Its TCP sends an RST+ACK segment and throws away all data in the queue. The server TCP also throws away all queued data and informs the server process via an error message. Both TCPs go to the **CLOSED** state immediately. Note that no ACK segment is generated in response to the RST segment.

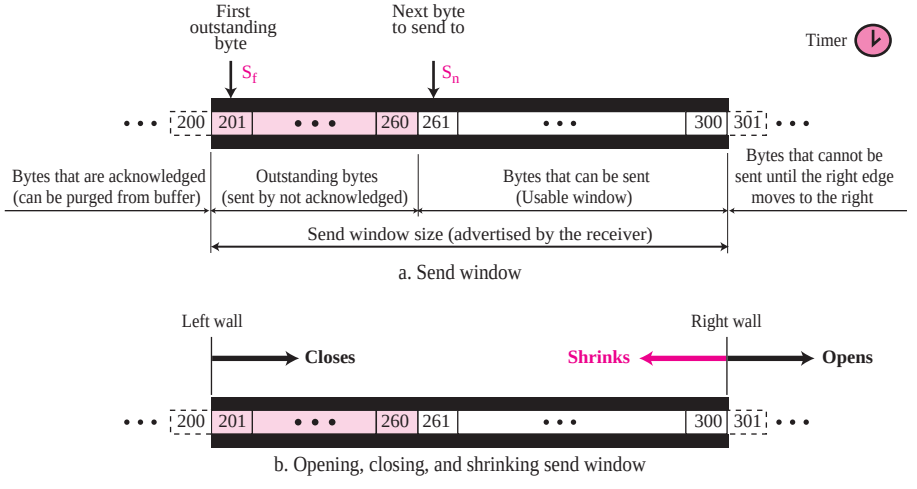
15.6 WINDOWS IN TCP

Before discussing data transfer in TCP and the issues such as flow, error, and congestion control, we describe the windows used in TCP. TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication. However, to make the discussion simple, we make an unrealistic assumption that communication is only unidirectional (say from client to server); the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

Send Window

Figure 15.22 shows an example of a send window. The window we have used is of size 100 bytes (normally thousands of bytes), but later we see that the send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control). The figure shows how a send window *opens*, *closes*, or *shrinks*.

Figure 15.22 *Send window in TCP*



The send window in TCP is similar to one used with the Selective Repeat protocol (Chapter 13), but with some differences:

1. One difference is the nature of entities related to the window. The window in SR numbers pockets, but the window in the TCP numbers bytes. Although actual

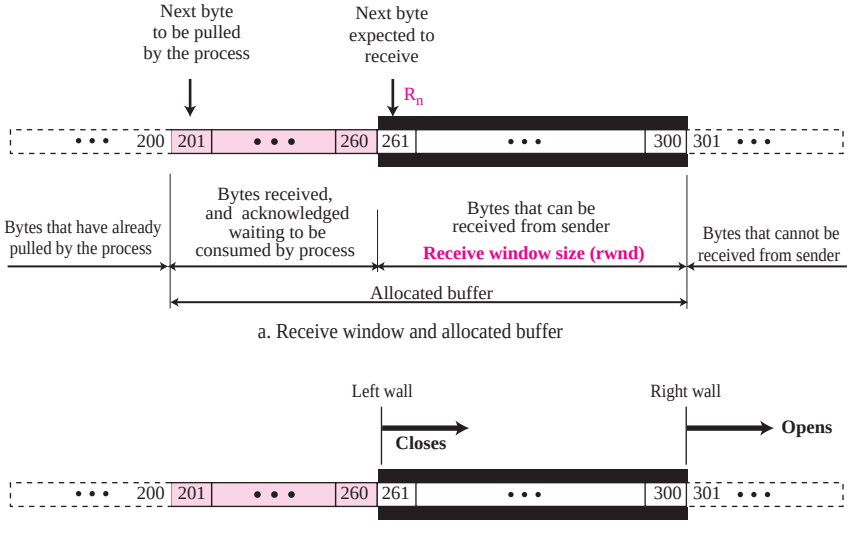
transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.

2. The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
3. Another difference is the number of timers. The theoretical Selective Repeat protocol may use several timers for each packet sent, but the TCP protocol uses only one timer. We later explain the use of this timer in error control.

Receive Window

Figure 15.23 shows an example of a receive window. The window we have used is of size 100 bytes (normally thousands of bytes). The figure also shows how the receive window opens and closes; in practice, the window should never shrink.

Figure 15.23 *Receive window in TCP*



There are two differences between the receive window in TCP and the one we used for SR in Chapter 13.

1. The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller or equal to the buffer size, as shown in the above figure. The receiver window size determines the number of bytes that the receive window can accept from the sender before being

overwhelmed (flow control). In other words, the receive window size, normally called *rwnd*, can be determined as:

rwnd = buffer size – number of waiting bytes to be pulled

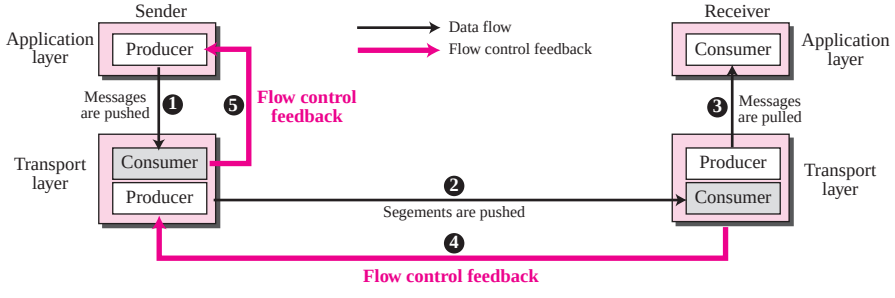
2. The second difference is the way acknowledgments are used in the TCP protocol. Remember that an acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN discussed in Chapter 13). The new versions of TCP, however, uses both cumulative and selective acknowledgements as we will discuss later in the option section.

15.7 FLOW CONTROL

As discussed in Chapter 13, *flow control* balances the rate a producer creates data with the rate a consumer can use the data. TCP separates flow control from error control. In this section we discuss flow control, ignoring error control. We temporarily assume that the logical channel between the sending and receiving TCP is error-free.

Figure 15.24 shows unidirectional data transfer between a sender and a receiver; bidirectional data transfer can be deduced from unidirectional one as discussed in Chapter 13.

Figure 15.24 *Data flow and flow control feedbacks in TCP*



The figure shows that data travel from the sending process down to the sending TCP, from the sending TCP to the receiving TCP, and from receiving TCP up to the receiving process (paths 1, 2, and 3). Flow control feedbacks, however, are traveling from the receiving TCP to the sending TCP and from the sending TCP up to the sending process (paths 4 and 5). Most implementations of TCP do not provide flow control feedback from the receiving process to the receiving TCP; they let the receiving process pull data from the receiving TCP whenever it is ready to do so. In other words, the receiving TCP controls the sending TCP; the sending TCP controls the sending process.

Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by sending TCP when its window is full. This means that our discussion of flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

Opening and Closing Windows

To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established. The receive window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves its right wal l to the right) when more bytes are pulled by the process. We assume that it does not shrink (the right wall does not move to the left).

The opening, closing, and shrinking of the send window is controlled by the receiver. The send window closes (moves its left wall to the right) when a new acknowledgement allows it to do so. The send window opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so. The send window shrinks on occasion. We assume that this situation does not occur.

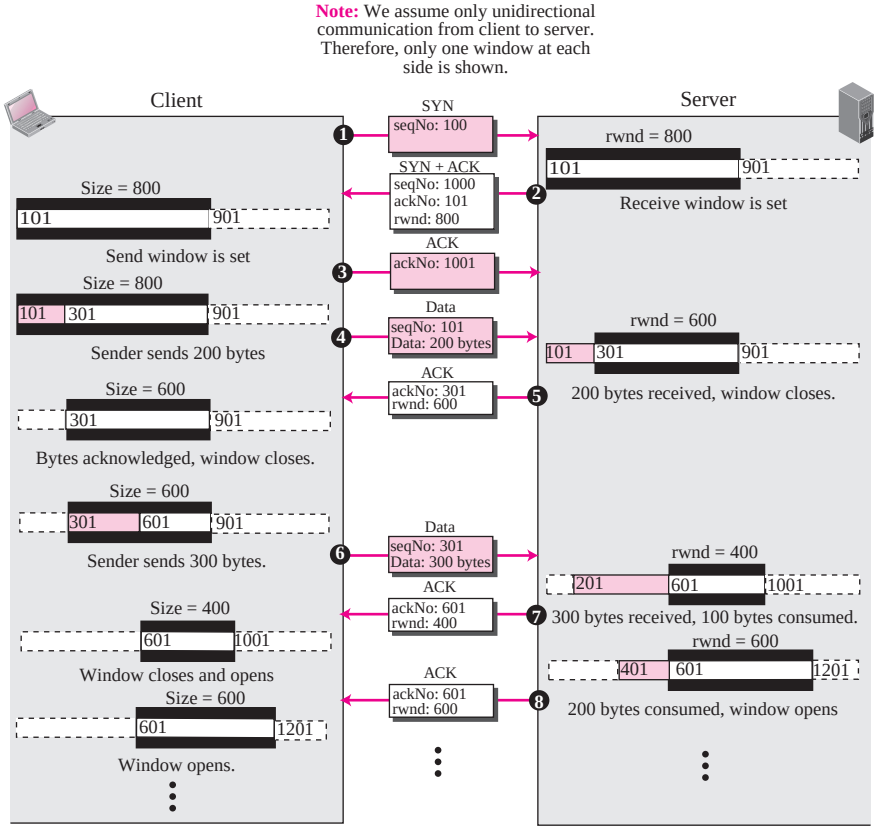
A Scenario

We show how the send and receive windows are set during the connection establishment phase, and how their situations will change during data transfer. Figure 15.25 shows a simple example of unidirectional data transfer (from client to server). For the time being, we ignore error control, assuming that no segment is corrupted, lost, duplicated, or arrived out of order. Note that we have shown only two windows for unidirectional data transfer.

Eight segments are exchanged between the client and server:

1. The first segment is from the client to the server (a SYN segment) to request connection. The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (rwnd = 800). Note that the number of the next byte to arrive is 101.
2. The second segment is from the server to the client. This is an ACK + SYN segment. The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.
3. The third segment is the ACK segment from the client to the server.
4. After the client has set its window with the size (800) dictated by the server, the process pushes 200 bytes of data. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. The segment shows the starting byte number as 101 and the segment carries 200 bytes. The window of the client is then adjusted to show 200 bytes of data are sent but waiting for acknowledgment. When this segment is received at the server, the bytes are stored, and the receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.

Figure 15.25 *An example of flow control*



5. The fifth segment is the feedback from the server to the client. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. The window size, however, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.
6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.
7. In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400. When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of rwnd = 400

advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.

8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new rwnd value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. We need to mention that the sending of this segment depends on the policy imposed by the implementation. Some implementations may not allow advertisement of the rwnd at this time; the server then needs to receive some data before doing so. After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

Shrinking of Windows

As we said before, the receive window cannot shrink. But the send window can shrink if the receiver defines a value for rwnd that results in shrinking the window. Some implementations do not allow the shrinking of the send window. The limitation does not allow the right wall of the send window to move to the left. In other words, the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new rwnd values to prevent the shrinking of the send window:

$$\textbf{new ackNo} + \textbf{new rwnd} \geq \textbf{last ackNo} + \textbf{last rwnd}$$

The left side of the inequality represents the new position of the right wall with respect to the sequence number space; the right side shows the old position of the right wall. The relationship shows that the right wall should not move to the left. The inequality is a mandate for the receiver to check its advertisment. However, note that the inequality is valid only if $S_f < S_n$; we need to remember that all calculations are in modulo 2^{32}.

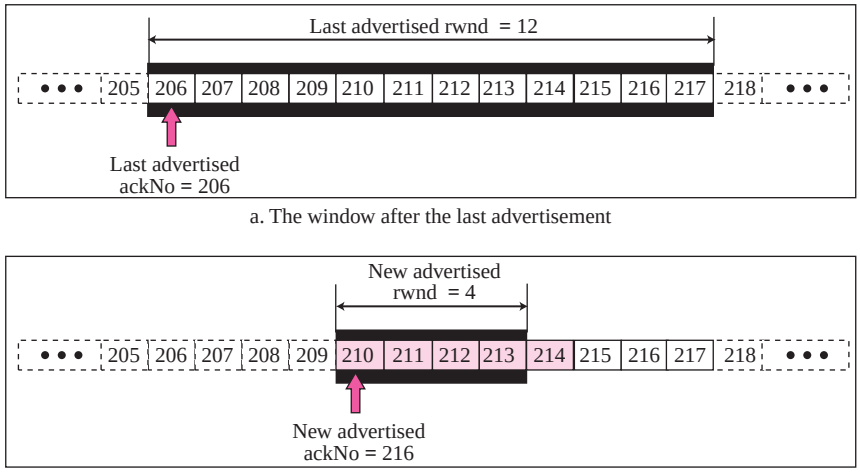
Example 15.2

Figure 15.26 shows the reason for this mandate by some implementations. Part a of the figure shows values of last acknowledgment and rwnd. Part b shows the situation in which the sender has sent bytes 206 to 214. Bytes 206 to 209 are acknowledged and purged. The new advertisment, however, defines the new value of rwnd as 4, in which $210 + 4 < 206 + 12$. When the send window shrinks, it creates a problem: byte 214 which has been already sent is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a because the receiver does not know which of the bytes 210 to 217 has already been sent. One way to prevent this situation is to let the receiver postpone its feedback until enough buffer locations are available in its window. In other words, the receiver should wait until more bytes are consumed by its process to meet the relationship described above.

Window Shutdown

We said that shrinking the send window by moving its right wall to the left is strongly discouraged. However, there is one exception: the receiver can temporarily shut down the window by sending a rwnd of 0. This can happen if for some reason the receiver does not want to receive any data from the sender for a while. In this case, the sender does not actually shrink the size of the window, but stops sending data until a new advertisement has arrived. As we will see later, even when the window is shut down by

Figure 15.26 *Example 15.2*



an order from the receiver, the sender can always send a segment with 1 byte of data. This is called probing and is used to prevent a deadlock (see the section on TCP timers).

Silly Window Syndrome

A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both. Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently. The inefficiency is even worse after accounting for the data link layer and physical layer overhead. This problem is called the **silly window syndrome.** For each site, we first describe how the problem is created and then give a proposed solution.

Syndrome Created by the Sender

The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time. The application program writes 1 byte at a time into the buffer of the sending TCP. If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data. The result is a lot of 41-byte segments that are traveling through an internet.

The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block. How long should the sending TCP wait? If it waits too long, it may delay the process. If it does not wait long enough, it may end up sending small segments. Nagle found an elegant solution.

Nagle's Algorithm **Nagle's algorithm** is simple:

1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.

The elegance of Nagle's algorithm is in its simplicity and in the fact that it takes into account the speed of the application program that creates the data and the speed of the network that transports the data. If the application program is faster than the network, the segments are larger (maximum-size segments). If the application program is slower than the network, the segments are smaller (less than the maximum segment size).

Syndrome Created by the Receiver

The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time. Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Also suppose that the input buffer of the receiving TCP is 4 kilobytes. The sender sends the first 4 kilobytes of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data. The receiving application reads the first byte of data from the input buffer of the receiving TCP. Now there is 1 byte of space in the incoming buffer. The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, takes this advertisement as good news and sends a segment carrying only 1 byte of data. The procedure will continue. One byte of data is consumed and a segment carrying 1 byte of data is sent. Again we have an efficiency problem and the silly window syndrome.

Two solutions have been proposed to prevent the silly window syndrome created by an application program that consumes data slower than they arrive.

Clark's Solution **Clark's solution** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.

Delayed Acknowledgment The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments. The delayed acknowledgment prevents the sending TCP from sliding its window. After the sending TCP has sent the data in the window, it stops. This kills the syndrome.

Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment. However, there also is a disadvantage in

that the delayed acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments.

TCP balances the advantages and disadvantages. It now defines that the acknowledgment should not be delayed by more than 500 ms.

15.8 ERROR CONTROL

TCP is a reliable transport layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.

TCP provides reliability using error control. Error control includes mechanisms for detecting and resending corrupted segments, resending lost segments, storing out-of-order segments until missing segments arrive, and detecting and discarding duplicated segments. Error control in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

Checksum

Each segment includes a checksum field, which is used to check for a corrupted segment. If a segment is corrupted as deleted by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment. We discussed how to calculate checksums earlier in the chapter.

Acknowledgment

TCP uses acknowledgments to confirm the receipt of data segments. Control segments that carry no data, but consume a sequence number, are also acknowledged. ACK segments are never acknowledged.

ACK segments do not consume sequence numbers and are not acknowledged.

Acknowledgment Type

In the past, TCP used only one type of acknowledgment: cumulative acknowledgment. Today, some TCP implementations also use selective acknowledgment.

Cumulative Acknowledgment (ACK) TCP was originally designed to acknowledge receipt of segments cumulatively. The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as positive cumulative acknowledgment or ACK. The word "positive" indicates that no feedback is provided for discarded, lost, or duplicate segments. The 32-bit ACK field in the TCP header is used for cumulative acknowledgments and its value is valid only when the ACK flag bit is set to 1.

Selective Acknowledgment (SACK) More and more implementations are adding another type of acknowledgment called **selective acknowledgment** or **SACK.** A