CHAPTER 3

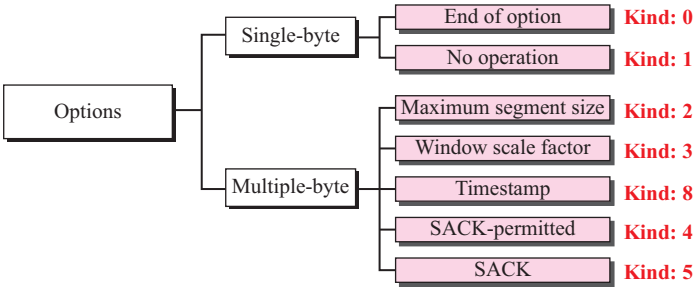
Extra Materials for Chapter 3

In this document, we discuss one topic that was briefly discuss in Chapter 3 of the textbook. These materials may be useful for those readers who need to work with TCP.

3.1 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 3.1).

Figure 3.1 *List of Options*



Single-Byte Options

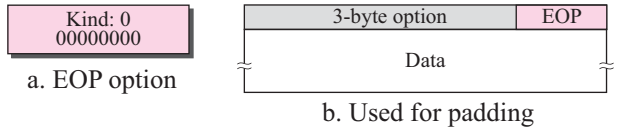
There are two single-byte options: end-of-option and no-operation.

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 3.2 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 3.2 *End-of-option*



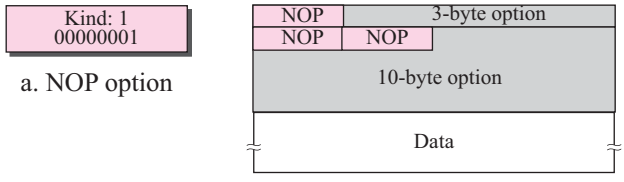
The EOP option imparts two pieces of information to the destination:

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 3.3 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 3.3 *No-operation option*



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 3.4 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 3.4 *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 3.5 shows the format of the window-scale-factor option.

Figure 3.5 *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.

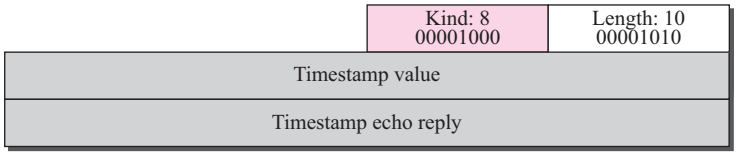
The window scale factor can also be determined only during the connection establishment phase. During data transfer, the size of the window (specified in the header) may be changed, but it must be multiplied by the same window scale factor.

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 3.6. 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 **time-stamp option** has two applications: it measures the round-trip time and prevents wraparound sequence numbers.

Figure 3.6 *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.

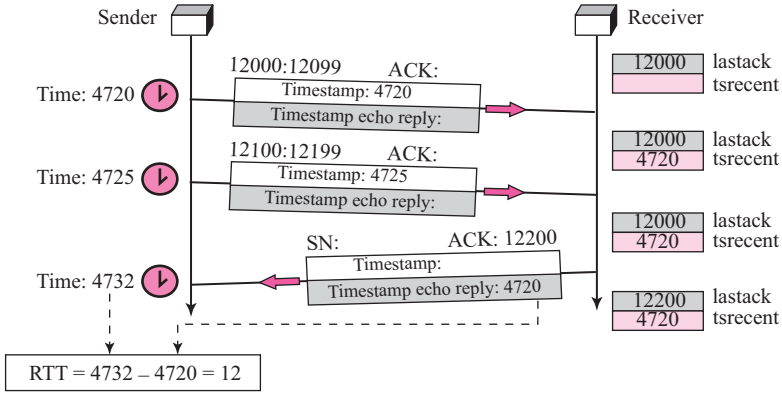
The receiver needs to keep track of two variables. The first, *lastack*, is the value of the last acknowledgment sent. The second, *tsrecent*, is the value of the recent timestamp that has not yet echoed. When the receiver receives a segment that contains the byte matching the value of *lastack*, it 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 3.1

Figure 3.7 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 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*,

Figure 3.7 *Example 3.1*



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.

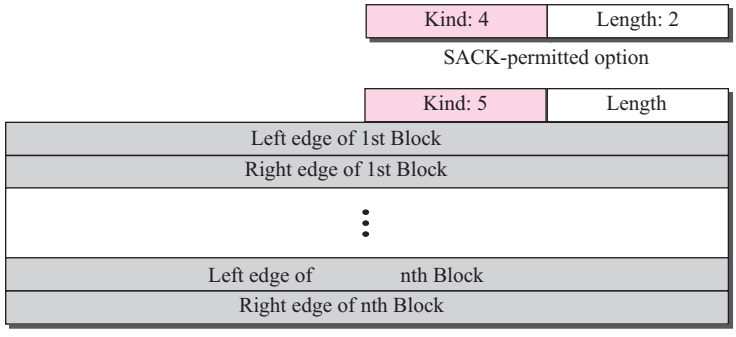
SACK-Permitted and SACK Options

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 3.8.

Figure 3.8 *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.

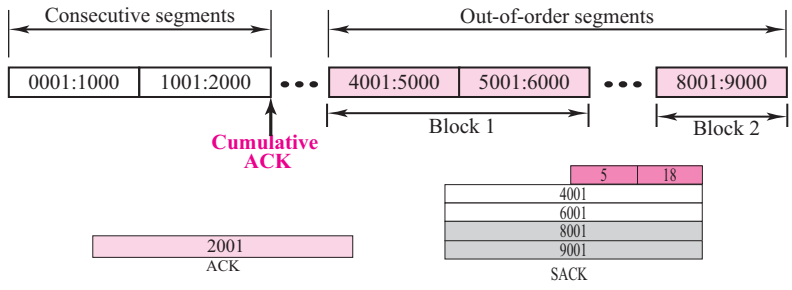
The first block of the SACK option can be used to report the duplicates. This is used only if the implementation allows this feature.

Example 3.2

Let us see how the SACK option is used to list out-of-order blocks. In Figure 3.9 an end has received five segments of data.

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

Figure 3.9 *Example 3.2*

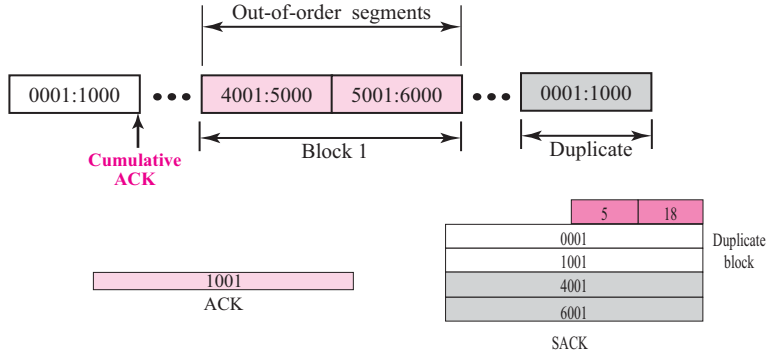


order with a gap between the second and third and a gap between the fourth and the fifth. An ACK and a SACK together can easily clear the situation for the sender. The value of ACK is 2001, which means that the sender need not worry about bytes 1 to 2000. The SACK has two blocks. The first block announces 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 3.3

Figure 3.10 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

Figure 3.10 *Example 3.3*

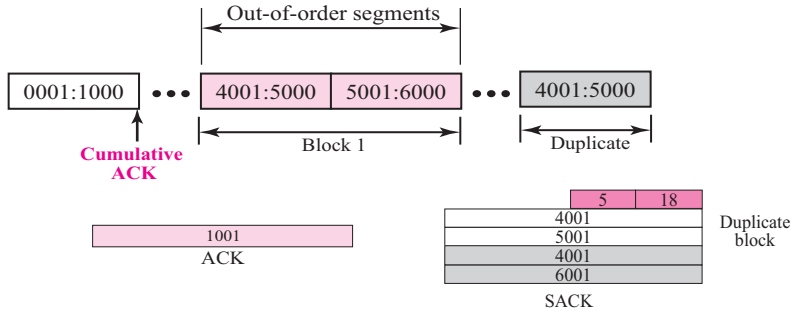


segment. To show both out-of-order and duplicate data, SACK uses the first block, in this case, to show the duplicate data and other blocks to show out-of-order data. Note that only the first block can be used for duplicate data. The natural question is how the sender, when it receives these ACK and 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 3.4

Figure 3.11 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.

Figure 3.11 *Example 3.4*



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 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.