Chapter 6

Transport Layer II

Congestion is a situation in a network in which the load on the network, the number of packets sent to the network, is greater than the capacity of the network, the number of packets a network can handle. Congestion in a network may occur if the load on the network—the number of packets sent to the network—is greater than the capacity of the network—the number of packets a network can handle.

Congestion control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity. You may ask why there is congestion on a network. Congestion happens in any system that involves waiting. For example, congestion happens on a freeway because any abnormality in the flow, such as an accident during rush hour, creates blockage.

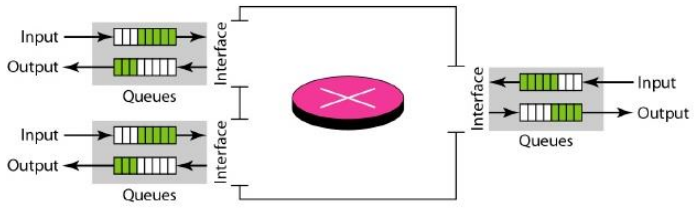


Figure 6.1 Buffering in the interfaces of a router

Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing. A router, for example, has an input queue and an output queue for each interface. When a packet arrives at the incoming interface, it undergoes three steps before departing, as shown in Fig. 6.1:

1. The packet is put at the end of the input queue while waiting to be checked.
2. The processing module of the router removes the packet from the input queue once it reaches the front of the queue and uses its routing table and the destination address to find the route.
3. The packet is put in the appropriate output queue and waits its turn to be sent.

We need to be aware of two issues. First, I the rate of packet arrival is higher than the packet processing rate, the input queues become longer and longer. Second, if the packet departure rate is less than the packet processing rate, the output queues become longer and longer.

6.1 Terminology

Congestion Window:

Each side of a TCP connection consists of a receive buffer, a send buffer, and several variables. The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the congestion window. The congestion window, denoted cwnd, imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd (receive window), that is:

```
LastByteSent – LastByteAcked _ min{cwnd, rwnd}
```

In order to focus on congestion control (as opposed to flow control), let us henceforth assume that the TCP receive buffer is so large that the receive-window constraint can be ignored; thus, the amount of unacknowledged data at the sender is solely limited by cwnd. The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate. To see this, consider a connection for which loss and packet transmission delays are negligible. Then, roughly, at the beginning of every round-trip time (RTT), the constraint permits the sender to send cwnd bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data. Pertinently, RTT is the time it takes for a small packet to travel from client to server and then back to the client. Thus the sender's send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

Let us define a "loss event" at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped. The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

- ***Timeout***

  Sender does not receive any acknowledgement within a predefined interval

- ***3 duplicate Acknowledgement (3 duplicate ACK)***

  When a receiver receives a segment, it sends an ACK to the sender. Thus, the sender receives an ACK for each successful transmission. Suppose that the receiver receives up to Nth segment, so it sends an ACK N+1. However, it receives N+2 segment next, NOT N+1. It resends the ACK N+1. Thus, the sender receives the same ACK N+1 twice and this is the first duplicate ACK. If the receiver receives any segment other than N+1 segment next time, it will resend the same ACK N+1. Now the sender is receiving the same ACK three times (i.e., 2nd duplicate ACK). If the receiver receives any other segment than N+1 again, it resends the previous ACK. Thus, the sender receives duplicate ACK again, which we call 3 duplicate ACK. The scenario is shown in Fig. 6.2.

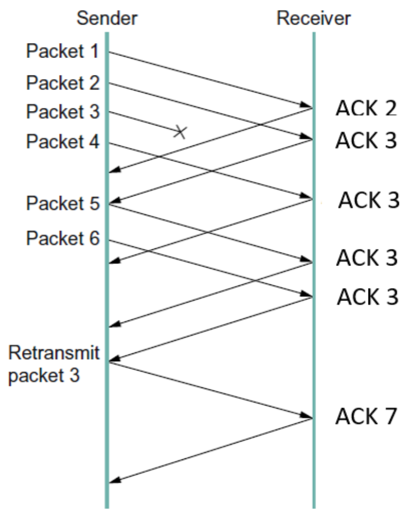


Figure 6.2 Illustration of 3-duplicate ACK

6.2 TCP Congestion Control Algorithm

There are many TCP congestion control algorithms such as

- TCP Tahoe
- TCP Reno
- TCP new Reno
- TCP Vegas
- TCP BBR

In this lecture, we will discuss TCP Reno in details.

The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery.

- **Slow start**
  When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS [RFC 3390], resulting in an initial sending rate of roughly MSS/RTT. For example, if MSS = 500 bytes and RTT = 200 msec, the resulting initial sending rate is only about 20 kbps. Since the available bandwidth to the TCP sender may be much larger than MSS/RTT, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the slow-start state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged. In the example of Figure 6.3, TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the

TCP send rate starts slow but grows exponentially during the slow start phase. The expansion of cwnd is shown in Fig. 6.3.

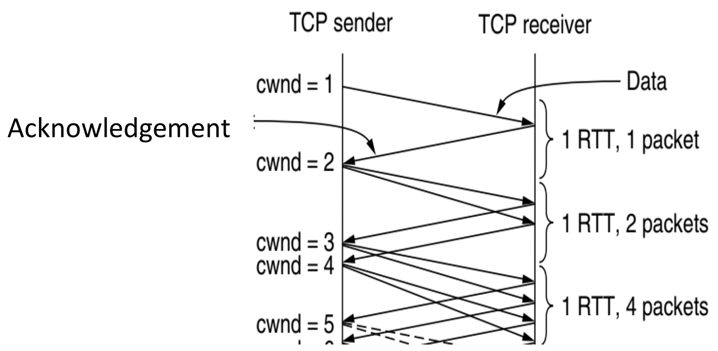


Figure 6.3 Slow start from an initial congestion window of one segment.

To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssth)**. Initially this value is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection. TCP keeps increasing the congestion window in slow start until a timeout occurs or the congestion window exceeds the threshold (or the receiver's window is filled).

- **Congestion avoidance**
  We have already observed that the congestion window increases exponentially in the slow start stge. A question may arise that when the growth will stop? Whenever the slow start threshold is crossed, TCP switches from slow start to congestion avoidance stage. In this mode, the congestion window is increased by one segment every round-trip time. Like slow start, this is usually implemented with an increase for every segment that is acknowledged, rather than an increase once per RTT. The procedure is illustrated in Fig. 6.4.

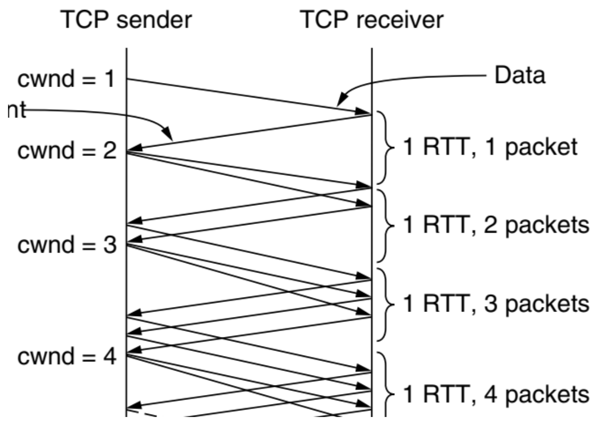


Figure 6.4 Linear increase of cwnd in congestion avoidance

But when should congestion avoidance's linear increase (of 1 MSS per RTT) end? The linear increase stops when one of the following two loss event occur:

Timeout

The value of cwnd is set to 1 MSS, and the value of ssthresh is updated to half the value of cwnd when the loss event occurred. The slow start state is then entered. In Fig. 6.5, we see that there is a slow start stage at which cwnd increases as 0 1 2 4 8 16. Since 16 is the threshold, the process enters the congestion avoidance stage after reaching cwnd size 16. Then, the cwnd is increased by 1 in each transmission round. However, when cwnd is 20, the timeout occurs (that is, no ACK is received within the pre-specified time). As is mentioned before, the *ssth* becomes half of its cwnd value. That is, *ssth* = 20/2 = 10. And the cwnd size is reduced to 1. After this change, the slow start stage is followed.

3-duplicate ACK

TCP sets the ssthresh to the half of the cwnd at which *3-duplicate ACK* occurs and cwnd to the ssthresh when the triple duplicate ACKs were received. Then congestion avoidance procedure is carried on. In Fig. 6.6, we see a 3-duplicate ACK occurrence at the 9th transmission round. Since event occurs at cwnd=20, the new threshold (ssth) becomes 20/2=10 and the cwnd at the transmission round 10 becomes 20/2 = 10. Then, the cwnd is continued to be increased linearly (1 MSS per RTT).

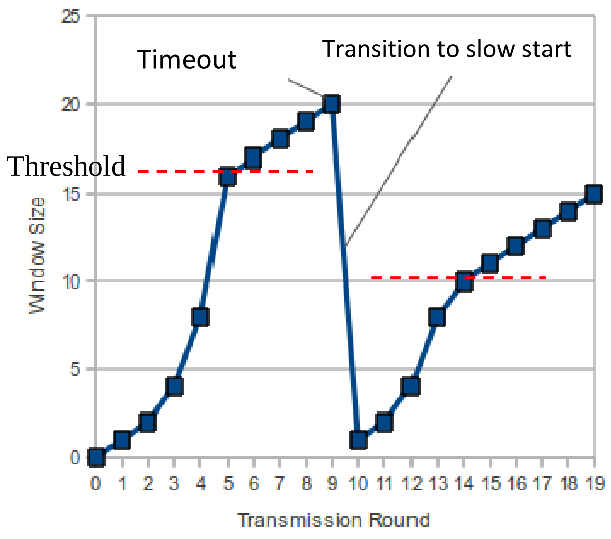


Figure 6.5 Timeout in congestion avoidance

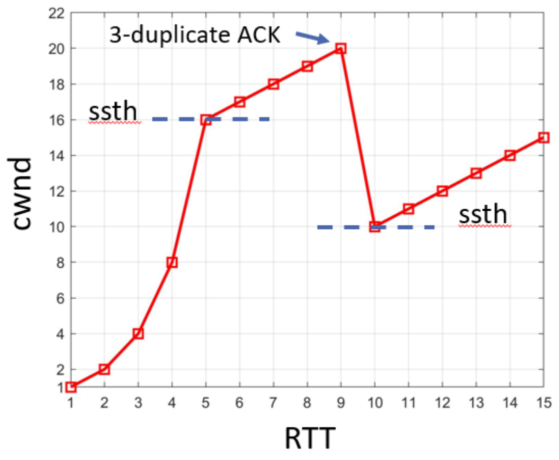


Figure 6.6 3-duplicate ACK in congestion avoidance

- Fast Recovery

  This is the stage at which TCP cwnd control procedure enters when 3-duplicate ACK occurs. The missing segment is resent. In this stage, cwnd increases linearly similar to the congestion avoidance stage.

Another Example

In Fig. 6.7, we see cwnd is initially 1 MSS. In RTT 1, cwnd=2. In RTT 2, cwnd increases to 4. In RTT 3, cwnd becomes 8; then it increases to 16 in the next RTT. In RTT 5, cwnd reaches the slow start threshold (ssth) of 32. Then TCP stops the exponential increase of cwnd and starts linear increase (cwnd increases by 1 per RTT). At RTT 14, we observe a transition from the congestion avoidance stage. Now a question is that which loss event has occurred at this RTT: timeout/3-duplicate ACK? To get this answer, let’s observe cwnd at the next RTT. It is 20 (i.e., 40/2=20). Since cwnd is half of the previous cwnd, 3-duplicate ACK occurred at RTT 14. At this moment (at RTT 16), the congestion avoidance will start as the cwnd is equal to the threshold.

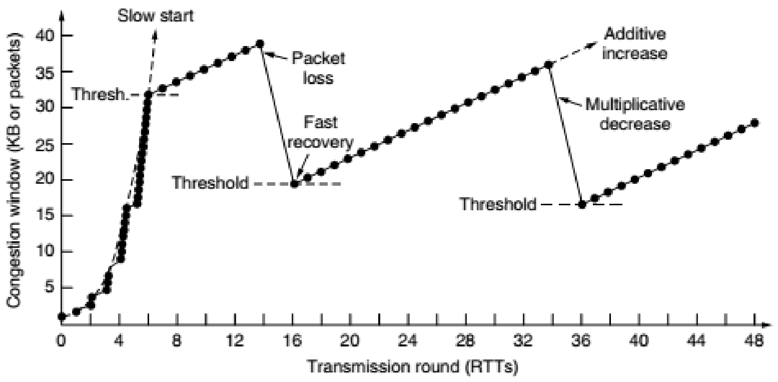


Figure 6.7 TCP reno example

Exercise:

Identify Timeout/3 duplicate ACK incidence. Also identify slow start, congestion avoidance period.

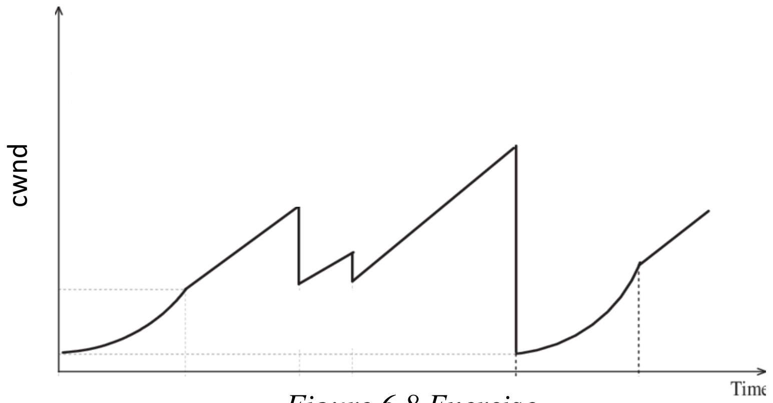


Figure 6.8 Exercise

6.3 Transport Layer Flow Control

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.

Figure 6.9 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 and closes. Figure 10 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 close.

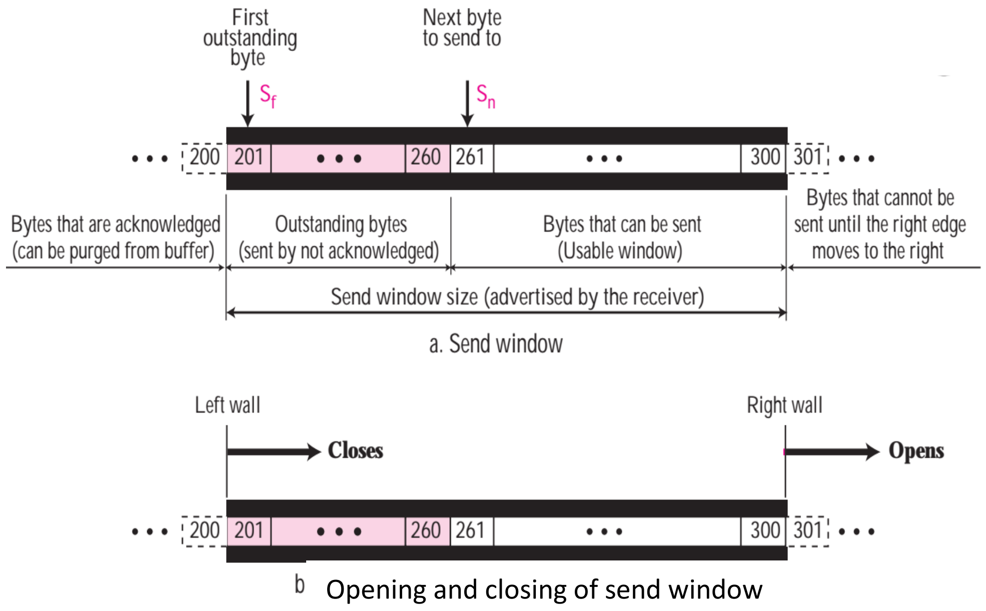
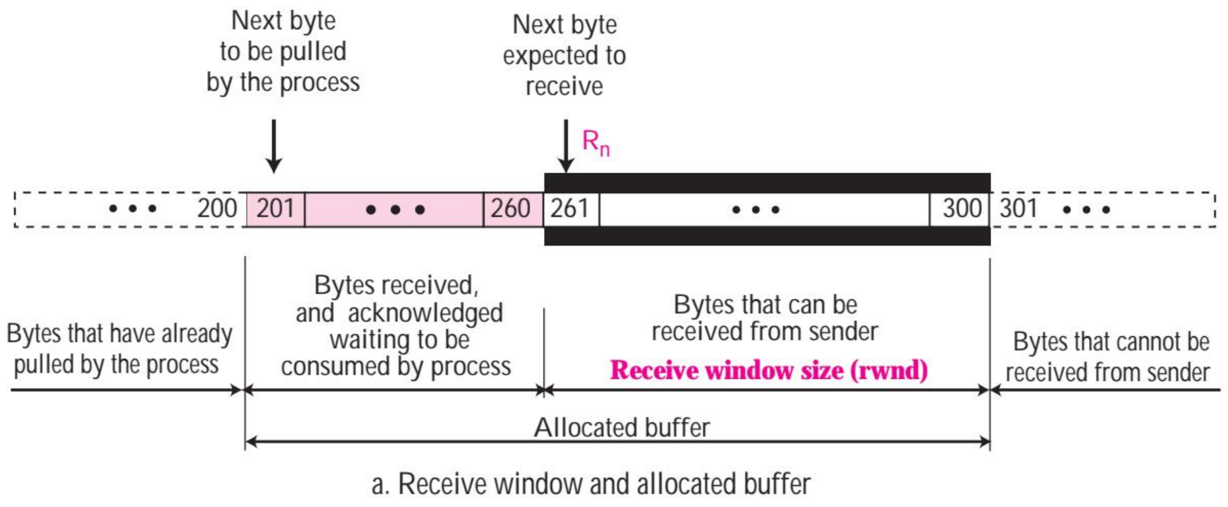

First outstanding byte
Next byte to send to
S_f
S_n
200
201
260
261
300
301
Bytes that are acknowledged (can be purged from buffer)
Outstanding bytes (sent by not acknowledged)
Bytes that can be sent (Usable window)
Bytes that cannot be sent until the right edge moves to the right
Send window size (advertised by the receiver)
a. Send window
Left wall
Right wall
Closes
Opens
b Opening and closing of send window

Figure 6.9 Send window in TCP


Next byte to be pulled by the process
Next byte expected to receive
R_n
200
201
260
261
300
301
Bytes that have already pulled by the process
Bytes received, and acknowledged waiting to be consumed by process
Bytes that can be received from sender
Receive window size (rwnd)
Bytes that cannot be received from sender
Allocated buffer
a. Receive window and allocated buffer

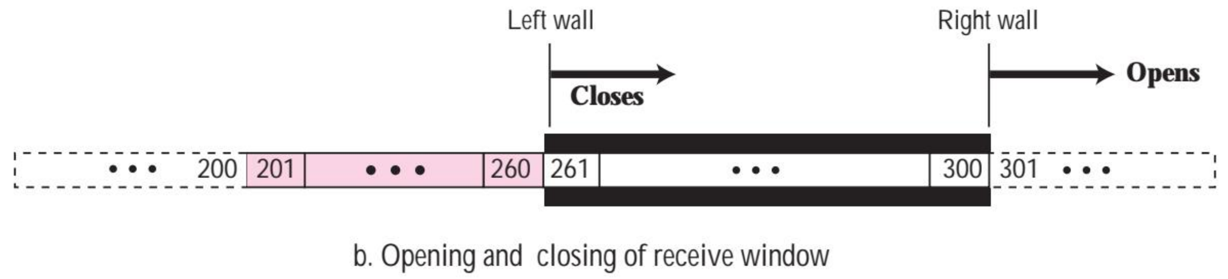


Figure 6.10 Receive window in TCP

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. We assume that the logical channel between the sending and receiving TCP is error-free. 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 6.11 shows a simple example of unidirectional data transfer (from client to server. 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.

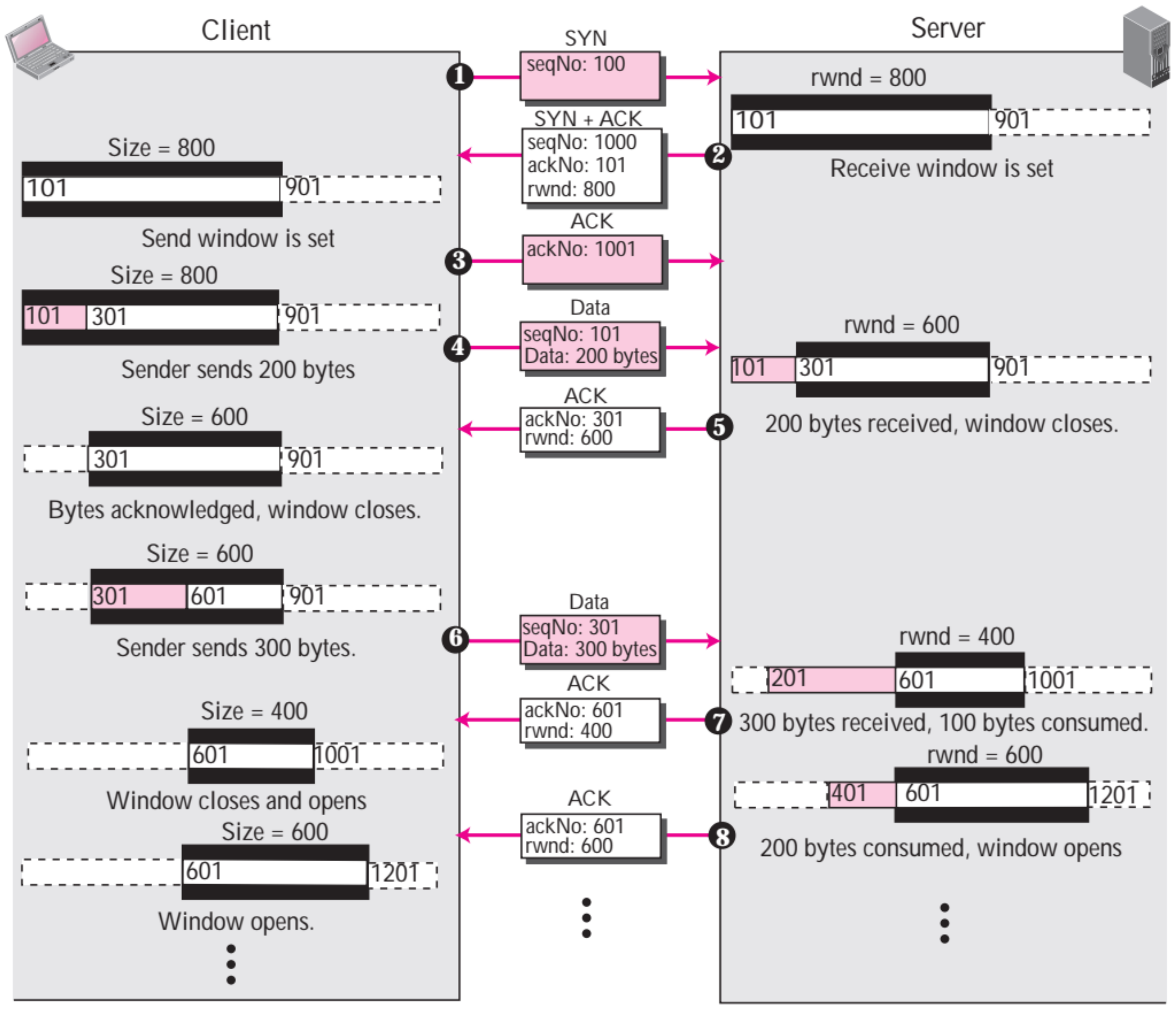


Figure 6.11 An example of flow control

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.