Chapter 24

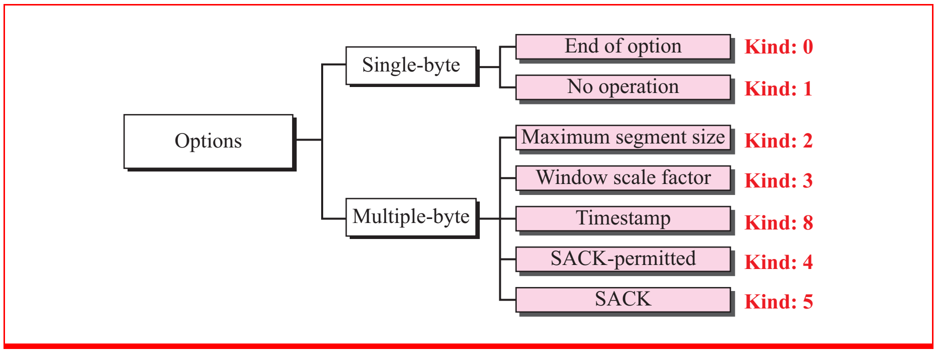
Extra Materials for Chapter 24

In this document, we discuss two topics that was briefly discuss in Chapter 24 of the textbook. These materials may be useful for those readers who need to work with TCP.

24.1 TCP OPTIONS

The TCP header can have up to 40 bytes of optional information. Options convey additional information to the destination or align other options. We can define two categories of options: 1-byte options and multiple-byte options. The first category contains two types of options: end of option list and no operation. The second category, in most implementations, contains five types of options: maximum segment size, window scale factor, timestamp, SACK-permitted, and SACK (see Figure 24-1).

Figure 24-1 *List of Options*



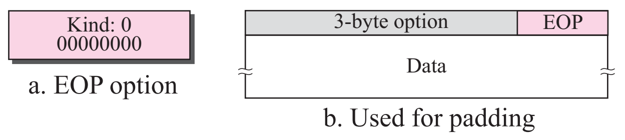
24.1.1 Single-Byte Options

There are two single-byte options: end-of-option and no-operation.

End of Option (EOP)

The **end-of-option (EOP) option** is a 1-byte option used for padding at the end of the option section. It can only be used as the last option. Only one occurrence of this option is allowed. After this option, the receiver looks for the payload data. Figure 24-2 shows an example. A 3-byte option is used after the header; the data section follows this option. One EOP option is inserted to align the data with the boundary of the next word.

Figure 24-2 *End-of-option*



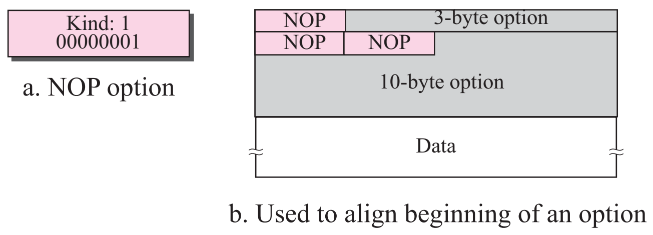
The EOP option imparts two pieces of information to the destination:

1. There are no more options in the header.
2. Data from the application program starts at the beginning of the next 32-bit word.

No Operation (NOP)

The **no-operation** (**NOP**) **option** is also a 1-byte option used as a filler. However, it normally comes before another option to help align it in a four-word slot. For example, in Figure 24-3 it is used to align one 3-byte option such as the window scale factor and one 10-byte option such as the timestamp.

Figure 24-3 *No-operation option*



24.1.2 Multiple-Byte Options

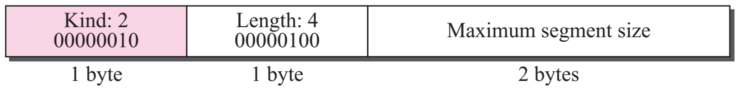
There are five multiple-byte options: maximum segment size, window scale factor, timestamp, SACK-permitted and SACK.

Maximum Segment Size (MSS)

The **maximum-segment-size option** defines the size of the biggest unit of data that can be received by the destination of the TCP segment. In spite of its name, it defines the maximum size of the data, not the maximum size of the segment. Since the field is 16 bits long, the value can be 0 to 65,535 bytes. Figure 24-4 shows the format of this option.

MSS is determined during connection establishment. Each party defines the MSS for the segments it will receive during the connection. If a party does not define this, the default values is 536 bytes.

Figure 24-4 *Maximum-segment-size option*



Window Scale Factor

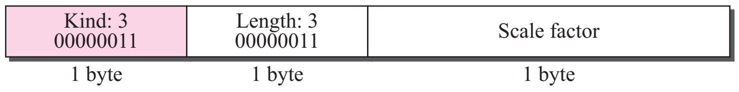
The window size field in the header defines the size of the sliding window. This field is 16 bits long, which means that the window can range from 0 to 65,535 bytes. Although this seems like a very large window size, it still may not be sufficient, especially if the data are traveling through a *long fat pipe,* a long channel with a wide bandwidth.

To increase the window size, a **window scale factor** is used. The new window size is found by first raising 2 to the number specified in the window scale factor. Then this result is multiplied by the value of the window size in the header.

$$\textbf{New window size} = \textbf{(window size defined in the header)} \times 2^{\textbf{(window scale factor)}}$$

Figure 24-5 shows the format of the window-scale-factor option.

Figure 24-5 *Window-scale-factor option*



The scale factor is sometimes called the *shift count* because multiplying a number by a power of 2 is the same as a left shift in a bitwise operation. In other words, the actual value of the window size can be determined by taking the value of the window size advertisement in the packet and shifting it to the left in the amount of the window scale factor.

For example, suppose the value of the window scale factor is 3. An end point receives an acknowledgment in which the window size is advertised as 32,768. The size of window this end can use is $32{,}768 \times 2^3$ or 262,144 bytes. The same value can be obtained if we shift the number 32,768 three bits to the left.

Although the scale factor could be as large as 255, the largest value allowed by TCP/IP is 14, which means that the maximum window size is $2^{16} \times 2^{14} = 2^{30}$, which is less than the maximum value for the sequence number. Note that the size of the window cannot be greater than the maximum value of the sequence number.

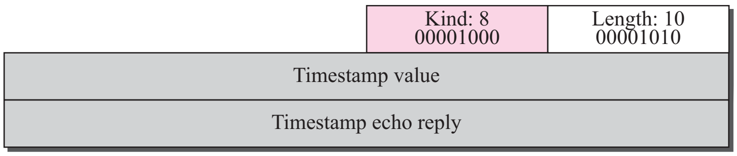
The window scale factor can also be determined only during the connection establishment phase. During data transfer, the size of the window (specified in the header) may be changed, but it must be multiplied by the same window scale factor.

Note that one end may set the value of the window scale factor to 0, which means that although it supports this option, it does not want to use it for this connection.

Timestamp

This is a 10-byte option with the format shown in Figure 24-6. Note that the end with the active open announces a timestamp in the connection request segment (SYN segment). If it receives a timestamp in the next segment (SYN + ACK) from the other end, it is allowed to use the timestamp; otherwise, it does not use it any more. The **timestamp option** has two applications: it measures the round-trip time and prevents wraparound sequence numbers.

Figure 24-6 *Timestamp option*



Measuring RTT

Timestamp can be used to measure the round-trip time (RTT). TCP, when ready to send a segment, reads the value of the system clock and inserts this value, a 32-bit number, in the timestamp value field. The receiver, when sending an acknowledgment for this segment or an cumulative acknowledgment that covers the bytes in this segment, copies the timestamp received in the timestamp echo reply. The sender, upon receiving the acknowledgment, subtracts the value of the timestamp echo reply from the time shown by the clock to find RTT.

Note that there is no need for the sender's and receiver's clocks to be synchronized because all calculations are based on the sender clock. Also note that the sender does not have to remember or store the time a segment left because this value is carried by the segment itself.

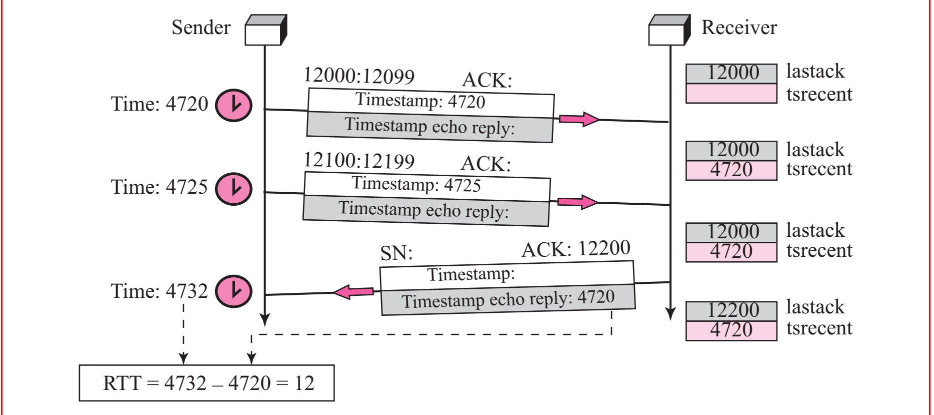
The receiver needs to keep track of two variables. The first, *lastack*, is the value of the last acknowledgment sent. The second, *tsrecent,* is the value of the recent timestamp that has not yet echoed. When the receiver receives a segment that contains the byte matching the value of *lastack*, it inserts the value of the timestamp field in the *tsrecent* variable. When it sends an acknowledgment, it inserts the value of tsrecent in the echo reply field.

Example 24.1

Figure 24-7 shows an example that calculates the round-trip time for one end. Everything must be flipped if we want to calculate the RTT for the other end. The sender simply inserts the value of the clock (for example, the number of seconds past midnight) in the timestamp field for the first and second segment. When an acknowledgment comes (the third segment), the value of the clock is checked and the value of the echo reply field is subtracted from the current time. RTT is 12 s in this scenario.

The receiver's function is more involved. It keeps track of the last acknowledgment sent (12000). When the first segment arrives, it contains the bytes 12000 to 12099. The first byte is the same as the value of *lastack*. It then copies the timestamp

Figure 24-7 *Example 24.1*



value (4720) into the *tsrecent* variable. The value of lastack is still 12000 (no new acknowledgment has been sent). When the second segment arrives, since none of the byte numbers in this segment include the value of *lastack,* the value of the timestamp field is ignored. When the receiver decides to send an cumulative acknowledgment with acknowledgment 12200, it changes the value of *lastack* to 12200 and inserts the value of tsrecent in the echo reply field. The value of tsrecent will not change until it is replaced by a new segment that carries byte 12200 (next segment).

Note that as the example shows, the RTT calculated is the time difference between sending the first segment and receiving the third segment. This is actually the meaning of RTT: the time difference between a packet sent and the acknowledgment received. The third segment carries the acknowledgment for the first and second segments.

PAWS

The timestamp option has another application, **protection against wrapped sequence numbers (PAWS).** The sequence number defined in the TCP protocol is only 32 bits long. Although this is a large number, it could be wrapped around in a high-speed connection. This implies that if a sequence number is *n* at one time, it could be *n* again during the lifetime of the same connection. Now if the first segment is duplicated and arrives during the second round of the sequence numbers, the segment belonging to the past is wrongly taken as the segment belonging to the new round.

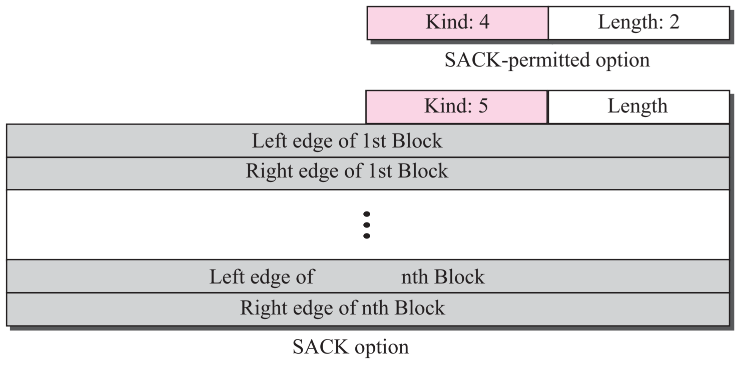
One solution to this problem is to increase the size of the sequence number, but this involves increasing the size of the window as well as the format of the segment and more. The easiest solution is to include the timestamp in the identification of a segment. In other words, the identity of a segment can be defined as the combination of timestamp and sequence number. This means increasing the size of the identification. Two segments 400:12,001 and 700:12,001 definitely belong to different incarnations. The first was sent at time 400, the second at time 700.

SACK-Permitted and SACK Options

As we discussed in the textbook, the acknowledgment field in the TCP segment is designed as an cumulative acknowledgment, which means it reports the receipt of the last consecutive byte: it does not report the bytes that have arrived out of order. It is also silent about duplicate segments. This may have a negative effect on TCP's performance. If some packets are lost or dropped, the sender must wait until a time-out and then send all packets that have not been acknowledged. The receiver may receive duplicate packets. To improve performance, selective acknowledgment (SACK) was proposed. Selective acknowledgment allows the sender to have a better idea of which segments are actually lost and which have arrived out of order. The new proposal even includes a list for duplicate packets. The sender can then send only those segments that are really lost. The list of duplicate segments can help the sender find the segments which have been retransmitted by a short time-out.

The proposal defines two new options: SACK-permitted and SACK as shown in Figure 24-8.

Figure 24-8 *SACK-permitted and SACK options*



The ***SACK*-permitted option** of two bytes is used only during connection establishment. The host that sends the SYN segment adds this option to show that it can support the SACK option. If the other end, in its SYN + ACK segment, also includes this option, then the two ends can use the SACK option during data transfer. Note that the SACK-permitted option is not allowed during the data transfer phase.

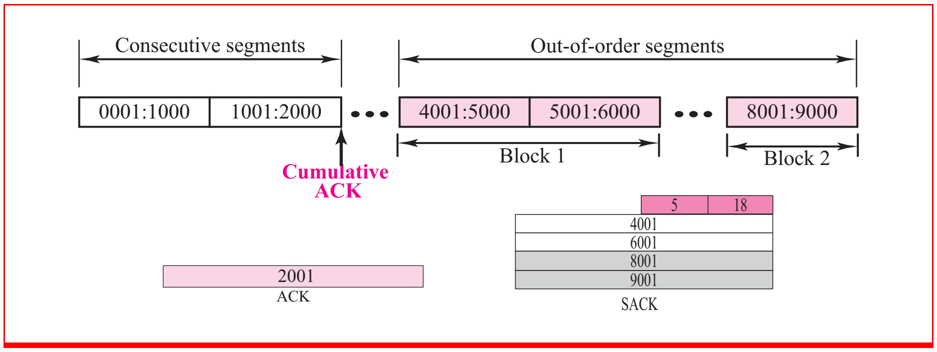
The **SACK option,** of variable length, is used during data transfer only if both ends agree (if they have exchanged SACK-permitted options during connection establishment). The option includes a list for blocks arriving out of order. Each block occupies two 32-bit numbers that define the beginning and the end of the blocks. We will show the use of this option in examples; for the moment, remember that the allowed size of an option in TCP is only 40 bytes. This means that a SACK option cannot define more than 4 blocks. The information for 5 blocks occupies $(5 \times 2) \times 4 + 2$ or 42 bytes, which is beyond the available size for the option section in a segment. If the SACK option is used with other options, then the number of blocks may be reduced.

The first block of the SACK option can be used to report the duplicates. This is used only if the implementation allows this feature.

Example 24.2

Let us see how the SACK option is used to list out-of-order blocks. In Figure 24-9 an end has received five segments of data.

Figure 24-9 *Example 24.2*



The first and second segments are in consecutive order. An cumulative acknowledgment can be sent to report the reception of these two segments. Segments 3, 4, and 5, however, are out of order with a gap between the second and third and a gap between the fourth and the fifth. An ACK and a SACK together can easily clear the situation for the sender. The value of ACK is 2001, which means that the sender need not worry about bytes 1 to 2000. The SACK has two blocks. The first block announces that bytes 4001 to 6000 have arrived out of order. The second block shows that bytes 8001 to 9000 have also arrived out of order. This means that bytes 2001 to 4000 and bytes 6001 to 8000 are lost or discarded. The sender can resend only these bytes.

Example 24.3

Figure 24-10 shows how a duplicate segment can be detected with a combination of ACK and SACK. In this case, we have some out-of-order segments (in one block) and one duplicate segment. To show both out-of-order and duplicate data, SACK uses the first block, in this case, to show the duplicate data and other blocks to show out-of-order data. Note that only the first block can be used for duplicate data. The natural question is how the sender, when it receives these ACK and SACK values, knows that the first block is for duplicate data (compare this example with the previous example). The answer is that the bytes in the first block are already acknowledged in the ACK field; therefore, this block must be a duplicate.

Example 24.4

Figure 24-11 shows what happens if one of the segments in the out-of-order section is also duplicated. In this example, one of the segments (4001:5000) is duplicated.

The SACK option announces this duplicate data first and then the out-of-order block. This time, however, the duplicated block is not yet acknowledged by ACK, but

Figure 24-10 *Example 24.3*

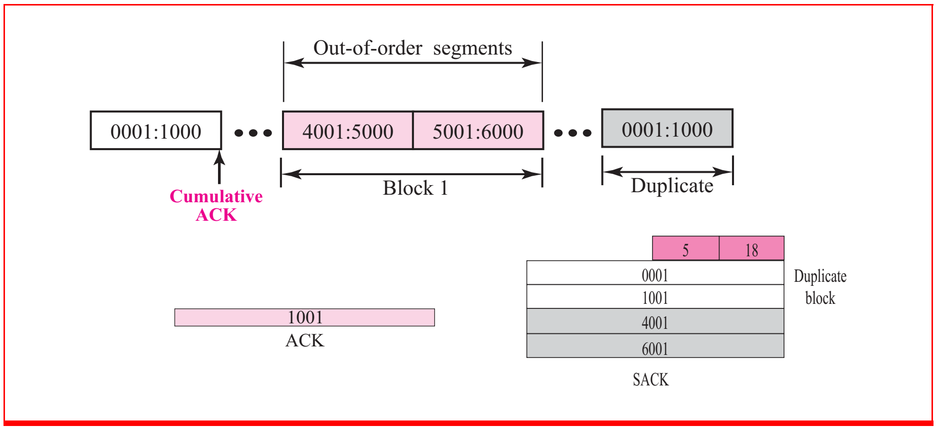
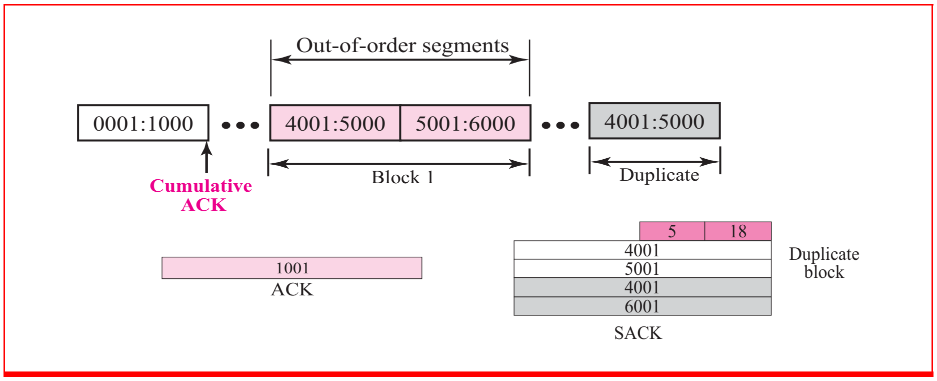


Figure 24-11 *Example 24.4*



because it is part of the out-of-order block (4001:5000 is part of 4001:6000), it is understood by the sender that it defines the duplicate data.

24.2 SCTP PACKETS

As we discussed in Chapter 24 of the textbook, an SCTP packet is made of a general header and several chunks.

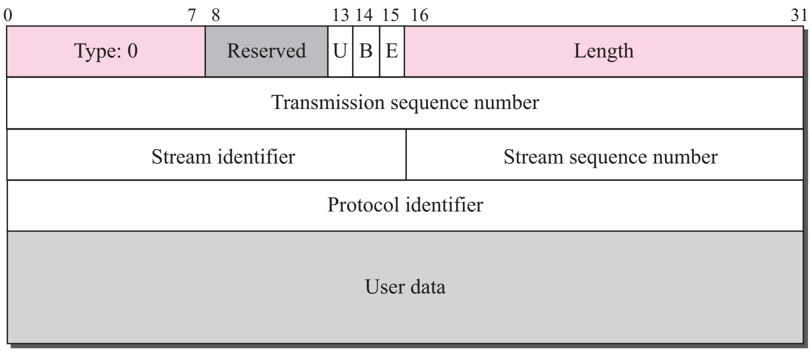
24.2.1 SCTP Chunks

SCTP defines fourteen types of chunks that we describe in this document.

DATA

The **DATA chunk** carries the user data. A packet may contain zero or more data chunks. Figure 24-12 shows the format of a DATA chunk.

Figure 24-12 *DATA chunk*



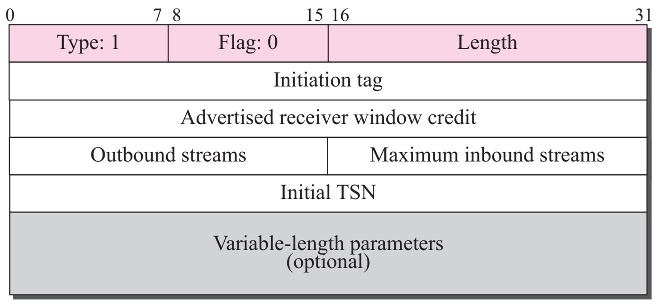
The descriptions of the common fields are the same. The type field has a value of 0. The flag field has 5 reserved bits and 3 defined bits: U, B, and E. The U (unordered) field, when set to 1, signals unordered data (explained later). In this case, the value of the stream sequence number is ignored. The B (beginning) and E (end) bits together define the position of a chunk in a message that is fragmented. When B = 1 and E = 1, there is no fragmentation (first and last); the whole message is carried in one chunk. When B = 1 and E = 0, it is the first fragment. When B = 0 and E = 1, it is the last fragment. When B = 0 and E = 0, it is a middle fragment (neither the first nor the last). Note that the value of the length field does not include padding. This value cannot be less than 17 because a DATA chunk must always carry at least one byte of data.

- **Transmission sequence number (TSN).** This 32-bit field defines the transmission sequence number. It is a sequence number that is initialized in an INIT chunk for one direction and in the INIT ACK chunk for the opposite direction.
- **Stream identifier (SI).** This 16-bit field defines each stream in an association. All chunks belonging to the same stream in one direction carry the same stream identifier.
- **Stream sequence number (SSN).** This 16-bit field defines a chunk in a particular stream in one direction.
- **Protocol identifier.** This 32-bit field can be used by the application program to define the type of data. It is ignored by the SCTP layer.
- **User data.** This field carries the actual user data. SCTP has some specific rules about the user data field. First, no chunk can carry data belonging to more than one message, but a message can be spread over several data chunks. Second, this field cannot be empty; it must have at least one byte of user data. Third, if the data cannot end at a 32-bit boundary, padding must be added. These padding bytes are not included in the value of the length field.

INIT

The **INIT chunk** (initiation chunk) is the first chunk sent by an end point to establish an association. The packet that carries this chunk cannot carry any other control or data chunks. The value of the verification tag for this packet is 0, which means no tag has yet been defined. The format is shown in Figure 24-13.

Figure 24-13 *INIT chunk*



The three common fields (type, flag, and length) are as before. The value of the type field is 1. The value of the flag field is zero (no flags); and the value of the length field is a minimum of 20 (more if there are optional parameters). The other fields are explained below:

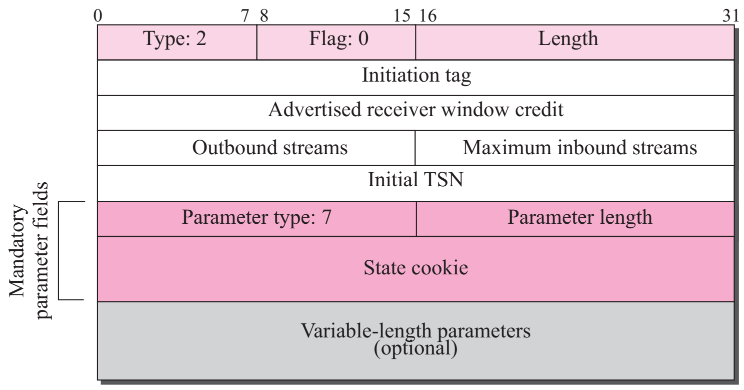
- **Initiation tag.** This 32-bit field defines the value of the verification tag for packets traveling in the opposite direction. As we mentioned before, all packets have a verification tag in the general header; this tag is the same for all packets traveling in one direction in an association. The value of this tag is determined during association establishment. The end point that initiates the association defines the value of this tag in the initiation tag field. This value is used as the verification tag in the rest of the packets sent from the other direction. For example, when end point A starts an association with end point B, A defines an initiation tag value, say *x*, which is used as the verification tag for all packets sent from B to A. The initiation tag is a random number between 0 and $2^{32} - 1$. The value of 0 defines no association and is permitted only by the general header of the INIT chunk.
- **Advertised receiver window credit.** This 32-bit field is used in flow control and defines the initial amount of data in bytes that the sender of the INIT chunk can allow. It is the *rwnd* value that will be used by the receiver to know how much data to send. Note that, in SCTP, sequence numbers are in terms of chunks.
- **Outbound stream.** This 16-bit field defines the number of streams that the initiator of the association suggests for streams in the outbound direction. It may be reduced by the other end point.
- **Maximum inbound stream.** This 16-bit field defines the maximum number of streams that the initiator of the association can support in the inbound direction. Note that this is a maximum number and cannot be increased by the other end point.
- **Initial TSN.** This 32-bit field initializes the transmission sequence number (TSN) in the outbound direction. Note that each data chunk in an association has to have one TSN. The value of this field is also a random number less than 2^{32}.

- **Variable-length parameters.** These optional parameters may be added to the INIT chunk to define the IP address of sending end point, the number of IP addresses the end point can support (multihome), the preservation of the cookie state, the type of addresses, and support of explicit congestion notification (ECN).

INIT ACK

The **INIT ACK chunk (**initiation acknowledgment chunk**)** is the second chunk sent during association establishment. The packet that carries this chunk cannot carry any other control or data chunks. The value of the verification tag for this packet (located in the general header) is the value of the initiation tag defined in the received INIT chunk. The format is shown in Figure 24-14.

Figure 24-14 *INIT ACK chunk*



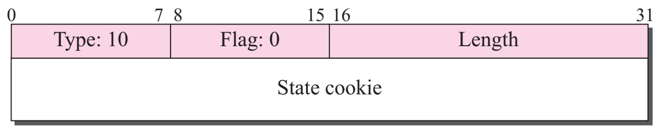
Note that the fields in the main part of the chunk are the same as those defined in the INIT chunk. However, a mandatory parameter is required for this chunk. The parameter of type 7 defines the state cookie sent by the sender of this chunk. The chunk can also have optional parameters. Note that the initiation tag field in this chunk initiates the value of the verification tag for future packets traveling from the opposite direction.

COOKIE ECHO

The **COOKIE ECHO chunk** is the third chunk sent during association establishment. It is sent by the end point that receives an INIT ACK chunk (normally the sender of the INIT chunk). The packet that carries this chunk can also carry user data. The format is shown in Figure 24-15.

Note that this is a very simple chunk of type 10. In the information section it echoes the state cookie that the end point has previously received in the INIT ACK. The receiver of the INIT ACK cannot open the cookie.

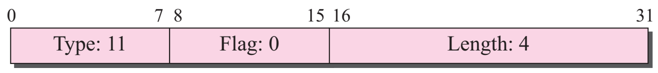
Figure 24-15 *COOKIE ECHO chunk*



COOKIE ACK

The **COOKIE ACK chunk** is the fourth and last chunk sent during association establishment. It is sent by an end point that receives a COOKIE ECHO chunk. The packet that carries this chunk can also carry user data. The format is shown in Figure 24-16.

Figure 24-16 *COOKIE ACK*

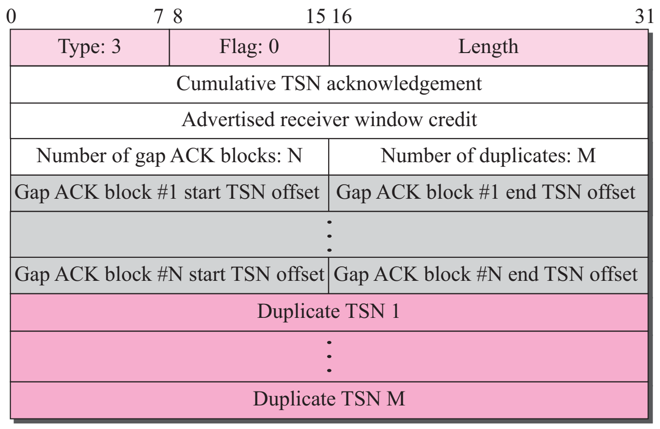


Note that this is a very simple chunk of type 11. The length of the chunk is exactly 4 bytes.

SACK

The **SACK chunk** (selective ACK chunk) acknowledges the receipt of data packets. Figure 24-17 shows the format of the SACK chunk.

Figure 24-17 *SACK chunk*


0
7 8
15 16
31
Type: 3
Flag: 0
Length
Cumulative TSN acknowledgement
Advertised receiver window credit
Number of gap ACK blocks: N
Number of duplicates: M
Gap ACK block #1 start TSN offset
Gap ACK block #1 end TSN offset
Gap ACK block #N start TSN offset
Gap ACK block #N end TSN offset
Duplicate TSN 1
Duplicate TSN M

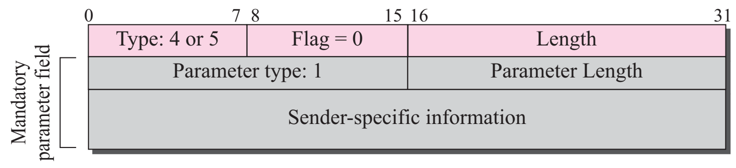
The common fields are the same as discussed previously. The type field has a value of 3. The flag bits are all set to 0s.

- **Cumulative TSN acknowledgment.** This 32-bit field defines the TSN of the last data chunk received in sequence.
- **Advertised receiver window credit.** This 32-bit field is the updated value for the receiver window size.
- **Number of gap ACK blocks.** This 16-bit field defines the number of gaps in the data chunk received after the cumulative TSN. Note that the term *gap* is misleading here: the gap defines the sequence of received chunks, not the missing chunks.
- **Number of duplicates.** This 16-bit field defines the number of duplicate chunks following the cumulative TSN.
- **Gap ACK block start offset.** For each gap block, this 16-bit field gives the starting TSN relative to the cumulative TSN.
- **Gap ACK block end offset.** For each gap block, this 16-bit field gives the ending TSN relative to the cumulative TSN.
- **Duplicate TSN.** For each duplicate chunk, this 32-bit field gives the TSN of the duplicate chunk.

HEARTBEAT and HEARTBEAT ACK

The **HEARTBEAT chunk** and **HEARTBEAT ACK chunk** are similar except for the type field. The first has a type of 4 and the second a type of 5. Figure 24-18 shows the format of these chunks. These two chunks are used to periodically probe the condition of an association. An end point sends a HEARTBEAT chunk; the peer responds with a HEARTBEAT ACK if it is alive. The format has the common three fields and mandatory parameter fields that provide sender-specific information. This information in the HEARTBEAT chunk includes the local time and the address of the sender. It is copied without change into the HEARTBEAT ACK chunk.

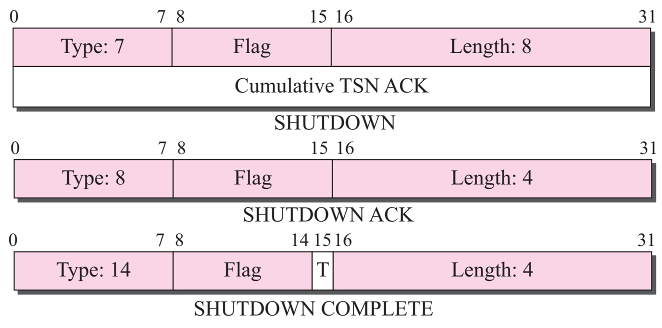
Figure 24-18 *HEARTBEAT and HEARTBEAT ACK chunks*



SHUTDOWN, SHUTDOWN ACK, and SHUTDOWN COMPLETE

These three chunks (used for closing an association) are similar. The **SHUTDOWN chunk,** type 7, is eight bytes in length; the second four bytes define the cumulative TSN. The **SHUTDOWN ACK chunk,** type 8, is four bytes in length. The **SHUTDOWN COMPLETE chunk,** type 14, is also 4 bytes long, and has a one bit flag, the T flag. The T flag shows that the sender does not have a TCB table. Figure 24-19 shows the formats.

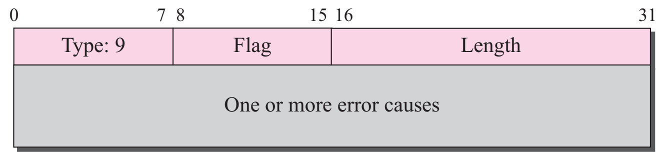
Figure 24-19 *SHUTDOWN, SHUTDOWN ACK, and SHUTDOWN COMPLETE chunks*



ERROR

The **ERROR chunk** is sent when an end point finds some error in a received packet. Note that the sending of an ERROR chunk does not imply the aborting of the association. (This would require an ABORT chunk.) Figure 24-20 shows the format of the ERROR chunk.

Figure 24-20 *ERROR chunk*



The errors are defined in Table 24.1.

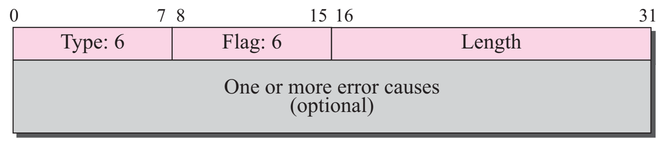
Table 24.1 *Errors*

Code	*Description*
1	Invalid stream identifier
2	Missing mandatory parameter
3	State cookie error
4	Out of resource
5	Unresolvable address
6	Unrecognized chunk type
7	Invalid mandatory parameters
8	Unrecognized parameter
9	No user data
10	Cookie received while shutting down

ABORT

The **ABORT chunk** is sent when an end point finds a fatal error and needs to abort the association. The error types are the same as those for the ERROR chunk (see Table 24.1). Figure 24-21 shows the format of an ABORT chunk.

Figure 24-21 *ABORT chunk*



FORWARD TSN

This is a chunk recently added to the standard (see RFC 3758) to inform the receiver to adjust its cumulative TSN. It provides partial reliable service.