C) Rules for packets carrying a SHUTDOWN COMPLETE chunk:

- When sending a SHUTDOWN COMPLETE chunk, if the receiver of the SHUTDOWN ACK chunk has a TCB, then the destination endpoint's tag **MUST** be used and the T bit **MUST NOT** be set. Only where no TCB exists **SHOULD** the sender use the Verification Tag from the SHUTDOWN ACK chunk and **MUST** set the T bit.
- The receiver of a SHUTDOWN COMPLETE chunk accepts the packet if the Verification Tag field of the packet matches its own tag and the T bit is not set OR if it is set to its Peer's Tag and the T bit is set in the Chunk Flags. Otherwise, the receiver **MUST** silently discard the packet and take no further action. An endpoint **MUST** ignore the SHUTDOWN COMPLETE chunk if it is not in the SHUTDOWN-ACK-SENT state.

D) Rules for packets carrying a COOKIE ECHO chunk:

- When sending a COOKIE ECHO chunk, the endpoint **MUST** use the value of the Initiate Tag received in the INIT ACK chunk.
- The receiver of a COOKIE ECHO chunk follows the procedures in Section 5.

E) Rules for packets carrying a SHUTDOWN ACK chunk:

- If the receiver is in COOKIE-ECHOED or COOKIE-WAIT state, the procedures in Section 8.4 **SHOULD** be followed; in other words, it is treated as an OOTB packet.

9. Termination of Association

An endpoint **SHOULD** terminate its association when it exits from service. An association can be terminated by either abort or shutdown. An abort of an association is abortive by definition in that any data pending on either end of the association is discarded and not delivered to the peer. A shutdown of an association is considered a graceful close where all data in queue by either endpoint is delivered to the respective peers. However, in the case of a shutdown, SCTP does not support a half-open state (like TCP), wherein one side might continue sending data while the other end is closed. When either endpoint performs a shutdown, the association on each peer will stop accepting new data from its user and only deliver data in queue at the time of sending or receiving the SHUTDOWN chunk.

9.1. Abort of an Association

When an endpoint decides to abort an existing association, it **MUST** send an ABORT chunk to its peer endpoint. The sender **MUST** fill in the peer's Verification Tag in the outbound packet and **MUST NOT** bundle any DATA chunk with the ABORT chunk. If the association is aborted on request of the upper layer, a "User-Initiated Abort" error cause (see Section 3.3.10.12) **SHOULD** be present in the ABORT chunk.

An endpoint **MUST NOT** respond to any received packet that contains an ABORT chunk (also see Section 8.4).

An endpoint receiving an ABORT chunk **MUST** apply the special Verification Tag check rules described in Section 8.5.1.

After checking the Verification Tag, the receiving endpoint **MUST** remove the association from its record and **SHOULD** report the termination to its upper layer. If a "User-Initiated Abort" error cause is present in the ABORT chunk, the Upper Layer Abort Reason **SHOULD** be made available to the upper layer.

9.2. Shutdown of an Association

Using the SHUTDOWN primitive (see Section 11.1), the upper layer of an endpoint in an association can gracefully close the association. This will allow all outstanding DATA chunks from the peer of the shutdown initiator to be delivered before the association terminates.

Upon receipt of the SHUTDOWN primitive from its upper layer, the endpoint enters the SHUTDOWN-PENDING state and remains there until all outstanding data has been acknowledged by its peer. The endpoint accepts no new data from its upper layer but retransmits data to the peer endpoint if necessary to fill gaps.

Once all its outstanding data has been acknowledged, the endpoint sends a SHUTDOWN chunk to its peer, including in the Cumulative TSN Ack field the last sequential TSN it has received from the peer. It **SHOULD** then start the T2-shutdown timer and enter the SHUTDOWN-SENT state. If the timer expires, the endpoint **MUST** resend the SHUTDOWN chunk with the updated last sequential TSN received from its peer.

The rules in Section 6.3 **MUST** be followed to determine the proper timer value for T2-shutdown. To indicate any gaps in TSN, the endpoint **MAY** also bundle a SACK chunk with the SHUTDOWN chunk in the same SCTP packet.

An endpoint **SHOULD** limit the number of retransmissions of the SHUTDOWN chunk to the protocol parameter 'Association.Max.Retrans'. If this threshold is exceeded, the endpoint **SHOULD** destroy the TCB and **SHOULD** report the peer endpoint unreachable to the upper layer (and thus the association enters the CLOSED state). The reception of any packet from its peer (i.e., as the peer sends all of its queued DATA chunks) **SHOULD** clear the endpoint's retransmission count and restart the T2-shutdown timer, giving its peer ample opportunity to transmit all of its queued DATA chunks that have not yet been sent.

Upon reception of the SHUTDOWN chunk, the peer endpoint does the following:

- enter the SHUTDOWN-RECEIVED state,
- stop accepting new data from its SCTP user, and
- verify, by checking the Cumulative TSN Ack field of the chunk, that all its outstanding DATA chunks have been received by the SHUTDOWN chunk sender.

Once an endpoint has reached the SHUTDOWN-RECEIVED state, it **MUST** ignore ULP shutdown requests but **MUST** continue responding to SHUTDOWN chunks from its peer.

If there are still outstanding DATA chunks left, the SHUTDOWN chunk receiver **MUST** continue to follow normal data transmission procedures defined in Section 6, until all outstanding DATA chunks are acknowledged; however, the SHUTDOWN chunk receiver **MUST NOT** accept new data from its SCTP user.

While in the SHUTDOWN-SENT state, the SHUTDOWN chunk sender **MUST** immediately respond to each received packet containing one or more DATA chunks with a SHUTDOWN chunk and restart the T2-shutdown timer. If a SHUTDOWN chunk by itself cannot acknowledge all of the received DATA chunks (i.e., there are TSNs that can be acknowledged that are larger than the cumulative TSN and thus gaps exist in the TSN sequence) or if duplicate TSNs have been received, then a SACK chunk **MUST** also be sent.

The sender of the SHUTDOWN chunk **MAY** also start an overall guard timer T5-shutdown-guard to bound the overall time for the shutdown sequence. At the expiration of this timer, the sender **SHOULD** abort the association by sending an ABORT chunk. If the T5-shutdown-guard timer is used, it **SHOULD** be set to the **RECOMMENDED** value of 5 times 'RTO.Max'.

If the receiver of the SHUTDOWN chunk has no more outstanding DATA chunks, the SHUTDOWN chunk receiver **MUST** send a SHUTDOWN ACK chunk and start a T2-shutdown timer of its own, entering the SHUTDOWN-ACK-SENT state. If the timer expires, the endpoint **MUST** resend the SHUTDOWN ACK chunk.

The sender of the SHUTDOWN ACK chunk **SHOULD** limit the number of retransmissions of the SHUTDOWN ACK chunk to the protocol parameter 'Association.Max.Retrans'. If this threshold is exceeded, the endpoint **SHOULD** destroy the TCB and **SHOULD** report the peer endpoint unreachable to the upper layer (and thus the association enters the CLOSED state).

Upon the receipt of the SHUTDOWN ACK chunk, the sender of the SHUTDOWN chunk **MUST** stop the T2-shutdown timer, send a SHUTDOWN COMPLETE chunk to its peer, and remove all record of the association.

Upon reception of the SHUTDOWN COMPLETE chunk, the endpoint verifies that it is in the SHUTDOWN-ACK-SENT state; if it is not, the chunk **SHOULD** be discarded. If the endpoint is in the SHUTDOWN-ACK-SENT state, the endpoint **SHOULD** stop the T2-shutdown timer and remove all knowledge of the association (and thus the association enters the CLOSED state).

An endpoint **SHOULD** ensure that all its outstanding DATA chunks have been acknowledged before initiating the shutdown procedure.

An endpoint **SHOULD** reject any new data request from its upper layer if it is in the SHUTDOWN-PENDING, SHUTDOWN-SENT, SHUTDOWN-RECEIVED, or SHUTDOWN-ACK-SENT state.

If an endpoint is in the SHUTDOWN-ACK-SENT state and receives an INIT chunk (e.g., if the SHUTDOWN COMPLETE chunk was lost) with source and destination transport addresses (either in the IP addresses or in the INIT chunk) that belong to this association, it **SHOULD** discard the INIT chunk and retransmit the SHUTDOWN ACK chunk.

Note: Receipt of a packet containing an INIT chunk with the same source and destination IP addresses as used in transport addresses assigned to an endpoint but with a different port number indicates the initialization of a separate association.

The sender of the INIT or COOKIE ECHO chunk **SHOULD** respond to the receipt of a SHUTDOWN ACK chunk with a stand-alone SHUTDOWN COMPLETE chunk in an SCTP packet with the Verification Tag field of its common header set to the same tag that was received in the packet containing the SHUTDOWN ACK chunk. This is considered an OOTB packet as defined in Section 8.4. The sender of the INIT chunk lets T1-init continue running and remains in the COOKIE-WAIT or COOKIE-ECHOED state. Normal T1-init timer expiration will cause the INIT or COOKIE chunk to be retransmitted and thus start a new association.

If a SHUTDOWN chunk is received in the COOKIE-WAIT or COOKIE ECHOED state, the SHUTDOWN chunk **SHOULD** be silently discarded.

If an endpoint is in the SHUTDOWN-SENT state and receives a SHUTDOWN chunk from its peer, the endpoint **SHOULD** respond immediately with a SHUTDOWN ACK chunk to its peer and move into the SHUTDOWN-ACK-SENT state, restarting its T2-shutdown timer.

If an endpoint is in the SHUTDOWN-ACK-SENT state and receives a SHUTDOWN ACK, it **MUST** stop the T2-shutdown timer, send a SHUTDOWN COMPLETE chunk to its peer, and remove all record of the association.

10. ICMP Handling

Whenever an ICMP message is received by an SCTP endpoint, the following procedures **MUST** be followed to ensure proper utilization of the information being provided by layer 3.

ICMP1) An implementation **MAY** ignore all ICMPv4 messages where the type field is not set to "Destination Unreachable".

ICMP2) An implementation **MAY** ignore all ICMPv6 messages where the type field is not "Destination Unreachable", "Parameter Problem", or "Packet Too Big".

ICMP3) An implementation **SHOULD** ignore any ICMP messages where the code indicates "Port Unreachable".

ICMP4) An implementation **MAY** ignore all ICMPv6 messages of type "Parameter Problem" if the code is not "Unrecognized Next Header Type Encountered".

ICMP5) An implementation **MUST** use the payload of the ICMP message (v4 or v6) to locate the association that sent the message to which ICMP is responding. If the association cannot be found, an implementation **SHOULD** ignore the ICMP message.

ICMP6) An implementation **MUST** validate that the Verification Tag contained in the ICMP message matches the Verification Tag of the peer. If the Verification Tag is not 0 and does not match, discard the ICMP message. If it is 0 and the ICMP message contains enough bytes to verify that the chunk type is an INIT chunk and that the Initiate Tag matches the tag of the peer, continue with ICMP7. If the ICMP message is too short or the chunk type or the Initiate Tag does not match, silently discard the packet.

ICMP7) If the ICMP message is either an ICMPv6 message of type "Packet Too Big" or an ICMPv4 message of type "Destination Unreachable" and code "Fragmentation Needed", an implementation **SHOULD** process this information as defined for PMTU discovery.

ICMP8) If the ICMP code is "Unrecognized Next Header Type Encountered" or "Protocol Unreachable", an implementation **MUST** treat this message as an abort with the T bit set if it does not contain an INIT chunk. If it does contain an INIT chunk and the association is in the COOKIE-WAIT state, handle the ICMP message like an ABORT chunk.

ICMP9) If the ICMP type is "Destination Unreachable", the implementation **MAY** move the destination to the unreachable state or, alternatively, increment the path error counter. SCTP **MAY** provide information to the upper layer indicating the reception of ICMP messages when reporting a network status change.

These procedures differ from [RFC1122] and from its requirements for processing of port-unreachable messages and the requirements that an implementation **MUST** abort associations in response to a protocol unreachable message. Port-unreachable messages are not processed, since an implementation will send an ABORT chunk, not a port-unreachable message. The stricter handling of the protocol unreachable message is due to security concerns for hosts that do not support SCTP.

11. Interface with Upper Layer

The Upper Layer Protocols (ULPs) request services by passing primitives to SCTP and receive notifications from SCTP for various events.

The primitives and notifications described in this section can be used as a guideline for implementing SCTP. The following functional description of ULP interface primitives is shown for illustrative purposes. Different SCTP implementations can have different ULP interfaces. However, all SCTP implementations are expected to provide a certain minimum set of services to guarantee that all SCTP implementations can support the same protocol hierarchy.

Please note that this section is informational only.

[RFC6458] and Section 7 ("Socket API Considerations") of [RFC7053] define an extension of the socket API for SCTP as described in this document.

11.1. ULP-to-SCTP

The following sections functionally characterize a ULP/SCTP interface. The notation used is similar to most procedure or function calls in high-level languages.

The ULP primitives described below specify the basic functions that SCTP performs to support inter-process communication. Individual implementations define their own exact format and provide combinations or subsets of the basic functions in single calls.

11.1.1. Initialize

```
INITIALIZE ([local port],[local eligible address list])
-> local SCTP instance name
```

This primitive allows SCTP to initialize its internal data structures and allocate necessary resources for setting up its operation environment. Once SCTP is initialized, ULP can communicate directly with other endpoints without re-invoking this primitive.

SCTP will return a local SCTP instance name to the ULP.

Mandatory attributes:
None.

Optional attributes:
local port: SCTP port number, if ULP wants it to be specified.

local eligible address list: an address list that the local SCTP endpoint binds. By default, if an address list is not included, all IP addresses assigned to the host are used by the local endpoint.

Implementation Note: If this optional attribute is supported by an implementation, it will be the responsibility of the implementation to enforce that the IP source address field of any SCTP packets sent by this endpoint contains one of the IP addresses indicated in the local eligible address list.

11.1.2. Associate

```
ASSOCIATE(local SCTP instance name,
initial destination transport addr list, outbound stream count)
-> association id [,destination transport addr list]
[,outbound stream count]
```

This primitive allows the upper layer to initiate an association to a specific peer endpoint.

The peer endpoint is specified by one or more of the transport addresses that defines the endpoint (see Section 1.3). If the local SCTP instance has not been initialized, the ASSOCIATE is considered an error.

An association id, which is a local handle to the SCTP association, will be returned on successful establishment of the association. If SCTP is not able to open an SCTP association with the peer endpoint, an error is returned.

Other association parameters can be returned, including the complete destination transport addresses of the peer as well as the outbound stream count of the local endpoint. One of the transport addresses from the returned destination addresses will be selected by the local

endpoint as the default primary path for sending SCTP packets to this peer. The returned "destination transport addr list" can be used by the ULP to change the default primary path or to force sending a packet to a specific transport address.

Implementation Note: If the ASSOCIATE primitive is implemented as a blocking function call, the ASSOCIATE primitive can return association parameters in addition to the association id upon successful establishment. If ASSOCIATE primitive is implemented as a non-blocking call, only the association id is returned and association parameters are passed using the COMMUNICATION UP notification.

Mandatory attributes:

local SCTP instance name: obtained from the INITIALIZE operation.

initial destination transport addr list: a non-empty list of transport addresses of the peer endpoint with which the association is to be established.

outbound stream count: the number of outbound streams the ULP would like to open towards this peer endpoint.

Optional attributes:

None.

11.1.3. Shutdown

```
SHUTDOWN(association id) -> result
```

Gracefully closes an association. Any locally queued user data will be delivered to the peer. The association will be terminated only after the peer acknowledges all the SCTP packets sent. A success code will be returned on successful termination of the association. If attempting to terminate the association results in a failure, an error code is returned.

Mandatory attributes:

association id: local handle to the SCTP association.

Optional attributes:

None.

11.1.4. Abort

```
ABORT(association id [, Upper Layer Abort Reason]) -> result
```

Ungracefully closes an association. Any locally queued user data will be discarded, and an ABORT chunk is sent to the peer. A success code will be returned on successful abort of the association. If attempting to abort the association results in a failure, an error code is returned.

Mandatory attributes:

association id: local handle to the SCTP association.

Optional attributes:

Upper Layer Abort Reason: reason of the abort to be passed to the peer.

11.1.5. Send

```
SEND(association id, buffer address, byte count [,context]
[,stream id] [,life time] [,destination transport address]
[,unordered flag] [,no-bundle flag] [,payload protocol-id]
[,sack-immediately flag]) -> result
```

This is the main method to send user data via SCTP.

Mandatory attributes:

association id: local handle to the SCTP association.

buffer address: the location where the user message to be transmitted is stored.

byte count: the size of the user data in number of bytes.

Optional attributes:

context: optional information provided that will be carried in the SEND FAILURE notification to the ULP if the transportation of this user message fails.

stream id: indicates which stream to send the data on. If not specified, stream 0 will be used.

life time: specifies the life time of the user data. The user data will not be sent by SCTP after the life time expires. This parameter can be used to avoid efforts to transmit stale user messages. SCTP notifies the ULP if the data cannot be initiated to transport (i.e., sent to the destination via SCTP's SEND primitive) within the life time variable. However, the user data will be transmitted if SCTP has attempted to transmit a chunk before the life time expired.

Implementation Note: In order to better support the data life time option, the transmitter can hold back the assigning of the TSN number to an outbound DATA chunk to the last moment. And, for implementation simplicity, once a TSN number has been assigned, the sender considers the send of this DATA chunk as committed, overriding any life time option attached to the DATA chunk.

destination transport address: specified as one of the destination transport addresses of the peer endpoint to which this packet is sent. Whenever possible, SCTP uses this destination transport address for sending the packets, instead of the current primary path.

unordered flag: this flag, if present, indicates that the user would like the data delivered in an unordered fashion to the peer (i.e., the U flag is set to 1 on all DATA chunks carrying this message).

no-bundle flag: instructs SCTP not to delay the sending of DATA chunks for this user data just to allow it to be bundled with other outbound DATA chunks. When faced with network congestion, SCTP might still bundle the data, even when this flag is present.

payload protocol-id: a 32-bit unsigned integer that is to be passed to the peer, indicating the type of payload protocol data being transmitted. Note that the upper layer is responsible for the host to network byte order conversion of this field, which is passed by SCTP as 4 bytes of opaque data.

sack-immediately flag: set the I bit on the last DATA chunk used for the user message to be transmitted.

11.1.6. Set Primary

```
SETPRIMARY(association id, destination transport address,
[source transport address]) -> result
```

Instructs the local SCTP to use the specified destination transport address as the primary path for sending packets.

The result of attempting this operation is returned. If the specified destination transport address is not present in the "destination transport address list" returned earlier in an ASSOCIATE primitive or COMMUNICATION UP notification, an error is returned.

Mandatory attributes:

association id: local handle to the SCTP association.

destination transport address: specified as one of the transport addresses of the peer endpoint, which is used as the primary address for sending packets. This overrides the current primary address information maintained by the local SCTP endpoint.

Optional attributes:

source transport address: optionally, some implementations can allow you to set the default source address placed in all outgoing IP datagrams.

11.1.7. Receive

```
RECEIVE(association id, buffer address, buffer size [,stream id])
-> byte count [,transport address] [,stream id]
[,stream sequence number] [,partial flag] [,payload protocol-id]
```

This primitive reads the first user message in the SCTP in-queue into the buffer specified by ULP, if there is one available. The size of the message read, in bytes, will be returned. It might, depending on the specific implementation, also return other information, such as the sender's address, the stream id on which it is received, whether there are more messages available for retrieval, etc. For ordered messages, their Stream Sequence Number might also be returned.

Depending upon the implementation, if this primitive is invoked when no message is available, the implementation returns an indication of this condition or blocks the invoking process until data does become available.

Mandatory attributes:

association id: local handle to the SCTP association.

buffer address: the memory location indicated by the ULP to store the received message.

buffer size: the maximum size of data to be received, in bytes.

Optional attributes:

stream id: to indicate which stream to receive the data on.

stream sequence number: the Stream Sequence Number assigned by the sending SCTP peer.

partial flag: if this returned flag is set to 1, then this primitive contains a partial delivery of the whole message. When this flag is set, the stream id and stream sequence number accompanies this primitive. When this flag is set to 0, it indicates that no more deliveries will be received for this stream sequence number.

payload protocol-id: a 32-bit unsigned integer that is received from the peer indicating the type of payload protocol of the received data. Note that the upper layer is responsible for the host to network byte order conversion of this field, which is passed by SCTP as 4 bytes of opaque data.

11.1.8. Status

```
STATUS(association id) -> status data
```

This primitive returns a data block containing the following information:

- association connection state,
- destination transport address list,
- destination transport address reachability states,
- current receiver window size,
- current congestion window sizes,
- number of unacknowledged DATA chunks,
- number of DATA chunks pending receipt,
- primary path,
- most recent SRTT on primary path,
- RTO on primary path,
- SRTT and RTO on other destination addresses, etc.

Mandatory attributes:

association id: local handle to the SCTP association.

Optional attributes:
None.

11.1.9. Change Heartbeat

```
CHANGE HEARTBEAT(association id, destination transport address,
new state [,interval]) -> result
```

Instructs the local endpoint to enable or disable heartbeat on the specified destination transport address.

The result of attempting this operation is returned.

Note: Even when enabled, heartbeat will not take place if the destination transport address is not idle.

Mandatory attributes:
association id: local handle to the SCTP association.

destination transport address: specified as one of the transport addresses of the peer endpoint.

new state: the new state of heartbeat for this destination transport address (either enabled or disabled).

Optional attributes:
interval: if present, indicates the frequency of the heartbeat if this is to enable heartbeat on a destination transport address. This value is added to the RTO of the destination transport address. This value, if present, affects all destinations.

11.1.10. Request Heartbeat

```
REQUESTHEARTBEAT(association id, destination transport address)
-> result
```

Instructs the local endpoint to perform a heartbeat on the specified destination transport address of the given association. The returned result indicates whether the transmission of the HEARTBEAT chunk to the destination address is successful.

Mandatory attributes:
association id: local handle to the SCTP association.

destination transport address: the transport address of the association on which a heartbeat is issued.

Optional attributes:
None.

11.1.11. Get SRTT Report

```
GETSRTTREPORT(association id, destination transport address)
-> srtt result
```

Instructs the local SCTP to report the current SRTT measurement on the specified destination transport address of the given association. The returned result can be an integer containing the most recent SRTT in milliseconds.

Mandatory attributes:
association id: local handle to the SCTP association.

destination transport address: the transport address of the association on which the SRTT measurement is to be reported.

Optional attributes:
None.

11.1.12. Set Failure Threshold

```
SETFAILURETHRESHOLD(association id, destination transport address,
failure threshold) -> result
```

This primitive allows the local SCTP to customize the reachability failure detection threshold 'Path.Max.Retrans' for the specified destination address. Note that this can also be done using the SETPROTOCOLPARAMETERS primitive (Section 11.1.13).

Mandatory attributes:
association id: local handle to the SCTP association.

destination transport address: the transport address of the association on which the failure detection threshold is to be set.

failure threshold: the new value of 'Path.Max.Retrans' for the destination address.

Optional attributes:
None.

11.1.13. Set Protocol Parameters

```
SETPROTOCOLPARAMETERS(association id,
[destination transport address,] protocol parameter list)
-> result
```

This primitive allows the local SCTP to customize the protocol parameters.

Mandatory attributes:

association id: local handle to the SCTP association.

protocol parameter list: the specific names and values of the protocol parameters (e.g., 'Association.Max.Retrans' (see Section 16) or other parameters like the DSCP) that the SCTP user wishes to customize.

Optional attributes:

destination transport address: some of the protocol parameters might be set on a per-destination-transport-address basis.

11.1.14. Receive Unsent Message

```
RECEIVE_UNSENT(data retrieval id, buffer address, buffer size
[,stream id] [, stream sequence number] [,partial flag]
[,payload protocol-id])
```

This primitive reads a user message that has never been sent into the buffer specified by ULP.

Mandatory attributes:

data retrieval id: the identification passed to the ULP in the SEND FAILURE notification.

buffer address: the memory location indicated by the ULP to store the received message.

buffer size: the maximum size of data to be received, in bytes.

Optional attributes:

stream id: this is a return value that is set to indicate which stream the data was sent to.

stream sequence number: this value is returned, indicating the Stream Sequence Number that was associated with the message.

partial flag: if this returned flag is set to 1, then this message is a partial delivery of the whole message. When this flag is set, the stream id and stream sequence number accompanies this primitive. When this flag is set to 0, it indicates that no more deliveries will be received for this stream sequence number.

payload protocol-id: The 32-bit unsigned integer that was set to be sent to the peer, indicating the type of payload protocol of the received data.

11.1.15. Receive Unacknowledged Message

```
RECEIVE_UNACKED(data retrieval id, buffer address, buffer size,
[,stream id] [,stream sequence number] [,partial flag]
[,payload protocol-id])
```

This primitive reads a user message that has been sent and has not been acknowledged by the peer into the buffer specified by ULP.

Mandatory attributes:

data retrieval id: the identification passed to the ULP in the SEND FAILURE notification.

buffer address: the memory location indicated by the ULP to store the received message.

buffer size: the maximum size of data to be received, in bytes.

Optional attributes:

stream id: this is a return value that is set to indicate which stream the data was sent to.

stream sequence number: this value is returned, indicating the Stream Sequence Number that was associated with the message.

partial flag: if this returned flag is set to 1, then this message is a partial delivery of the whole message. When this flag is set, the stream id and stream sequence number accompanies this primitive. When this flag is set to 0, it indicates that no more deliveries will be received for this stream sequence number.

payload protocol-id: the 32-bit unsigned integer that was sent to the peer indicating the type of payload protocol of the received data.

11.1.16. Destroy SCTP Instance

```
DESTROY(local SCTP instance name)
```

Mandatory attributes:

local SCTP instance name: this is the value that was passed to the application in the initialize primitive and it indicates which SCTP instance is to be destroyed.

Optional attributes:

None.

11.2. SCTP-to-ULP

It is assumed that the operating system or application environment provides a means for the SCTP to asynchronously signal the ULP process. When SCTP does signal a ULP process, certain information is passed to the ULP.

Implementation Note: In some cases, this might be done through a separate socket or error channel.

11.2.1. DATA ARRIVE Notification

SCTP invokes this notification on the ULP when a user message is successfully received and ready for retrieval.

The following might optionally be passed with the notification:

association id: local handle to the SCTP association.

stream id: to indicate which stream the data is received on.

11.2.2. SEND FAILURE Notification

If a message cannot be delivered, SCTP invokes this notification on the ULP.

The following might optionally be passed with the notification:

association id: local handle to the SCTP association.

data retrieval id: an identification used to retrieve unsent and unacknowledged data.

mode: indicates whether no part of the message never has been sent or if at least part of it has been sent but it is not completely acknowledged.

cause code: indicating the reason of the failure, e.g., size too large, message life time expiration, etc.

context: optional information associated with this message (see Section 11.1.5).

11.2.3. NETWORK STATUS CHANGE Notification

When a destination transport address is marked inactive (e.g., when SCTP detects a failure) or marked active (e.g., when SCTP detects a recovery), SCTP invokes this notification on the ULP.

The following is passed with the notification:

association id: local handle to the SCTP association.

destination transport address: this indicates the destination transport address of the peer endpoint affected by the change.

new-status: this indicates the new status.

11.2.4. COMMUNICATION UP Notification

This notification is used when SCTP becomes ready to send or receive user messages or when a lost communication to an endpoint is restored.

Implementation Note: If the ASSOCIATE primitive is implemented as a blocking function call, the association parameters are returned as a result of the ASSOCIATE primitive itself. In that case, the COMMUNICATION UP notification is optional at the association initiator's side.

The following is passed with the notification:

association id: local handle to the SCTP association.

status: this indicates what type of event has occurred.

destination transport address list: the complete set of transport addresses of the peer.

outbound stream count: the maximum number of streams allowed to be used in this association by the ULP.

inbound stream count: the number of streams the peer endpoint has requested with this association (this might not be the same number as 'outbound stream count').

11.2.5. COMMUNICATION LOST Notification

When SCTP loses communication to an endpoint completely (e.g., via Heartbeats) or detects that the endpoint has performed an abort operation, it invokes this notification on the ULP.

The following is passed with the notification:

association id: local handle to the SCTP association.

status: this indicates what type of event has occurred; the status might indicate that a failure OR a normal termination event occurred in response to a shutdown or abort request.

The following might be passed with the notification:

last-acked: the TSN last acked by that peer endpoint.

last-sent: the TSN last sent to that peer endpoint.

Upper Layer Abort Reason: the abort reason specified in case of a user-initiated abort.

11.2.6. COMMUNICATION ERROR Notification

When SCTP receives an ERROR chunk from its peer and decides to notify its ULP, it can invoke this notification on the ULP.

The following can be passed with the notification:

association id: local handle to the SCTP association.

error info: this indicates the type of error and optionally some additional information received through the ERROR chunk.

11.2.7. RESTART Notification

When SCTP detects that the peer has restarted, it might send this notification to its ULP.

The following can be passed with the notification:

association id: local handle to the SCTP association.

11.2.8. SHUTDOWN COMPLETE Notification

When SCTP completes the shutdown procedures (Section 9.2), this notification is passed to the upper layer.

The following can be passed with the notification:

association id: local handle to the SCTP association.

12. Security Considerations

12.1. Security Objectives

As a common transport protocol designed to reliably carry time-sensitive user messages, such as billing or signaling messages for telephony services, between two networked endpoints, SCTP has the following security objectives:

- availability of reliable and timely data transport services
- integrity of the user-to-user information carried by SCTP

12.2. SCTP Responses to Potential Threats

SCTP could potentially be used in a wide variety of risk situations. It is important for operators of systems running SCTP to analyze their particular situations and decide on the appropriate counter-measures.

Operators of systems running SCTP might consult [RFC2196] for guidance in securing their site.

12.2.1. Countering Insider Attacks

The principles of [RFC2196] might be applied to minimize the risk of theft of information or sabotage by insiders. Such procedures include publication of security policies, control of access at the physical, software, and network levels, and separation of services.

12.2.2. Protecting against Data Corruption in the Network

Where the risk of undetected errors in datagrams delivered by the lower-layer transport services is considered to be too great, additional integrity protection is required. If this additional protection were provided in the application layer, the SCTP header would remain vulnerable to deliberate integrity attacks. While the existing SCTP mechanisms for detection of packet replays are considered sufficient for normal operation, stronger protections are needed to protect SCTP when the operating environment contains significant risk of deliberate attacks from a sophisticated adversary.

The SCTP Authentication extension SCTP-AUTH [RFC4895] **MAY** be used when the threat environment requires stronger integrity protections but does not require confidentiality.

12.2.3. Protecting Confidentiality

In most cases, the risk of breach of confidentiality applies to the signaling data payload, not to the SCTP or lower-layer protocol overheads. If that is true, encryption of the SCTP user data only might be considered. As with the supplementary checksum service, user data encryption **MAY** be performed by the SCTP user application. [RFC6083] **MAY** be used for this. Alternately, the user application **MAY** use an implementation-specific API to request that the IP Encapsulating Security Payload (ESP) [RFC4303] be used to provide confidentiality and integrity.

Particularly for mobile users, the requirement for confidentiality might include the masking of IP addresses and ports. In this case, ESP **SHOULD** be used instead of application-level confidentiality. If ESP is used to protect confidentiality of SCTP traffic, an ESP cryptographic transform that includes cryptographic integrity protection **MUST** be used, because, if there is a confidentiality threat, there will also be a strong integrity threat.

Regardless of where confidentiality is provided, the Internet Key Exchange Protocol version 2 (IKEv2) [RFC7296] **SHOULD** be used for key management of ESP.

Operators might consult [RFC4301] for more information on the security services available at and immediately above the Internet Protocol layer.

12.2.4. Protecting against Blind Denial-of-Service Attacks

A blind attack is one where the attacker is unable to intercept or otherwise see the content of data flows passing to and from the target SCTP node. Blind denial-of-service attacks can take the form of flooding, masquerade, or improper monopolization of services.

12.2.4.1. Flooding

The objective of flooding is to cause loss of service and incorrect behavior at target systems through resource exhaustion, interference with legitimate transactions, and exploitation of buffer-related software bugs. Flooding can be directed either at the SCTP node or at resources in the intervening IP Access Links or the Internet. Where the latter entities are the target, flooding will manifest itself as loss of network services, including potentially the breach of any firewalls in place.

In general, protection against flooding begins at the equipment design level, where it includes measures such as:

- avoiding commitment of limited resources before determining that the request for service is legitimate.
- giving priority to completion of processing in progress over the acceptance of new work.
- identification and removal of duplicate or stale queued requests for service.
- not responding to unexpected packets sent to non-unicast addresses.

Network equipment is expected to be capable of generating an alarm and log if a suspicious increase in traffic occurs. The log provides information, such as the identity of the incoming link and source address(es) used, which will help the network or SCTP system operator to take protective measures. Procedures are expected to be in place for the operator to act on such alarms if a clear pattern of abuse emerges.

The design of SCTP is resistant to flooding attacks, particularly in its use of a four-way startup handshake, its use of a cookie to defer commitment of resources at the responding SCTP node until the handshake is completed, and its use of a Verification Tag to prevent insertion of extraneous packets into the flow of an established association.

ESP might be useful in reducing the risk of certain kinds of denial-of-service attacks.

Support for the Host Name Address parameter has been removed from the protocol. Endpoints receiving INIT or INIT ACK chunks containing the Host Name Address parameter **MUST** send an ABORT chunk in response and **MAY** include an "Unresolvable Address" error cause.

12.2.4.2. Blind Masquerade

Masquerade can be used to deny service in several ways:

- by tying up resources at the target SCTP node to which the impersonated node has limited access. For example, the target node can by policy permit a maximum of one SCTP association with the impersonated SCTP node. The masquerading attacker can attempt to establish an association purporting to come from the impersonated node so that the latter cannot do so when it requires it.
- by deliberately allowing the impersonation to be detected, thereby provoking counter-measures that cause the impersonated node to be locked out of the target SCTP node.
- by interfering with an established association by inserting extraneous content such as a SHUTDOWN chunk.

SCTP reduces the risk of blind masquerade attacks through IP spoofing by use of the four-way startup handshake. Because the initial exchange is memoryless, no lockout mechanism is triggered by blind masquerade attacks. In addition, the packet containing the INIT ACK chunk with the State Cookie is transmitted back to the IP address from which it received the packet containing the INIT chunk. Thus, the attacker would not receive the INIT ACK chunk containing the State Cookie. SCTP protects against insertion of extraneous packets into the flow of an established association by use of the Verification Tag.

Logging of received INIT chunks and abnormalities, such as unexpected INIT ACK chunks, might be considered as a way to detect patterns of hostile activity. However, the potential usefulness of such logging has to be weighed against the increased SCTP startup processing it implies, rendering the SCTP node more vulnerable to flooding attacks. Logging is pointless without the establishment of operating procedures to review and analyze the logs on a routine basis.

12.2.4.3. Improper Monopolization of Services

Attacks under this heading are performed openly and legitimately by the attacker. They are directed against fellow users of the target SCTP node or of the shared resources between the attacker and the target node. Possible attacks include the opening of a large number of associations between the attacker's node and the target or transfer of large volumes of information within a legitimately established association.

Policy limits are expected to be placed on the number of associations per adjoining SCTP node. SCTP user applications are expected to be capable of detecting large volumes of illegitimate or "no-op" messages within a given association and either logging or terminating the association as a result, based on local policy.

12.3. SCTP Interactions with Firewalls

It is helpful for some firewalls if they can inspect just the first fragment of a fragmented SCTP packet and unambiguously determine whether it corresponds to an INIT chunk (for further information, please refer to [RFC1858]). Accordingly, we stress the requirements, as stated in Section 3.1, that (1) an INIT chunk **MUST NOT** be bundled with any other chunk in a packet and (2) a packet containing an INIT chunk **MUST** have a zero Verification Tag. The receiver of an INIT chunk **MUST** silently discard the INIT chunk and all further chunks if the INIT chunk is bundled with other chunks or the packet has a non-zero Verification Tag.

12.4. Protection of Non-SCTP-capable Hosts

To provide a non-SCTP-capable host with the same level of protection against attacks as for SCTP-capable ones, all SCTP implementations **MUST** implement the ICMP handling described in Section 10.

When an SCTP implementation receives a packet containing multiple control or DATA chunks and the processing of the packet would result in sending multiple chunks in response, the sender of the response chunk(s) **MUST NOT** send more than one packet containing chunks other than DATA chunks. This requirement protects the network for triggering a packet burst in response to a single packet. If bundling is supported, multiple response chunks that fit into a single packet **MAY** be bundled together into one single response packet. If bundling is not supported, then the sender **MUST NOT** send more than one response chunk and **MUST** discard all other responses. Note that this rule does not apply to a SACK chunk, since a SACK chunk is, in itself, a response to DATA chunks, and a SACK chunk does not require a response of more DATA chunks.

An SCTP implementation **MUST** abort the association if it receives a SACK chunk acknowledging a TSN that has not been sent.

An SCTP implementation that receives an INIT chunk that would require a large packet in response, due to the inclusion of multiple "Unrecognized Parameter" parameters, **MAY** (at its discretion) elect to omit some or all of the "Unrecognized Parameter" parameters to reduce the size of the INIT ACK chunk. Due to a combination of the size of the State Cookie parameter and the number of addresses a receiver of an INIT chunk indicates to a peer, it is always possible that

the INIT ACK chunk will be larger than the original INIT chunk. An SCTP implementation **SHOULD** attempt to make the INIT ACK chunk as small as possible to reduce the possibility of byte amplification attacks.

13. Network Management Considerations

The MIB module for SCTP defined in [RFC3873] applies for the version of the protocol specified in this document.

14. Recommended Transmission Control Block (TCB) Parameters

This section details a set of parameters that are expected to be contained within the TCB for an implementation. This section is for illustrative purposes and is not considered to be requirements on an implementation or as an exhaustive list of all parameters inside an SCTP TCB. Each implementation might need its own additional parameters for optimization.

14.1. Parameters Necessary for the SCTP Instance

Associations: A list of current associations and mappings to the data consumers for each association. This might be in the form of a hash table or other implementation-dependent structure. The data consumers might be process identification information, such as file descriptors, named pipe pointer, or table pointers dependent on how SCTP is implemented.

Secret Key: A secret key used by this endpoint to compute the MAC. This **SHOULD** be a cryptographic quality random number with a sufficient length. Discussion in [RFC4086] can be helpful in selection of the key.

Address List: The list of IP addresses that this instance has bound. This information is passed to one's peer(s) in INIT and INIT ACK chunks.

SCTP Port: The local SCTP port number to which the endpoint is bound.

14.2. Parameters Necessary per Association (i.e., the TCB)

Peer Verification Tag: Tag value to be sent in every packet and is received in the INIT or INIT ACK chunk.

My Verification Tag: Tag expected in every inbound packet and sent in the INIT or INIT ACK chunk.

State: COOKIE-WAIT, COOKIE-ECHOED, ESTABLISHED, SHUTDOWN-PENDING, SHUTDOWN-SENT, SHUTDOWN-RECEIVED, SHUTDOWN-ACK-SENT.

Note: No "CLOSED" state is illustrated, since, if an association is "CLOSED", its TCB **SHOULD** be removed.

Peer Transport Address List: A list of SCTP transport addresses to which the peer is bound. This information is derived from the INIT or INIT ACK chunk and is used to associate an inbound packet with a given association. Normally, this information is hashed or keyed for quick lookup and access of the TCB.

Primary Path: This is the current primary destination transport address of the peer endpoint. It might also specify a source transport address on this endpoint.

Overall Error Count: The overall association error count.

Overall Error Threshold: The threshold for this association that, if the Overall Error Count reaches, will cause this association to be torn down.

Peer Rwnd: Current calculated value of the peer's rwnd.

Next TSN: The next TSN number to be assigned to a new DATA chunk. This is sent in the INIT or INIT ACK chunk to the peer and incremented each time a DATA chunk is assigned a TSN (normally, just prior to transmit or during fragmentation).

Last Rcvd TSN: This is the last TSN received in 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.

Mapping Array: An array of bits or bytes indicating which out-of-order TSNs 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 might be in the form of a circular buffer or bit array.

Ack State: This flag indicates if the next received packet is to be responded to with a SACK chunk. This is initialized to 0. When a packet is received, it is incremented. If this value reaches 2 or more, a SACK chunk is sent and the value is reset to 0. Note: This is used only when no DATA chunks are received out of order. When DATA chunks are out of order, SACK chunks are not delayed (see Section 6).

Inbound Streams: An array of structures to track the inbound streams, normally including the next sequence number expected and possibly the stream number.

Outbound Streams: An array of structures to track the outbound streams, normally including the next sequence number to be sent on the stream.

Reasm Queue: A reassembly queue.

Receive Buffer: A buffer to store received user data that has not been delivered to the upper layer.

Local Transport Address List: The list of local IP addresses bound in to this association.

Association Maximum DATA Chunk Size: The smallest Path Maximum DATA Chunk Size of all destination addresses.

14.3. Per Transport Address Data

For each destination transport address in the peer's address list derived from the INIT or INIT ACK chunk, a number of data elements need to be maintained, including:

Error Count: The current error count for this destination.

Error Threshold: Current error threshold for this destination, i.e., what value marks the destination down if error count reaches this value.

cwnd: The current congestion window.

ssthresh: The current ssthresh value.

RTO: The current retransmission timeout value.

SRTT: The current smoothed round-trip time.

RTTVAR: The current RTT variation.

partial bytes acked: The tracking method for increase of cwnd when in congestion avoidance mode (see Section 7.2.2).

state: The current state of this destination, i.e., DOWN, UP, ALLOW-HEARTBEAT, NO-HEARTBEAT, etc.

PMTU: The current known PMTU.

PMDCS: The current known PMDCS.

Per Destination Timer: A timer used by each destination.

RTO-Pending: A flag used to track if one of the DATA chunks sent to this address is currently being used to compute an RTT. If this flag is 0, the next DATA chunk sent to this destination is expected to be used to compute an RTT and this flag is expected to be set. Every time the RTT calculation completes (i.e., the DATA chunk is acknowledged), clear this flag.

last-time: The time to which this destination was last sent. This can be used to determine if the sending of a HEARTBEAT chunk is needed.

14.4. General Parameters Needed

Out Queue: A queue of outbound DATA chunks.

In Queue: A queue of inbound DATA chunks.

15. IANA Considerations

This document defines five registries that IANA maintains:

- through definition of additional chunk types,
- through definition of additional chunk flags,
- through definition of additional parameter types,
- through definition of additional cause codes within ERROR chunks, or
- through definition of additional payload protocol identifiers.

IANA has performed the following updates for the above five registries:

- In the "Chunk Types" registry, IANA has replaced the registry reference to [RFC4960] and [RFC6096] with a reference to this document.

  In addition, in the Notes section, the reference to Section 3.2 of [RFC6096] has been updated with a reference to Section 15.2 of this document.

  Finally, each reference to [RFC4960] has been replaced with a reference to this document for the following chunk types:
  - Payload Data (DATA)
  - Initiation (INIT)
  - Initiation Acknowledgement (INIT ACK)
  - Selective Acknowledgement (SACK)
  - Heartbeat Request (HEARTBEAT)
  - Heartbeat Acknowledgement (HEARTBEAT ACK)
  - Abort (ABORT)
  - Shutdown (SHUTDOWN)
  - Shutdown Acknowledgement (SHUTDOWN ACK)
  - Operation Error (ERROR)
  - State Cookie (COOKIE ECHO)
  - Cookie Acknowledgement (COOKIE ACK)
  - Reserved for Explicit Congestion Notification Echo (ECNE)
  - Reserved for Congestion Window Reduced (CWR)
  - Shutdown Complete (SHUTDOWN COMPLETE)
  - Reserved for IETF-defined Chunk Extensions

- In the "Chunk Parameter Types" registry, IANA has replaced the registry reference to [RFC4960] with a reference to this document.

  IANA has changed the name of the "Unrecognized Parameters" chunk parameter type to "Unrecognized Parameter" in the "Chunk Parameter Types" registry.

In addition, each reference to [RFC4960] has been replaced with a reference to this document for the following chunk parameter types:

- Heartbeat Info
- IPv4 Address
- IPv6 Address
- State Cookie
- Unrecognized Parameter
- Cookie Preservative
- Host Name Address
- Supported Address Types

IANA has added a reference to this document for the following chunk parameter type:

- Reserved for ECN Capable (0x8000)

Also, IANA has added the value 65535 to be reserved for IETF-defined extensions.

- In the "Chunk Flags" registry, IANA replaced the registry reference to [RFC6096] with a reference to this document.

  In addition, each reference to [RFC4960] has been replaced with a reference to this document for the following DATA chunk flags:

  - E bit
  - B bit
  - U bit

  IANA has also replaced the reference to [RFC7053] with a reference to this document for the following DATA chunk flag:

  - I bit

  IANA has replaced the reference to [RFC4960] with a reference to this document for the following ABORT chunk flag:

  - T bit

  IANA has replaced the reference to [RFC4960] with a reference to this document for the following SHUTDOWN COMPLETE chunk flag:
  - T bit

- In the "Error Cause Codes" registry, IANA has replaced the registry reference to [RFC4960] with a reference to this document.

  IANA has changed the name of the "User Initiated Abort" error cause to "User-Initiated Abort" and the name of the "Stale Cookie Error" error cause to "Stale Cookie" in the "Error Cause Codes" registry.

In addition, each reference to [RFC4960] has been replaced with a reference to this document for the following cause codes:

- Invalid Stream Identifier
- Missing Mandatory Parameter
- Stale Cookie
- Out of Resource
- Unresolvable Address
- Unrecognized Chunk Type
- Invalid Mandatory Parameter
- Unrecognized Parameters
- No User Data
- Cookie Received While Shutting Down
- Restart of an Association with New Addresses

IANA has also replaced each reference to [RFC4460] with a reference to this document for the following cause codes:

- User-Initiated Abort
- Protocol Violation

- In the "SCTP Payload Protocol Identifiers" registry, IANA has replaced the registry reference to [RFC4960] with a reference to this document.

  IANA has replaced the reference to [RFC4960] with a reference to this document for the following SCTP payload protocol identifier:

  - Reserved by SCTP

SCTP requires that the IANA "Port Numbers" registry be opened for SCTP port registrations; Section 15.6 describes how. An IESG-appointed Expert Reviewer supports IANA in evaluating SCTP port allocation requests.

In the "Service Name and Transport Protocol Port Number Registry", IANA has replaced each reference to [RFC4960] with a reference to this document for the following SCTP port numbers:

- 9 (discard)
- 20 (ftp-data)
- 21 (ftp)
- 22 (ssh)
- 80 (http)
- 179 (bgp)
- 443 (https)

Furthermore, in the "Hypertext Transfer Protocol (HTTP) Digest Algorithm Values" registry, IANA has replaced the reference to Appendix B of [RFC4960] with a reference to Appendix A of this document.

In addition, in the "ONC RPC Netids (Standards Action)" registry, IANA has replaced each reference to [RFC4960] with a reference to this document for the following netids:

- sctp
- sctp6

In the "IPFIX Information Elements" registry, IANA has replaced each reference to [RFC4960] with a reference to this document for the following elements with the name:

- sourceTransportPort
- destinationTransportPort
- collectorTransportPort
- exporterTransportPort
- postNAPTSourceTransportPort
- postNAPTDestinationTransportPort

15.1. IETF-Defined Chunk Extension

The assignment of new chunk type codes is done through an IETF Review action, as defined in [RFC8126]. Documentation for a new chunk **MUST** contain the following information:

a) A long and short name for the new chunk type.

b) A detailed description of the structure of the chunk, which **MUST** conform to the basic structure defined in Section 3.2.

c) A detailed definition and description of intended use of each field within the chunk, including the chunk flags if any. Defined chunk flags will be used as initial entries in the chunk flags table for the new chunk type.

d) A detailed procedural description of the use of the new chunk type within the operation of the protocol.

The last chunk type (255) is reserved for future extension if necessary.

For each new chunk type, IANA creates a registration table for the chunk flags of that type. The procedure for registering particular chunk flags is described in Section 15.2.

15.2. IETF-Defined Chunk Flags Registration

The assignment of new chunk flags is done through an RFC Required action, as defined in [RFC8126]. Documentation for the chunk flags **MUST** contain the following information:

a) A name for the new chunk flag.

b) A detailed procedural description of the use of the new chunk flag within the operation of the protocol. It **MUST** be considered that implementations not supporting the flag will send 0 on transmit and just ignore it on receipt.

IANA selects a chunk flags value. This **MUST** be one of 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, or 0x80, which **MUST** be unique within the chunk flag values for the specific chunk type.

15.3. IETF-Defined Chunk Parameter Extension

The assignment of new chunk parameter type codes is done through an IETF Review action, as defined in [RFC8126]. Documentation of the chunk parameter **MUST** contain the following information:

a) Name of the parameter type.

b) Detailed description of the structure of the parameter field. This structure **MUST** conform to the general Type-Length-Value format described in Section 3.2.1.

c) Detailed definition of each component of the parameter value.

d) Detailed description of the intended use of this parameter type and an indication of whether and under what circumstances multiple instances of this parameter type can be found within the same chunk.

e) Each parameter type **MUST** be unique across all chunks.

15.4. IETF-Defined Additional Error Causes

Additional cause codes can be allocated through a Specification Required action as defined in [RFC8126]. Provided documentation **MUST** include the following information:

a) Name of the error condition.

b) Detailed description of the conditions under which an SCTP endpoint issues an ERROR (or ABORT) chunk with this cause code.

c) Expected action by the SCTP endpoint that receives an ERROR (or ABORT) chunk containing this cause code.

d) Detailed description of the structure and content of data fields that accompany this cause code.

The initial word (32 bits) of a cause code parameter **MUST** conform to the format shown in Section 3.3.10, that is:

- first 2 bytes contain the cause code value
- last 2 bytes contain the length of the error cause.

15.5. Payload Protocol Identifiers

The assignment of payload protocol identifiers is done using the First Come First Served policy, as defined in [RFC8126].

Except for value 0, which is reserved to indicate an unspecified payload protocol identifier in a DATA chunk, an SCTP implementation will not be responsible for standardizing or verifying any payload protocol identifiers. An SCTP implementation simply receives the identifier from the upper layer and carries it with the corresponding payload data.

The upper layer, i.e., the SCTP user, **SHOULD** standardize any specific protocol identifier with IANA if it is so desired. The use of any specific payload protocol identifier is out of the scope of this specification.

15.6. Port Numbers Registry

SCTP services can use contact port numbers to provide service to unknown callers, as in TCP and UDP. An IESG-appointed Expert Reviewer supports IANA in evaluating SCTP port allocation requests, according to the procedure defined in [RFC8126]. The details of this process are defined in [RFC6335].

16. Suggested SCTP Protocol Parameter Values

The following protocol parameters are **RECOMMENDED**:

RTO.Initial: 1 second

RTO.Min: 1 second

RTO.Max: 60 seconds

Max.Burst: 4

RTO.Alpha: 1/8

RTO.Beta: 1/4

Valid.Cookie.Life: 60 seconds

Association.Max.Retrans: 10 attempts

Path.Max.Retrans: 5 attempts (per destination address)

Max.Init.Retransmits: 8 attempts

HB.interval: 30 seconds

HB.Max.Burst: 1

SACK.Delay: 200 milliseconds

Implementation Note: The SCTP implementation can allow ULP to customize some of these protocol parameters (see Section 11).

'RTO.Min' **SHOULD** be set as described above in this section.

17. References

17.1. Normative References

[ITU.V42.1994] International Telecommunications Union, "Error-correcting Procedures for DCEs Using Asynchronous-to-Synchronous Conversion", ITU-T Recommendation V.42, 1994.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/info/rfc1123>.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/info/rfc1982>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <https://www.rfc-editor.org/info/rfc4895>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC6083] Tuexen, M., Seggelmann, R., and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", RFC 6083, DOI 10.17487/RFC6083, January 2011, <https://www.rfc-editor.org/info/rfc6083>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

[RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/info/rfc8899>.

17.2. Informative References

[FALL96] Fall, K. and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", SIGCOM 99, V. 26, N. 3, pp 5-21, July 1996.

[SAVAGE99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control with a Misbehaving Receiver", ACM Computer Communications Review 29(5), October 1999.

[ALLMAN99] Allman, M. and V. Paxson, "On Estimating End-to-End Network Path Properties", SIGCOM 99, October 1999.

[WILLIAMS93] Williams, R., "A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS", SIGCOM 99, August 1993, <https://archive.org/stream/PainlessCRC/crc_v3.txt>.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1858] Ziemba, G., Reed, D., and P. Traina, "Security Considerations for IP Fragment Filtering", RFC 1858, DOI 10.17487/RFC1858, October 1995, <https://www.rfc-editor.org/info/rfc1858>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2196] Fraser, B., "Site Security Handbook", FYI 8, RFC 2196, DOI 10.17487/RFC2196, September 1997, <https://www.rfc-editor.org/info/rfc2196>.

[RFC2522] Karn, P. and W. Simpson, "Photuris: Session-Key Management Protocol", RFC 2522, DOI 10.17487/RFC2522, March 1999, <https://www.rfc-editor.org/info/rfc2522>.

[RFC2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, DOI 10.17487/RFC2960, October 2000, <https://www.rfc-editor.org/info/rfc2960>.

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/info/rfc3465>.

[RFC3873] Pastor, J. and M. Belinchon, "Stream Control Transmission Protocol (SCTP) Management Information Base (MIB)", RFC 3873, DOI 10.17487/RFC3873, September 2004, <https://www.rfc-editor.org/info/rfc3873>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, DOI 10.17487/RFC4301, December 2005, <https://www.rfc-editor.org/info/rfc4301>.

[RFC4460] Stewart, R., Arias-Rodriguez, I., Poon, K., Caro, A., and M. Tuexen, "Stream Control Transmission Protocol (SCTP) Specification Errata and Issues", RFC 4460, DOI 10.17487/RFC4460, April 2006, <https://www.rfc-editor.org/info/rfc4460>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC6096] Tuexen, M. and R. Stewart, "Stream Control Transmission Protocol (SCTP) Chunk Flags Registration", RFC 6096, DOI 10.17487/RFC6096, January 2011, <https://www.rfc-editor.org/info/rfc6096>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <https://www.rfc-editor.org/info/rfc6458>.

[RFC6951] Tuexen, M. and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication", RFC 6951, DOI 10.17487/RFC6951, May 2013, <https://www.rfc-editor.org/info/rfc6951>.

[RFC7053] Tuexen, M., Ruengeler, I., and R. Stewart, "SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol", RFC 7053, DOI 10.17487/RFC7053, November 2013, <https://www.rfc-editor.org/info/rfc7053>.

[RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <https://www.rfc-editor.org/info/rfc8260>.

[RFC8261] Tuexen, M., Stewart, R., Jesup, R., and S. Loreto, "Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets", RFC 8261, DOI 10.17487/RFC8261, November 2017, <https://www.rfc-editor.org/info/rfc8261>.

[RFC8540] Stewart, R., Tuexen, M., and M. Proshin, "Stream Control Transmission Protocol: Errata and Issues in RFC 4960", RFC 8540, DOI 10.17487/RFC8540, February 2019, <https://www.rfc-editor.org/info/rfc8540>.

Appendix A. CRC32c Checksum Calculation

We define a 'reflected value' as one that is the opposite of the normal bit order of the machine. The 32-bit CRC (Cyclic Redundancy Check) is calculated, as described for CRC32c and uses the polynomial code 0x11EDC6F41 (Castagnoli93) or $x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+x^{0}$. The CRC is computed using a procedure similar to ETHERNET CRC [ITU.V42.1994], modified to reflect transport-level usage.

CRC computation uses polynomial division. A message bit-string M is transformed to a polynomial, M(X), and the CRC is calculated from M(X) using polynomial arithmetic.

When CRCs are used at the link layer, the polynomial is derived from on-the-wire bit ordering: the first bit 'on the wire' is the high-order coefficient. Since SCTP is a transport-level protocol, it cannot know the actual serial-media bit ordering. Moreover, different links in the path between SCTP endpoints can use different link-level bit orders.

A convention therefore is established for mapping SCTP transport messages to polynomials for purposes of CRC computation. The bit-ordering for mapping SCTP messages to polynomials is that bytes are taken most-significant first, but, within each byte, bits are taken least-significant first. The first byte of the message provides the eight highest coefficients. Within each byte, the least-significant SCTP bit gives the most-significant polynomial coefficient within that byte, and the most-significant SCTP bit is the least-significant polynomial coefficient in that byte. (This bit ordering is sometimes called 'mirrored' or 'reflected' [WILLIAMS93].) CRC polynomials are to be transformed back into SCTP transport-level byte values, using a consistent mapping.

The SCTP transport-level CRC value can be calculated as follows:

- CRC input data is assigned to a byte stream, numbered from 0 to N-1.
- The transport-level byte stream is mapped to a polynomial value. An N-byte PDU with j bytes numbered 0 to N-1 is considered as coefficients of a polynomial M(x) of order 8*N-1, with bit 0 of byte j being coefficient $x^{8*(N-j)-8}$ and bit 7 of byte j being coefficient $x^{8*(N-j)-1}$.
- The CRC remainder register is initialized with all 1s and the CRC is computed with an algorithm that simultaneously multiplies by x^{32} and divides by the CRC polynomial.
- The polynomial is multiplied by x^{32} and divided by G(x), the generator polynomial, producing a remainder R(x) of degree less than or equal to 31.
- The coefficients of R(x) are considered a 32-bit sequence.
- The bit sequence is complemented. The result is the CRC polynomial.
- The CRC polynomial is mapped back into SCTP transport-level bytes. The coefficient of x^{31} gives the value of bit 7 of SCTP byte 0, and the coefficient of x^{24} gives the value of bit 0 of byte 0. The coefficient of x^{7} gives bit 7 of byte 3, and the coefficient of x^{0} gives bit 0 of byte 3. The resulting 4-byte transport-level sequence is the 32-bit SCTP checksum value.

Implementation Note: Standards documents, textbooks, and vendor literature on CRCs often follow an alternative formulation, in which the register used to hold the remainder of the long-division algorithm is initialized to zero rather than all ones, and instead the first 32 bits of the message are complemented. The long-division algorithm used in our formulation is specified such that the initial multiplication by 2^{32} and the long-division are combined into one simultaneous operation. For such algorithms, and for messages longer than 64 bits, the two specifications are precisely equivalent. That equivalence is the intent of this document.

Implementors of SCTP are warned that both specifications are to be found in the literature, sometimes with no restriction on the long-division algorithm. The choice of formulation in this document is to permit non-SCTP usage, where the same CRC algorithm can be used to protect messages shorter than 64 bits.

There can be a computational advantage in validating the association against the Verification Tag, prior to performing a checksum, as invalid tags will result in the same action as a bad checksum in most cases. The exceptions for this technique would be packets containing INIT chunks and some SHUTDOWN-COMPLETE chunks, as well as a stale COOKIE ECHO chunks. These special-case exchanges represent small packets and will minimize the effect of the checksum calculation.

The following non-normative sample code is taken from an open-source CRC generator [WILLIAMS93], using the "mirroring" technique and yielding a lookup table for SCTP CRC32c with 256 entries, each 32 bits wide. While neither especially slow nor especially fast, as software table-lookup CRCs go, it has the advantage of working on both big-endian and little-endian CPUs, using the same (host-order) lookup tables, and using only the predefined ntohl() and htonl() operations. The code is somewhat modified from [WILLIAMS93] to ensure portability between big-endian and little-endian architectures, use fixed-sized types to allow portability between 32-bit and 64-bit

platforms, and use general C code improvements. (Note that, if the byte endian-ness of the target architecture is known to be little endian, the final bit-reversal and byte-reversal steps can be folded into a single operation.)

```
<CODE BEGINS>
/*****************************************************************/
/* Note: The definitions for Ross Williams's table generator    */
/* would be TB_WIDTH=4, TB_POLY=0x1EDC6F41, TB_REVER=TRUE.      */
/* For Mr. Williams's direct calculation code, use the settings */
/* cm_width=32, cm_poly=0x1EDC6F41, cm_init=0xFFFFFFFF,         */
/* cm_refin=TRUE, cm_refot=TRUE, cm_xorot=0x00000000.           */
/*****************************************************************/

/* Example of the crc table file */
#ifndef __crc32cr_h__
#define __crc32cr_h__

#define CRC32C_POLY 0x1EDC6F41UL
#define CRC32C(c,d) (c=(c>>8)^crc_c[(c^(d))&0xFF])

uint32_t crc_c[256] = {
  0x00000000UL, 0xF26B8303UL, 0xE13B70F7UL, 0x1350F3F4UL,
  0xC79A971FUL, 0x35F1141CUL, 0x26A1E7E8UL, 0xD4CA64EBUL,
  0x8AD958CFUL, 0x78B2DBCCUL, 0x6BE22838UL, 0x9989AB3BUL,
  0x4D43CFD0UL, 0xBF284CD3UL, 0xAC78BF27UL, 0x5E133C24UL,
  0x105EC76FUL, 0xE235446CUL, 0xF165B798UL, 0x030E349BUL,
  0xD7C45070UL, 0x25AFD373UL, 0x36FF2087UL, 0xC494A384UL,
  0x9A879FA0UL, 0x68EC1CA3UL, 0x7BBCEF57UL, 0x89D76C54UL,
  0x5D1D08BFUL, 0xAF768BBCUL, 0xBC267848UL, 0x4E4DFB4BUL,
  0x20BD8EDEUL, 0xD2D60DDDUL, 0xC186FE29UL, 0x33ED7D2AUL,
  0xE72719C1UL, 0x154C9AC2UL, 0x061C6936UL, 0xF477EA35UL,
  0xAA64D611UL, 0x580F5512UL, 0x4B5FA6E6UL, 0xB93425E5UL,
  0x6DFE410EUL, 0x9F95C20DUL, 0x8CC531F9UL, 0x7EAEB2FAUL,
  0x30E349B1UL, 0xC288CAB2UL, 0xD1D83946UL, 0x23B3BA45UL,
  0xF779DEAEUL, 0x05125DADUL, 0x1642AE59UL, 0xE4292D5AUL,
  0xBA3A117EUL, 0x4851927DUL, 0x5B016189UL, 0xA96AE28AUL,
  0x7DA08661UL, 0x8FCB0562UL, 0x9C9BF696UL, 0x6EF07595UL,
  0x417B1DBCUL, 0xB3109EBFUL, 0xA0406D4BUL, 0x522BEE48UL,
  0x86E18AA3UL, 0x748A09A0UL, 0x67DAFA54UL, 0x95B17957UL,
  0xCBA24573UL, 0x39C9C670UL, 0x2A993584UL, 0xD8F2B687UL,
  0x0C38D26CUL, 0xFE53516FUL, 0xED03A29BUL, 0x1F682198UL,
  0x5125DAD3UL, 0xA34E59D0UL, 0xB01EAA24UL, 0x42752927UL,
  0x96BF4DCCUL, 0x64D4CECFUL, 0x77843D3BUL, 0x85EFBE38UL,
  0xDBFC821CUL, 0x2997011FUL, 0x3AC7F2EBUL, 0xC8AC71E8UL,
  0x1C661503UL, 0xEE0D9600UL, 0xFD5D65F4UL, 0x0F36E6F7UL,
  0x61C69362UL, 0x93AD1061UL, 0x80FDE395UL, 0x72966096UL,
  0xA65C047DUL, 0x5437877EUL, 0x4767748AUL, 0xB50CF789UL,
  0xEB1FCBADUL, 0x197448AEUL, 0x0A24BB5AUL, 0xF84F3859UL,
  0x2C855CB2UL, 0xDEEEDFB1UL, 0xCDBE2C45UL, 0x3FD5AF46UL,
  0x7198540DUL, 0x83F3D70EUL, 0x90A324FAUL, 0x62C8A7F9UL,
  0xB602C312UL, 0x44694011UL, 0x5739B3E5UL, 0xA55230E6UL,
  0xFB410CC2UL, 0x092A8FC1UL, 0x1A7A7C35UL, 0xE811FF36UL,
  0x3CDB9BDDUL, 0xCEB018DEUL, 0xDDE0EB2AUL, 0x2F8B6829UL,
  0x82F63B78UL, 0x709DB87BUL, 0x63CD4B8FUL, 0x91A6C88CUL,
  0x456CAC67UL, 0xB7072F64UL, 0xA457DC90UL, 0x563C5F93UL,
  0x082F63B7UL, 0xFA44E0B4UL, 0xE9141340UL, 0x1B7F9043UL,
  0xCFB5F4A8UL, 0x3DDE77ABUL, 0x2E8E845FUL, 0xDCE5075CUL,
```

```
  0x92A8FC17UL, 0x60C37F14UL, 0x73938CE0UL, 0x81F80FE3UL,
  0x55326B08UL, 0xA759E80BUL, 0xB4091BFFUL, 0x466298FCUL,
  0x1871A4D8UL, 0xEA1A27DBUL, 0xF94AD42FUL, 0x0B21572CUL,
  0xDFEB33C7UL, 0x2D80B0C4UL, 0x3ED04330UL, 0xCCBBC033UL,
  0xA24BB5A6UL, 0x502036A5UL, 0x4370C551UL, 0xB11B4652UL,
  0x65D122B9UL, 0x97BAA1BAUL, 0x84EA524EUL, 0x7681D14DUL,
  0x2892ED69UL, 0xDAF96E6AUL, 0xC9A99D9EUL, 0x3BC21E9DUL,
  0xEF087A76UL, 0x1D63F975UL, 0x0E330A81UL, 0xFC588982UL,
  0xB21572C9UL, 0x407EF1CAUL, 0x532E023EUL, 0xA145813DUL,
  0x758FE5D6UL, 0x87E466D5UL, 0x94B49521UL, 0x66DF1622UL,
  0x38CC2A06UL, 0xCAA7A905UL, 0xD9F75AF1UL, 0x2B9CD9F2UL,
  0xFF56BD19UL, 0x0D3D3E1AUL, 0x1E6DCDEEUL, 0xEC064EEDUL,
  0xC38D26C4UL, 0x31E6A5C7UL, 0x22B65633UL, 0xD0DDD530UL,
  0x0417B1DBUL, 0xF67C32D8UL, 0xE52CC12CUL, 0x1747422FUL,
  0x49547E0BUL, 0xBB3FFD08UL, 0xA86F0EFCUL, 0x5A048DFFUL,
  0x8ECEE914UL, 0x7CA56A17UL, 0x6FF599E3UL, 0x9D9E1AE0UL,
  0xD3D3E1ABUL, 0x21B862A8UL, 0x32E8915CUL, 0xC083125FUL,
  0x144976B4UL, 0xE622F5B7UL, 0xF5720643UL, 0x07198540UL,
  0x590AB964UL, 0xAB613A67UL, 0xB831C993UL, 0x4A5A4A90UL,
  0x9E902E7BUL, 0x6CFBAD78UL, 0x7FAB5E8CUL, 0x8DC0DD8FUL,
  0xE330A81AUL, 0x115B2B19UL, 0x020BD8EDUL, 0xF0605BEEUL,
  0x24AA3F05UL, 0xD6C1BC06UL, 0xC5914FF2UL, 0x37FACCF1UL,
  0x69E9F0D5UL, 0x9B8273D6UL, 0x88D28022UL, 0x7AB90321UL,
  0xAE7367CAUL, 0x5C18E4C9UL, 0x4F48173DUL, 0xBD23943EUL,
  0xF36E6F75UL, 0x0105EC76UL, 0x12551F82UL, 0xE03E9C81UL,
  0x34F4F86AUL, 0xC69F7B69UL, 0xD5CF889DUL, 0x27A40B9EUL,
  0x79B737BAUL, 0x8BDCB4B9UL, 0x988C474DUL, 0x6AE7C44EUL,
  0xBE2DA0A5UL, 0x4C4623A6UL, 0x5F16D052UL, 0xAD7D5351UL,
};

#endif


/* Example of table build routine */

#include <stdio.h>
#include <stdlib.h>

#define OUTPUT_FILE   "crc32cr.h"
#define CRC32C_POLY    0x1EDC6F41UL

static FILE *tf;

static uint32_t
reflect_32(uint32_t b)
{
  int i;
  uint32_t rw = 0UL;

  for (i = 0; i < 32; i++) {
    if (b & 1)
      rw |= 1UL << (31 - i);
    b >>= 1;
  }
  return (rw);
}

static uint32_t
```

```
build_crc_table (int index)
{
  int i;
  uint32_t rb;

  rb = reflect_32(index);

  for (i = 0; i < 8; i++) {
    if (rb & 0x80000000UL)
      rb = (rb << 1) ^ (uint32_t)CRC32C_POLY;
    else
      rb <<= 1;
  }
  return (reflect_32(rb));
}

int
main (void)
{
  int i;

  printf("\nGenerating CRC32c table file <%s>.\n",
  OUTPUT_FILE);
  if ((tf = fopen(OUTPUT_FILE, "w")) == NULL) {
    printf("Unable to open %s.\n", OUTPUT_FILE);
    exit (1);
  }
  fprintf(tf, "#ifndef __crc32cr_h__\n");
  fprintf(tf, "#define __crc32cr_h__\n\n");
  fprintf(tf, "#define CRC32C_POLY 0x%08XUL\n",
    (uint32_t)CRC32C_POLY);
  fprintf(tf,
    "#define CRC32C(c,d) (c=(c>>8)^crc_c[(c^(d))&0xFF])\n");
  fprintf(tf, "\nuint32_t crc_c[256] =\n{\n");
  for (i = 0; i < 256; i++) {
    fprintf(tf, "0x%08XUL,", build_crc_table (i));
    if ((i & 3) == 3)
      fprintf(tf, "\n");
    else
      fprintf(tf, " ");
  }
  fprintf(tf, "};\n\n#endif\n");

  if (fclose(tf) != 0)
    printf("Unable to close <%s>.\n", OUTPUT_FILE);
  else
    printf("\nThe CRC32c table has been written to <%s>.\n",
      OUTPUT_FILE);
  return (0);
}

/* Example of crc insertion */

#include "crc32cr.h"

uint32_t
generate_crc32c(unsigned char *buffer, unsigned int length)
{
```

```
  unsigned int i;
  uint32_t crc32 = 0xffffffffUL;
  uint32_t result;
  uint32_t byte0, byte1, byte2, byte3;

  for (i = 0; i < length; i++) {
    CRC32C(crc32, buffer[i]);
  }

  result = ~crc32;

  /*  result now holds the negated polynomial remainder,
   *  since the table and algorithm are "reflected" [williams95].
   *  That is, result has the same value as if we mapped the message
   *  to a polynomial, computed the host-bit-order polynomial
   *  remainder, performed final negation, and then did an
   *  end-for-end bit-reversal.
   *  Note that a 32-bit bit-reversal is identical to four in-place
   *  8-bit bit-reversals followed by an end-for-end byteswap.
   *  In other words, the bits of each byte are in the right order,
   *  but the bytes have been byteswapped.  So, we now do an explicit
   *  byteswap.  On a little-endian machine, this byteswap and
   *  the final ntohl cancel out and could be elided.
   */

  byte0 = result & 0xff;
  byte1 = (result>>8) & 0xff;
  byte2 = (result>>16) & 0xff;
  byte3 = (result>>24) & 0xff;
  crc32 = ((byte0 << 24) |
           (byte1 << 16) |
           (byte2 << 8)  |
           byte3);
  return (crc32);
}

int
insert_crc32(unsigned char *buffer, unsigned int length)
{
  SCTP_message *message;
  uint32_t crc32;

  message = (SCTP_message *)buffer;
  message->common_header.checksum = 0UL;
  crc32 = generate_crc32c(buffer,length);
  /* and insert it into the message */
  message->common_header.checksum = htonl(crc32);
  return (1);
}

int
validate_crc32(unsigned char *buffer, unsigned int length)
{
  SCTP_message *message;
  unsigned int i;
  uint32_t original_crc32;
  uint32_t crc32;
```

```
   /* save and zero checksum */
   message = (SCTP_message *)buffer;
   original_crc32 = ntohl(message->common_header.checksum);
   message->common_header.checksum = 0L;
   crc32 = generate_crc32c(buffer, length);
   return ((original_crc32 == crc32) ? 1 : -1);
}
<CODE ENDS>
```

Acknowledgements

An undertaking represented by this updated document is not a small feat and represents the summation of the initial coauthors of [RFC2960]: Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson.

Add to that, the comments from everyone who contributed to [RFC2960]: Mark Allman, R. J. Atkinson, Richard Band, Scott Bradner, Steve Bellovin, Peter Butler, Ram Dantu, R. Ezhirpavai, Mike Fisk, Sally Floyd, Atsushi Fukumoto, Matt Holdrege, Henry Houh, Christian Huitema, Gary Lehecka, Jonathan Lee, David Lehmann, John Loughney, Daniel Luan, Barry Nagelberg, Thomas Narten, Erik Nordmark, Lyndon Ong, Shyamal Prasad, Kelvin Porter, Heinz Prantner, Jarno Rajahalme, Raymond E. Reeves, Renee Revis, Ivan Arias Rodriguez, A. Sankar, Greg Sidebottom, Brian Wyld, La Monte Yarroll, and many others for their invaluable comments.

Then, add the coauthors of [RFC4460]: I. Arias-Rodriguez, K. Poon, and A. Caro.

Then, add to these the efforts of all the subsequent seven SCTP interoperability tests and those who commented on [RFC4460], as shown in its acknowledgements: Barry Zuckerman, La Monte Yarroll, Qiaobing Xie, Wang Xiaopeng, Jonathan Wood, Jeff Waskow, Mike Turner, John Townsend, Sabina Torrente, Cliff Thomas, Yuji Suzuki, Manoj Solanki, Sverre Slotte, Keyur Shah, Jan Rovins, Ben Robinson, Renee Revis, Ian Periam, RC Monee, Sanjay Rao, Sujith Radhakrishnan, Heinz Prantner, Biren Patel, Nathalie Mouellic, Mitch Miers, Bernward Meyknecht, Stan McClellan, Oliver Mayor, Tomas Orti Martin, Sandeep Mahajan, David Lehmann, Jonathan Lee, Philippe Langlois, Karl Knutson, Joe Keller, Gareth Keily, Andreas Jungmaier, Janardhan Iyengar, Mutsuya Irie, John Hebert, Kausar Hassan, Fred Hasle, Dan Harrison, Jon Grim, Laurent Glaude, Steven Furniss, Atsushi Fukumoto, Ken Fujita, Steve Dimig, Thomas Curran, Serkan Cil, Melissa Campbell, Peter Butler, Rob Brennan, Harsh Bhondwe, Brian Bidulock, Caitlin Bestler, Jon Berger, Robby Benedyk, Stephen Baucke, Sandeep Balani, and Ronnie Sellar.

A special thanks to Mark Allman, who actually should have been a coauthor of [RFC4460] for his work on the max-burst but managed to wiggle out due to a technicality.

Also, we would like to acknowledge Lyndon Ong and Phil Conrad for their valuable input and many contributions.

Furthermore, you have [RFC4960] and those who have commented upon that, including Alfred Hönes and Ronnie Sellars.

Then, add the coauthor of [RFC8540]: Maksim Proshin.

And people who have commented on [RFC8540]: Pontus Andersson, Eric W. Biederman, Cedric Bonnet, Spencer Dawkins, Gorry Fairhurst, Benjamin Kaduk, Mirja Kühlewind, Peter Lei, Gyula Marosi, Lionel Morand, Jeff Morriss, Tom Petch, Kacheong Poon, Julien Pourtet, Irene Rüngeler, Michael Welzl, and Qiaobing Xie.

And, finally, the people who have provided comments for this document, including Gorry Fairhurst, Martin Duke, Benjamin Kaduk, Tero Kivinen, Eliot Lear, Marcelo Ricardo Leitner, David Mandelberg, John Preuß Mattsson, Claudio Porfiri, Maksim Proshin, Ines Robles, Timo Völker, Magnus Westerlund, and Zhouming.

Our thanks cannot be adequately expressed to all of you who have participated in the coding, testing, and updating process of this document. All we can say is, Thank You!

Authors' Addresses

Randall R. Stewart
Netflix, Inc.
2455 Heritage Green Ave
Davenport, FL 33837
United States of America
Email: randall@lakerest.net

Michael Tüxen
Münster University of Applied Sciences
Stegerwaldstrasse 39
48565 Steinfurt
Germany
Email: tuexen@fh-muenster.de

Karen E. E. Nielsen
Kamstrup A/S
Industrivej 28
DK-8660 Skanderborg
Denmark
Email: kee@kamstrup.com