Triggering Transmission

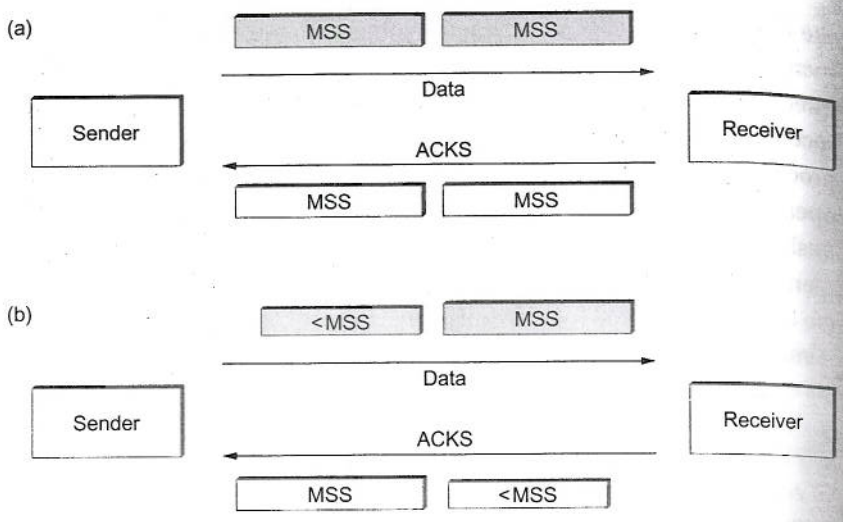
We next consider a surprisingly subtle issue: how TCP decides to transmit a segment. As described earlier, TCP supports a byte-stream abstraction; that is, application programs write bytes into the stream, and it is up to TCP to decide that it has enough bytes to send a segment. What factors govern this decision?

If we ignore the possibility of flow control—that is, we assume the window is wide open, as would be the case when a connection first starts—then TCP has three mechanisms to trigger the transmission of a segment. First, TCP maintains a variable, typically called the *maximum segment size* (MSS), and it sends a segment as soon as it has collected MSS bytes from the sending process. MSS is usually set to the size of the largest segment TCP can send without causing the local IP to fragment. That is, MSS is set to the maximum transmission unit (MTU) of the directly connected network, minus the size of the TCP and IP headers. The second thing that triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes. The final trigger for transmitting a segment is that a timer fires; the resulting segment contains as many bytes as are currently buffered for transmission. However, as we will soon see, this "timer" isn't exactly what you expect.

Silly Window Syndrome

Of course, we can't just ignore flow control, which plays an obvious role in throttling the sender. If the sender has MSS bytes of data to send and the window is open at least that much, then the sender transmits a full segment. Suppose, however, that the sender is accumulating bytes to send, but the window is currently closed. Now suppose an ACK arrives that effectively opens the window enough for the sender to transmit, say, MSS/2 bytes. Should the sender transmit a half-full segment or wait for the window to open to a full MSS? The original specification was silent on this point, and early implementations of TCP decided to go ahead and transmit a half-full segment. After all, there is no telling how long it will be before the window opens further.

It turns out that the strategy of aggressively taking advantage of any available window leads to a situation now known as the *silly window syndrome*. Figure 5.9 helps visualize what happens. If you think of a TCP stream as a conveyer belt with "full" containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then MSS-sized segments correspond to large containers and 1-byte segments correspond to very small containers. As long as the sender is sending MSS-sized segments and the receiver ACKs at least one MSS of data at a time, everything is good (Figure 5.9(a)). But, what if the receiver has to reduce the window, so that at some time the sender can't send a full MSS of data? If the sender aggressively fills a smaller-than-MSS empty container as soon as it arrives, then the receiver will ACK that smaller number of bytes, and hence the small container introduced into the system remains in the system indefinitely. That is, it is immediately filled and emptied at each end and is never coalesced with adjacent containers to



■ **FIGURE 5.9** Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small "container" enters the system and continues to circulate.

create larger containers, as in Figure 5.9(b). This scenario was discovered when early implementations of TCP regularly found themselves filling the network with tiny segments.

Note that the silly window syndrome is only a problem when either the sender transmits a small segment or the receiver opens the window a small amount. If neither of these happens, then the small container is never introduced into the stream. It's not possible to outlaw sending small segments; for example, the application might do a *push* after sending a single byte. It is possible, however, to keep the receiver from introducing a small container (i.e., a small open window). The rule is that after advertising a zero window the receiver must wait for space equal to an MSS before it advertises an open window.

Since we can't eliminate the possibility of a small container being introduced into the stream, we also need mechanisms to coalesce them. The receiver can do this by delaying ACKs—sending one combined ACK rather than multiple smaller ones—but this is only a partial solution because the receiver has no way of knowing how long it is safe to delay waiting either for another segment to arrive or for the application to read more data (thus opening the window). The ultimate solution falls to the

sender, which brings us back to our original issue: When does the TCP sender decide to transmit a segment?

Nagle's Algorithm

Returning to the TCP sender, if there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data, but the question is how long? If we wait too long, then we hurt interactive applications like Telnet. If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the silly window syndrome. The answer is to introduce a timer and to transmit when the timer expires.

While we could use a clock-based timer—for example, one that fires every 100 ms—Nagle introduced an elegant *self-clocking* solution. The idea is that as long as TCP has any data in flight, the sender will eventually receive an ACK. This ACK can be treated like a timer firing, triggering the transmission of more data. Nagle's algorithm provides a simple, unified rule for deciding when to transmit:

```
When the application produces data to send
  if both the available data and the window ≥ MSS
    send a full segment
  else
    if there is unACKed data in flight
      buffer the new data until an ACK arrives
    else
      send all the new data now
```

In other words, it's always OK to send a full segment if the window allows. It's also all right to immediately send a small amount of data if there are currently no segments in transit, but if there is anything in flight the sender must wait for an ACK before transmitting the next segment. Thus, an interactive application like Telnet that continually writes one byte at a time will send data at a rate of one segment per RTT. Some segments will contain a single byte, while others will contain as many bytes as the user was able to type in one round-trip time. Because some applications cannot afford such a delay for each write it does to a TCP connection, the socket interface allows the application to turn off Nagel's algorithm by setting the TCP_NODELAY option. Setting this option means that data is transmitted as soon as possible.