2.5 Reliable Transmission

As we saw in the previous section, frames are sometimes corrupted while in transit, with an error code like CRC used to detect such errors. While some error codes are strong enough also to correct errors, in practice the overhead is typically too large to handle the range of bit and burst errors that can be introduced on a network link. Even when error-correcting codes are used (e.g., on wireless links) some errors will be too severe to be corrected. As a result, some corrupt frames must be discarded. A link-level protocol that wants to deliver frames reliably must somehow recover from these discarded (lost) frames.

It's worth noting that reliability is a function that *may* be provided at the link level, but many modern link technologies omit this function. Furthermore, reliable delivery is frequently provided at higher levels, including both transport and sometimes, the application layer. Exactly where it should be provided is a matter of some debate and depends on many factors. We describe the basics of reliable delivery here, since the principles are common across layers, but you should be aware that we're not just talking about a link-layer function.

Reliable delivery is usually accomplished using a combination of two fundamental mechanisms—*acknowledgments* and *timeouts*. An acknowledgment (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received an earlier frame. By control frame we mean a header without any data, although a protocol can *piggyback* an ACK on a data frame it just happens to be sending in the opposite direction. The receipt of an acknowledgment indicates to the sender of the original frame that its frame was successfully delivered. If the sender does not receive an acknowledgment after a reasonable amount of time, then it *retransmits* the original frame. This action of waiting a reasonable amount of time is called a *timeout*.

The general strategy of using acknowledgments and timeouts to implement reliable delivery is sometimes called *automatic repeat request* (abbreviated ARQ). This section describes three different ARQ algorithms using generic language; that is, we do not give detailed information about a particular protocol's header fields.

Stop-and-Wait

The simplest ARQ scheme is the *stop-and-wait* algorithm. The idea of stop-and-wait is straightforward: After transmitting one frame, the sender waits for an acknowledgment before transmitting the next frame. If the acknowledgment does not arrive after a certain period of time, the sender times out and retransmits the original frame.

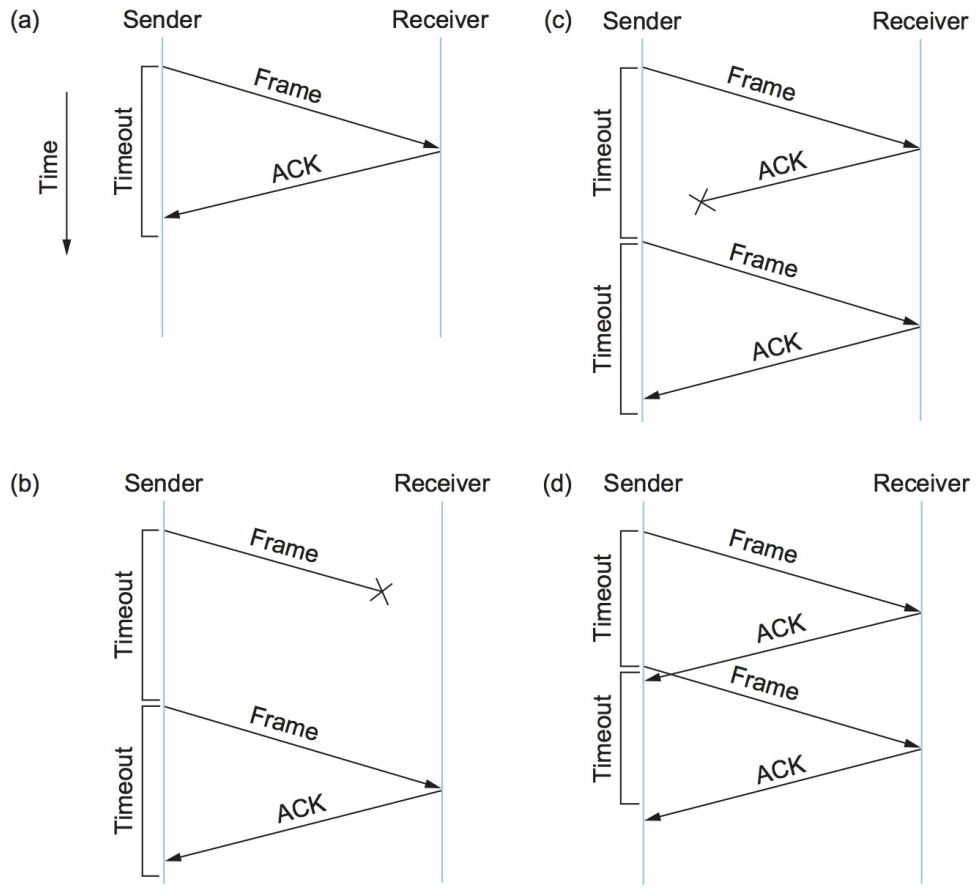

Figure 1. Timeline showing four different scenarios for the stop-and-wait algorithm. (a) The ACK is received before the timer expires; (b) the original frame is lost; (c) the ACK is lost; (d) the timeout fires too soon.

Figure 1 illustrates timeslines for four different scenarios that result from this basic algorithm. The sending side is represented on the left, the receiving side is depicted on the right, and time flows from top to bottom. Figure 1(a) shows the situation in which the ACK is received before the timer expires; (b) and (c) show the situation in which the original frame and the ACK, respectively, are lost; and (d) shows the situation in which the timeout fires too soon. Recall that by "lost" we mean that the frame was corrupted while in transit, that this corruption was detected by an error code on the receiver, and that the frame was subsequently discarded.

The packet timelines shown in this section are examples of a frequently used tool in teaching, explaining, and designing protocols. They are useful because they capture visually the behavior over time of a distributed system—something that can be quite hard to analyze. When designing a protocol, you often have to be prepared for the unexpected—a system crashes, a message gets lost, or something that you expected to happen quickly turns out to take a long time. These sorts of diagrams can often help us understand what might go wrong in such cases and thus help a protocol designer be prepared for every eventuality.

There is one important subtlety in the stop-and-wait algorithm. Suppose the sender sends a frame and the receiver acknowledges it, but the acknowledgment is either lost or delayed in arriving. This situation is illustrated in timelines (c) and (d) of Figure 1. In both cases, the sender times out and retransmits the original frame, but the receiver will think that it is the next frame, since it correctly received and acknowledged the first frame. This has the potential to cause duplicate copies of a frame to be delivered. To address this problem, the header for a stop-and-wait protocol usually includes a 1-bit sequence number—that is, the sequence number can take on the values 0 and 1—and the sequence numbers used for each frame alternate, as illustrated in Figure 2. Thus, when the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first ACK was lost).

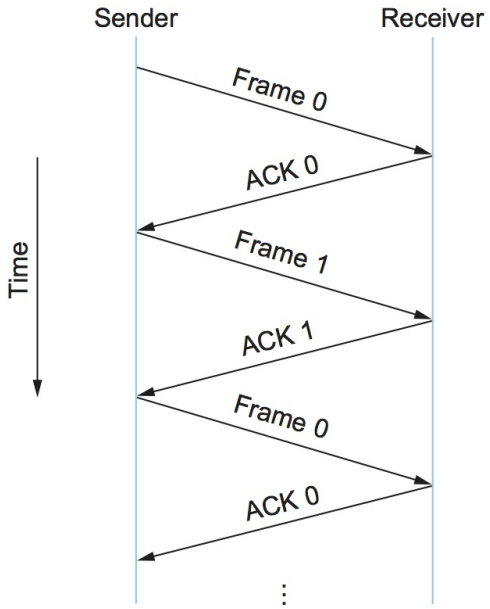

Figure 2. Timeline for stop-and-wait with 1-bit sequence number.

The main shortcoming of the stop-and-wait algorithm is that it allows the sender to have only one outstanding frame on the link at a time, and this may be far below the link's capacity. Consider, for example, a 1.5-Mbps link with a 45-ms round-trip time. This link has a delay × bandwidth product of 67.5 Kb, or approximately 8 KB. Since the sender can send only one frame per RTT, and assuming a frame size of 1 KB, this implies a maximum sending rate of

Bits-Per-Frame / Time-Per-Frame = 1024 x 8 / 0.045 = 182 kbps

or about one-eighth of the link's capacity. To use the link fully, then, we'd like the sender to be able to transmit up to eight frames before having to wait for an acknowledgment.

The significance of the delay × bandwidth product is that it represents the amount of data that could be in transit. We would like to be able to send this much data without waiting for the first acknowledgment. The principle at work here is often referred to as *keeping the pipe full*. The algorithms presented in the following two subsections do exactly this.

Sliding Window

Consider again the scenario in which the link has a delay × bandwidth product of 8 KB and frames are 1 KB in size. We would like the sender to be ready to transmit the ninth frame at pretty much the same moment that the ACK for the first frame arrives. The algorithm that allows us to do this is called *sliding window*, and an illustrative timeline is given in Figure 3.

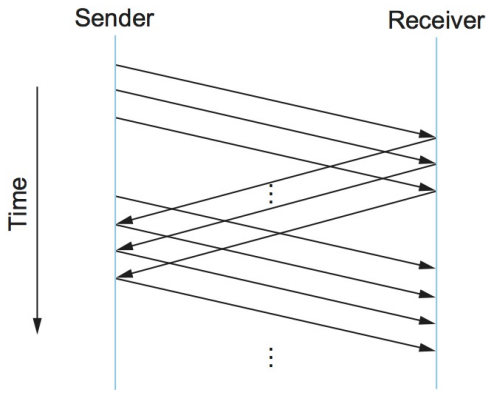

Figure 3. Timeline for the sliding window algorithm.

The Sliding Window Algorithm

The sliding window algorithm works as follows. First, the sender assigns a *sequence number*, denoted `SeqNum`, to each frame. For now, let's ignore the fact that `SeqNum` is implemented by a finite-size header field and instead assume that it can grow infinitely large. The sender maintains three variables: The *send window size*, denoted `SWS`, gives the upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit; `LAR` denotes the sequence number of the *last acknowledgment received*; and `LFS` denotes the sequence number of the *last frame sent*. The sender also maintains the following invariant:

```
LFS - LAR <= SWS
```

This situation is illustrated in Figure 4.

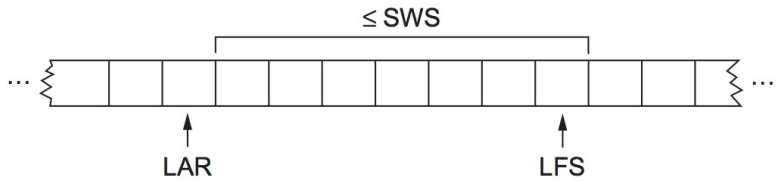

Figure 4. Sliding window on sender.

When an acknowledgment arrives, the sender moves `LAR` to the right, thereby allowing the sender to transmit another frame. Also, the sender associates a timer with each frame it transmits, and it retransmits the frame should the timer expire before an ACK is received. Notice that the sender has to be willing to buffer up to `SWS` frames since it must be prepared to retransmit them until they are acknowledged.

The receiver maintains the following three variables: The *receive window size*, denoted `RWS`, gives the upper bound on the number of out-of-order frames that the receiver is willing to accept; `LAF` denotes the sequence number of the *largest acceptable frame*; and `LFR` denotes the sequence number of the *last frame received*. The receiver also maintains the following invariant:

```
LAF - LFR <= RWS
```

This situation is illustrated in Figure 5.

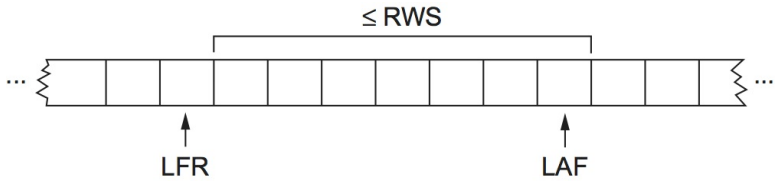

Figure 5. Sliding window on receiver.

When a frame with sequence number `SeqNum` arrives, the receiver takes the following action. If `SeqNum <= LFR` or `SeqNum > LAF`, then the frame is outside the receiver's window and it is discarded. If `LFR < SeqNum <= LAF`, then the frame is within the receiver's window and it is accepted. Now the receiver needs to decide whether or not to send an ACK. Let `SeqNumToAck` denote the largest sequence number not yet acknowledged, such that all frames with sequence numbers less than or equal to `SeqNumToAck` have been received. The receiver acknowledges the receipt of `SeqNumToAck`, even if higher numbered packets have been received. This acknowledgment is said to be cumulative. It then sets `LFR = SeqNumToAck` and adjusts `LAF = LFR + RWS`.

For example, suppose `LFR = 5` (i.e., the last ACK the receiver sent was for sequence number 5), and `RWS = 4`. This implies that `LAF = 9`. Should frames 7 and 8 arrive, they will be buffered because they are within the receiver's window. However, no ACK needs to be sent since frame 6 has yet to arrive. Frames 7 and 8 are said to have arrived out of order. (Technically, the receiver could resend an ACK for frame 5 when frames 7 and 8 arrive.) Should frame 6

then arrive—perhaps it is late because it was lost the first time and had to be retransmitted, or perhaps it was simply delayed—the receiver acknowledges frame 8, bumps `LFR` to 8, and sets `LAF` to 12. If frame 6 was in fact lost, then a timeout will have occurred at the sender, causing it to retransmit frame 6.

> It's unlikely that a packet could be delayed on a point-to-point link, this same algorithm is used on multi-hop connections where such delays are possible.

We observe that when a timeout occurs, the amount of data in transit decreases, since the sender is unable to advance its window until frame 6 is acknowledged. This means that when packet losses occur, this scheme is no longer keeping the pipe full. The longer it takes to notice that a packet loss has occurred, the more severe this problem becomes.

Notice that, in this example, the receiver could have sent a *negative acknowledgment* (NAK) for frame 6 as soon as frame 7 arrived. However, this is unnecessary since the sender's timeout mechanism is sufficient to catch this situation, and sending NAKs adds additional complexity to the receiver. Also, as we mentioned, it would have been legitimate to send additional acknowledgments of frame 5 when frames 7 and 8 arrived; in some cases, a sender can use duplicate ACKs as a clue that a frame was lost. Both approaches help to improve performance by allowing early detection of packet losses.

Yet another variation on this scheme would be to use *selective acknowledgments*. That is, the receiver could acknowledge exactly those frames it has received rather than just the highest numbered frame received in order. So, in the above example, the receiver could acknowledge the receipt of frames 7 and 8. Giving more information to the sender makes it potentially easier for the sender to keep the pipe full but adds complexity to the implementation.

The sending window size is selected according to how many frames we want to have outstanding on the link at a given time; `SWS` is easy to compute for a given delay $\times$ bandwidth product. On the other hand, the receiver can set `RWS` to whatever it wants. Two common settings are `RWS = 1`, which implies that the receiver will not buffer any frames that arrive out of order, and `RWS = SWS`, which implies that the receiver can buffer any of the frames the sender transmits. It makes no sense to set `RWS > SWS` since it's impossible for more than `SWS` frames to arrive out of order.

Finite Sequence Numbers and Sliding Window

We now return to the one simplification we introduced into the algorithm—our assumption that sequence numbers can grow infinitely large. In practice, of course, a frame's sequence number is specified in a header field of some finite size. For example, a 3-bit field means that there are eight possible sequence numbers, 0..7. This makes it necessary to reuse sequence numbers or, stated another way, sequence numbers wrap around. This introduces the problem of being able to distinguish between different incarnations of the same sequence numbers, which implies that the number of possible sequence numbers must be larger than the number of outstanding frames allowed. For example, stop-and-wait allowed one outstanding frame at a time and had two distinct sequence numbers.

Suppose we have one more number in our space of sequence numbers than we have potentially outstanding frames; that is, `SWS <= MaxSeqNum - 1`, where `MaxSeqNum` is the number of available sequence numbers. Is this sufficient? The answer depends on `RWS`. If `RWS = 1`, then `MaxSeqNum >= SWS + 1` is sufficient. If `RWS` is equal to `SWS`, then having a `MaxSeqNum` just one greater than the sending window size is not good enough. To see this, consider the situation in which we have the eight sequence numbers 0 through 7, and `SWS = RWS = 7`. Suppose the sender transmits frames 0..6, they are successfully received, but the ACKs are lost. The receiver is now expecting frames 7, 0..5, but the sender times out and sends frames 0..6. Unfortunately, the receiver is expecting the second incarnation of frames 0..5 but gets the first incarnation of these frames. This is exactly the situation we wanted to avoid.

It turns out that the sending window size can be no more than half as big as the number of available sequence numbers when `RWS = SWS`, or stated more precisely,

```
SWS < (MaxSeqNum + 1)/ 2
```

Intuitively, what this is saying is that the sliding window protocol alternates between the two halves of the sequence number space, just as stop-and-wait alternates between sequence numbers 0 and 1. The only difference is that it continually slides between the two halves rather than discretely alternating between them.

Note that this rule is specific to the situation where `RWS = SWS`. We leave it as an exercise to determine the more general rule that works for arbitrary values of `RWS` and `SWS`. Also note that the relationship between the window size and the sequence number space depends on an assumption that is so obvious that it is easy to overlook, namely that frames are not reordered in transit. This cannot happen on a direct point-to-point link since there is no way for one frame to overtake another during transmission. However, we will see the sliding window algorithm used in a different environments, and we will need to devise another rule.

Implementation of Sliding Window

The following routines illustrate how we might implement the sending and receiving sides of the sliding window algorithm. The routines are taken from a working protocol named, appropriately enough, Sliding Window Protocol (SWP). So as not to concern ourselves with the adjacent protocols in the protocol graph, we denote the protocol sitting above SWP as the high-level protocol (HLP) and the protocol sitting below SWP as the link-level protocol (LLP).

We start by defining a pair of data structures. First, the frame header is very simple: It contains a sequence number (`SeqNum`) and an acknowledgment number (`AckNum`). It also contains a `Flags` field that indicates whether the frame is an ACK or carries data.

```
typedef u_char SwpSeqno;

typedef struct {
    SwpSeqno   SeqNum;   /* sequence number of this frame */
    SwpSeqno   AckNum;   /* ack of received frame */
    u_char     Flags;          /* up to 8 bits worth of flags */
} SwpHdr;
```

Next, the state of the sliding window algorithm has the following structure. For the sending side of the protocol, this state includes variables `LAR` and `LFS`, as described earlier in this section, as well as a queue that holds frames that have been transmitted but not yet acknowledged (`sendQ`). The sending state also includes a *counting semaphore* called `sendWindowNotFull`. We will see how this is used below, but generally a semaphore is a synchronization primitive that supports `semWait` and `semSignal` operations. Every invocation of `semSignal` increments the semaphore by 1, and every invocation of `semWait` decrements `s` by 1, with the calling process blocked (suspended) should decrementing the semaphore cause its value to become less than 0. A process that is blocked during its call to `semWait` will be allowed to resume as soon as enough `semSignal` operations have been performed to raise the value of the semaphore above 0.

For the receiving side of the protocol, the state includes the variable `NFE`. This is the *next frame expected*, the frame with a sequence number one more that the last frame received (LFR), described earlier in this section. There is also a queue that holds frames that have been received out of order (`recvQ`). Finally, although not shown, the sender and receiver sliding window sizes are defined by constants `SWS` and `RWS`, respectively.

```
typedef struct {
    /* sender side state: */
    SwpSeqno    LAR;        /* seqno of last ACK received */
    SwpSeqno    LFS;        /* last frame sent */
    Semaphore   sendWindowNotFull;
    SwpHdr      hdr;        /* pre-initialized header */
    struct sendQ_slot {
        Event   timeout;    /* event associated with send-timeout */
        Msg     msg;
    }   sendQ[SWS];

    /* receiver side state: */
```

```
    SwpSeqno    NFE;       /* seqno of next frame expected */
    struct recvQ_slot {
        int     received;  /* is msg valid? */
        Msg     msg;
    }   recvQ[RWS];
} SwpState;
```

The sending side of SWP is implemented by procedure `sendSWP`. This routine is rather simple. First, `semWait` causes this process to block on a semaphore until it is OK to send another frame. Once allowed to proceed, `sendSWP` sets the sequence number in the frame's header, saves a copy of the frame in the transmit queue (`sendQ`), schedules a timeout event to handle the case in which the frame is not acknowledged, and sends the frame to the next-lower-level protocol, which we denote as `LINK`.

One detail worth noting is the call to `store_swp_hdr` just before the call to `msgAddHdr`. This routine translates the C structure that holds the SWP header (`state->hdr`) into a byte string that can be safely attached to the front of the message (`hbuf`). This routine (not shown) must translate each integer field in the header into network byte order and remove any padding that the compiler has added to the C structure. The issue of byte order is a non-trivial issue, but for now it is enough to assume that this routine places the most significant bit of a multiword integer in the byte with the highest address.

Another piece of complexity in this routine is the use of `semWait` and the `sendWindowNotFull` semaphore. `sendWindowNotFull` is initialized to the size of the sender's sliding window, `SWS` (this initialization is not shown). Each time the sender transmits a frame, the `semWait` operation decrements this count and blocks the sender should the count go to 0. Each time an ACK is received, the `semSignal` operation invoked in `deliverSWP` (see below) increments this count, thus unblocking any waiting sender.

```
static int
sendSWP(SwpState *state, Msg *frame)
{
    struct sendQ_slot *slot;
    hbuf[HLEN];

    /* wait for send window to open */
    semWait(&state->sendWindowNotFull);
    state->hdr.SeqNum = ++state->LFS;
    slot = &state->sendQ[state->hdr.SeqNum % SWS];
    store_swp_hdr(state->hdr, hbuf);
    msgAddHdr(frame, hbuf, HLEN);
    msgSaveCopy(&slot->msg, frame);
    slot->timeout = evSchedule(swpTimeout, slot, SWP_SEND_TIMEOUT);
    return send(LINK, frame);
}
```

Before continuing to the receive side of SWP, we need to reconcile a seeming inconsistency. On the one hand, we have been saying that a high-level protocol invokes the services of a low-level protocol by calling the `send` operation, so we would expect that a protocol that wants to send a message via SWP would call `send(SWP, packet)`. On the other hand, the procedure that implements SWP's send operation is called `sendSWP`, and its first argument is a state variable (`SwpState`). What gives? The answer is that the operating system provides glue code that translates the generic call to `send` into a protocol-specific call to `sendSWP`. This glue code maps the first argument to `send` (the magic protocol variable `SWP`) into both a function pointer to `sendSWP` and a pointer to the protocol state that SWP needs to do its job. The reason we have the high-level protocol indirectly invoke the protocol-specific function through the generic function call is that we want to limit how much information the high-level protocol has coded in it about the low-level protocol. This makes it easier to change the protocol graph configuration at some time in the future.

Now we move on to SWP's protocol-specific implementation of the `deliver` operation, which is given in procedure `deliverSWP`. This routine actually handles two different kinds of incoming messages: ACKs for frames sent earlier from this node and data frames arriving at this node. In a sense, the ACK half of this routine is the counterpart to the

sender side of the algorithm given in `sendSWP`. A decision as to whether the incoming message is an ACK or a data frame is made by checking the `Flags` field in the header. Note that this particular implementation does not support piggybacking ACKs on data frames.

When the incoming frame is an ACK, `deliverSWP` simply finds the slot in the transmit queue (`sendQ`) that corresponds to the ACK, cancels the timeout event, and frees the frame saved in that slot. This work is actually done in a loop since the ACK may be cumulative. The only other thing to notice about this case is the call to subroutine `swpInWindow`. This subroutine, which is given below, ensures that the sequence number for the frame being acknowledged is within the range of ACKs that the sender currently expects to receive.

When the incoming frame contains data, `deliverSWP` first calls `msgStripHdr` and `load_swp_hdr` to extract the header from the frame. Routine `load_swp_hdr` is the counterpart to `store_swp_hdr` discussed earlier; it translates a byte string into the C data structure that holds the SWP header. `deliverSWP` then calls `swpInWindow` to make sure the sequence number of the frame is within the range of sequence numbers that it expects. If it is, the routine loops over the set of consecutive frames it has received and passes them up to the higher-level protocol by invoking the `deliverHLP` routine. It also sends a cumulative ACK back to the sender, but does so by looping over the receive queue (it does not use the `SeqNumToAck` variable used in the prose description given earlier in this section).

```
static int
deliverSWP(SwpState state, Msg *frame)
{
    SwpHdr   hdr;
    char     *hbuf;

    hbuf = msgStripHdr(frame, HLEN);
    load_swp_hdr(&hdr, hbuf)
    if (hdr->Flags & FLAG_ACK_VALID)
    {
        /* received an acknowledgment—do SENDER side */
        if (swpInWindow(hdr.AckNum, state->LAR + 1, state->LFS))
        {
            do
            {
                struct sendQ_slot *slot;

                slot = &state->sendQ[++state->LAR % SWS];
                evCancel(slot->timeout);
                msgDestroy(&slot->msg);
                semSignal(&state->sendWindowNotFull);
            } while (state->LAR != hdr.AckNum);
        }
    }

    if (hdr.Flags & FLAG_HAS_DATA)
    {
        struct recvQ_slot *slot;

        /* received data packet—do RECEIVER side */
        slot = &state->recvQ[hdr.SeqNum % RWS];
        if (!swpInWindow(hdr.SeqNum, state->NFE, state->NFE + RWS - 1))
        {
            /* drop the message */
            return SUCCESS;
        }
        msgSaveCopy(&slot->msg, frame);
        slot->received = TRUE;
        if (hdr.SeqNum == state->NFE)
        {
            Msg m;

            while (slot->received)
            {
                deliver(HLP, &slot->msg);
                msgDestroy(&slot->msg);
```

```
                slot->received = FALSE;
                slot = &state->recvQ[++state->NFE % RWS];
            }
            /* send ACK: */
            prepare_ack(&m, state->NFE - 1);
            send(LINK, &m);
            msgDestroy(&m);
        }
    }
    return SUCCESS;
}
```

Finally, `swpInWindow` is a simple subroutine that checks to see if a given sequence number falls between some minimum and maximum sequence number.

```
static bool
swpInWindow(SwpSeqno seqno, SwpSeqno min, SwpSeqno max)
{
    SwpSeqno pos, maxpos;

    pos    = seqno - min;       /* pos *should* be in range [0..MAX) */
    maxpos = max - min + 1;     /* maxpos is in range [0..MAX] */
    return pos < maxpos;
}
```

Frame Order and Flow Control

The sliding window protocol is perhaps the best known algorithm in computer networking. What is easily confusing about the algorithm, however, is that it can be used to serve three different roles. The first role is the one we have been concentrating on in this section—to reliably deliver frames across an unreliable link. (In general, the algorithm can be used to reliably deliver messages across an unreliable network.) This is the core function of the algorithm.

The second role that the sliding window algorithm can serve is to preserve the order in which frames are transmitted. This is easy to do at the receiver—since each frame has a sequence number, the receiver just makes sure that it does not pass a frame up to the next-higher-level protocol until it has already passed up all frames with a smaller sequence number. That is, the receiver buffers (i.e., does not pass along) out-of-order frames. The version of the sliding window algorithm described in this section does preserve frame order, although we could imagine a variation in which the receiver passes frames to the next protocol without waiting for all earlier frames to be delivered. A question we should ask ourselves is whether we really need the sliding window protocol to keep the frames in order at the link level, or whether, instead, this functionality should be implemented by a protocol higher in the stack.

The third role that the sliding window algorithm sometimes plays is to support *flow control*—a feedback mechanism by which the receiver is able to throttle the sender. Such a mechanism is used to keep the sender from over-running the receiver—that is, from transmitting more data than the receiver is able to process. This is usually accomplished by augmenting the sliding window protocol so that the receiver not only acknowledges frames it has received but also informs the sender of how many frames it has room to receive. The number of frames that the receiver is capable of receiving corresponds to how much free buffer space it has. As in the case of ordered delivery, we need to make sure that flow control is necessary at the link level before incorporating it into the sliding window protocol.

One important concept to take away from this discussion is the system design principle we call *separation of concerns*. That is, you must be careful to distinguish between different functions that are sometimes rolled together in one mechanism, and you must make sure that each function is necessary and being supported in the most effective way. In this particular case, reliable delivery, ordered delivery, and flow control are sometimes combined in a single sliding window protocol, and we should ask ourselves if this is the right thing to do at the link level.

Concurrent Logical Channels

The data link protocol used in the original ARPANET provides an interesting alternative to the sliding window protocol, in that it is able to keep the pipe full while still using the simple stop-and-wait algorithm. One important consequence of this approach is that the frames sent over a given link are not kept in any particular order. The protocol also implies nothing about flow control.

The idea underlying the ARPANET protocol, which we refer to as *concurrent logical channels*, is to multiplex several logical channels onto a single point-to-point link and to run the stop-and-wait algorithm on each of these logical channels. There is no relationship maintained among the frames sent on any of the logical channels, yet because a different frame can be outstanding on each of the several logical channels the sender can keep the link full.

More precisely, the sender keeps 3 bits of state for each channel: a boolean, saying whether the channel is currently busy; the 1-bit sequence number to use the next time a frame is sent on this logical channel; and the next sequence number to expect on a frame that arrives on this channel. When the node has a frame to send, it uses the lowest idle channel, and otherwise it behaves just like stop-and-wait.

In practice, the ARPANET supported 8 logical channels over each ground link and 16 over each satellite link. In the ground-link case, the header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits. This is exactly the number of bits the sliding window protocol requires to support up to 8 outstanding frames on the link when `RWS = SWS`.

5.2 Reliable Byte Stream (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol is one that offers a reliable, connection-oriented, byte-stream service. Such a service has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. The Internet's Transmission Control Protocol is probably the most widely used protocol of this type; it is also the most carefully tuned. It is for these two reasons that this section studies TCP in detail, although we identify and discuss alternative design choices at the end of the section.

In terms of the properties of transport protocols given in the problem statement at the start of this chapter, TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time. Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.

In addition to the above features, TCP also implements a highly tuned congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from over-running the receiver, but so as to keep the sender from overloading the network. A description of TCP's congestion-control mechanism is postponed until the next chapter, where we discuss it in the larger context of how network resources are fairly allocated.

Since many people confuse congestion control and flow control, we restate the difference. *Flow control* involves preventing senders from over-running the capacity of receivers. *Congestion control* involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded. Thus, flow control is an end-to-end issue, while congestion control is concerned with how hosts and networks interact.

End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm as is often used at the link level, because TCP runs over the Internet rather than a physical point-to-point link, there are many important differences. This subsection identifies these differences and explains how they complicate TCP. The following subsections then describe how TCP addresses these and other complications.

First, whereas the link-level sliding window algorithm presented runs over a single physical link that always connects the same two computers, TCP supports logical connections between processes that are running on any two computers in the Internet. This means that TCP needs an explicit connection establishment phase during which the two sides of the connection agree to exchange data with each other. This difference is analogous to having to dial up the other party, rather than having a dedicated phone line. TCP also has an explicit connection teardown phase. One of the things that happens during connection establishment is that the two parties establish some shared state to enable the sliding window algorithm to begin. Connection teardown is needed so each host knows it is OK to free this state.

Second, whereas a single physical link that always connects the same two computers has a fixed round-trip time (RTT), TCP connections are likely to have widely different round-trip times. For example, a TCP connection between a host in San Francisco and a host in Boston, which are separated by several thousand kilometers, might have an RTT of 100 ms, while a TCP connection between two hosts in the same room, only a few meters apart, might have an RTT of only 1 ms. The same TCP protocol must be able to support both of these connections. To make matters worse, the TCP connection between hosts in San Francisco and Boston might have an RTT of 100 ms at 3 a.m., but an RTT of 500 ms at 3 p.m. Variations in the RTT are even possible during a single TCP connection that lasts only a few

minutes. What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive. (Certainly, the timeout for a point-to-point link must be a settable parameter, but it is not necessary to adapt this timer for a particular pair of nodes.)

A third difference is that packets may be reordered as they cross the Internet, but this is not possible on a point-to-point link where the first packet put into one end of the link must be the first to appear at the other end. Packets that are slightly out of order do not cause a problem since the sliding window algorithm can reorder packets correctly using the sequence number. The real issue is how far out of order packets can get or, said another way, how late a packet can arrive at the destination. In the worst case, a packet can be delayed in the Internet until the IP time to live (`TTL`) field expires, at which time the packet is discarded (and hence there is no danger of it arriving late). Knowing that IP throws packets away after their `TTL` expires, TCP assumes that each packet has a maximum lifetime. The exact lifetime, known as the *maximum segment lifetime* (MSL), is an engineering choice. The current recommended setting is 120 seconds. Keep in mind that IP does not directly enforce this 120-second value; it is simply a conservative estimate that TCP makes of how long a packet might live in the Internet. The implication is significant—TCP has to be prepared for very old packets to suddenly show up at the receiver, potentially confusing the sliding window algorithm.

Fourth, the computers connected to a point-to-point link are generally engineered to support the link. For example, if a link's delay $\times$ bandwidth product is computed to be 8 KB—meaning that a window size is selected to allow up to 8 KB of data to be unacknowledged at a given time—then it is likely that the computers at either end of the link have the ability to buffer up to 8 KB of data. Designing the system otherwise would be silly. On the other hand, almost any kind of computer can be connected to the Internet, making the amount of resources dedicated to any one TCP connection highly variable, especially considering that any one host can potentially support hundreds of TCP connections at the same time. This means that TCP must include a mechanism that each side uses to "learn" what resources (e.g., how much buffer space) the other side is able to apply to the connection. This is the flow control issue.

Fifth, because the transmitting side of a directly connected link cannot send any faster than the bandwidth of the link allows, and only one host is pumping data into the link, it is not possible to unknowingly congest the link. Said another way, the load on the link is visible in the form of a queue of packets at the sender. In contrast, the sending side of a TCP connection has no idea what links will be traversed to reach the destination. For example, the sending machine might be directly connected to a relatively fast Ethernet—and capable of sending data at a rate of 10 Gbps—but somewhere out in the middle of the network, a 1.5-Mbps link must be traversed. And, to make matters worse, data being generated by many different sources might be trying to traverse this same slow link. This leads to the problem of network congestion. Discussion of this topic is delayed until the next chapter.

We conclude this discussion of end-to-end issues by comparing TCP's approach to providing a reliable/ordered delivery service with the approach used by virtual-circut-based networks like the historically important X.25 network. In TCP, the underlying IP network is assumed to be unreliable and to deliver messages out of order; TCP uses the sliding window algorithm on an end-to-end basis to provide reliable/ordered delivery. In contrast, X.25 networks use the sliding window protocol within the network, on a hop-by-hop basis. The assumption behind this approach is that if messages are delivered reliably and in order between each pair of nodes along the path between the source host and the destination host, then the end-to-end service also guarantees reliable/ordered delivery.

The problem with this latter approach is that a sequence of hop-by-hop guarantees does not necessarily add up to an end-to-end guarantee. First, if a heterogeneous link (say, an Ethernet) is added to one end of the path, then there is no guarantee that this hop will preserve the same service as the other hops. Second, just because the sliding window protocol guarantees that messages are delivered correctly from node A to node B, and then from node B to node C, it does not guarantee that node B behaves perfectly. For example, network nodes have been known to introduce errors into messages while transferring them from an input buffer to an output buffer. They have also been known to accidentally reorder messages. As a consequence of these small windows of vulnerability, it is still necessary to provide true end-to-end checks to guarantee reliable/ordered service, even though the lower levels of the system also implement that functionality.

This discussion serves to illustrate one of the most important principles in system design—the *end-to-end argument*. In a nutshell, the end-to-end argument says that a function (in our example, providing reliable/ordered delivery) should not be provided in the lower levels of the system unless it can be completely and correctly implemented at that level. Therefore, this rule argues in favor of the TCP/IP approach. This rule is not absolute, however. It does allow for functions to be incompletely provided at a low level as a performance optimization. This is why it is perfectly consistent with the end-to-end argument to perform error detection (e.g., CRC) on a hop-by-hop basis; detecting and retransmitting a single corrupt packet across one hop is preferable to having to retransmit an entire file end-to-end.

Segment Format

TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection. Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet. Instead, TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure. This situation is illustrated in Figure 1, which, for simplicity, shows data flowing in only one direction. Remember that, in general, a single TCP connection supports byte streams flowing in both directions.

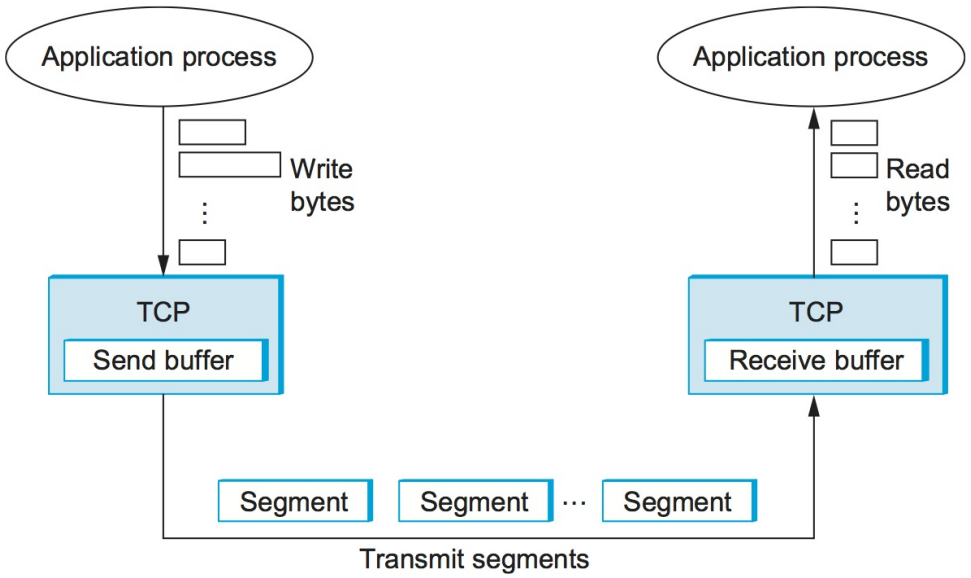

Figure 1. How TCP manages a byte stream.

The packets exchanged between TCP peers in Figure 1 are called *segments*, since each one carries a segment of the byte stream. Each TCP segment contains the header schematically depicted in Figure 2. The relevance of most of these fields will become apparent throughout this section. For now, we simply introduce them.

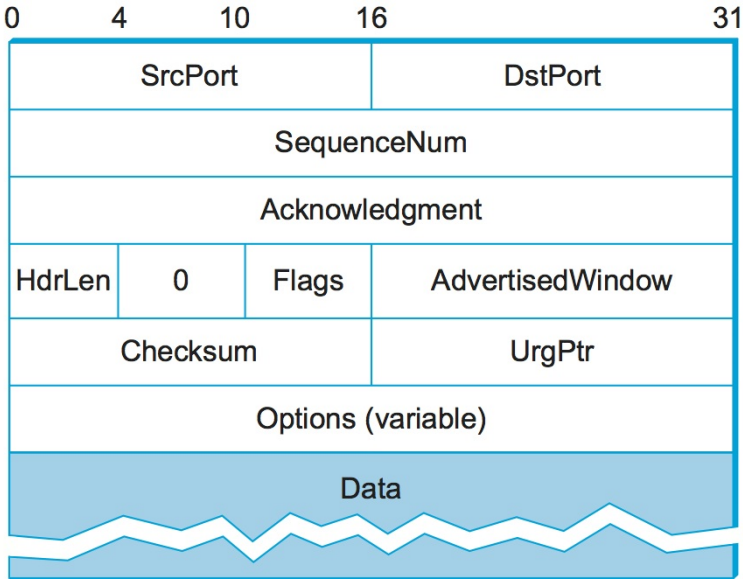

Figure 2. TCP header format.

The `SrcPort` and `DstPort` fields identify the source and destination ports, respectively, just as in UDP. These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. That is, TCP's demux key is given by the 4-tuple

```
(SrcPort, SrcIPAddr, DstPort, DstIPAddr)
```

Note that because TCP connections come and go, it is possible for a connection between a particular pair of ports to be established, used to send and receive data, and closed, and then at a later time for the same pair of ports to be involved in a second connection. We sometimes refer to this situation as two different *incarnations* of the same connection.

The `Acknowledgement`, `SequenceNum`, and `AdvertisedWindow` fields are all involved in TCP's sliding window algorithm. Because TCP is a byte-oriented protocol, each byte of data has a sequence number. The `SequenceNum` field contains the sequence number for the first byte of data carried in that segment, and the `Acknowledgement` and `AdvertisedWindow` fields carry information about the flow of data going in the other direction. To simplify our discussion, we ignore the fact that data can flow in both directions, and we concentrate on data that has a particular `SequenceNum` flowing in one direction and `Acknowledgement` and `AdvertisedWindow` values flowing in the opposite direction, as illustrated in Figure 3. The use of these three fields is described more fully later in this chapter.

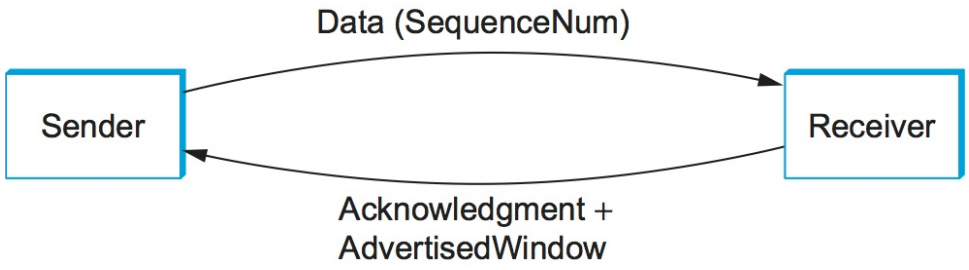

Figure 3. Simplified illustration (showing only one direction) of the TCP process, with data flow in one direction and ACKs in the other.

The 6-bit `Flags` field is used to relay control information between TCP peers. The possible flags include `SYN`, `FIN`, `RESET`, `PUSH`, `URG`, and `ACK`. The `SYN` and `FIN` flags are used when establishing and terminating a TCP connection, respectively. Their use is described in a later section. The `ACK` flag is set any time the `Acknowledgement` field is valid, implying that the receiver should pay attention to it. The `URG` flag signifies that this segment contains

urgent data. When this flag is set, the `UrgPtr` field indicates where the nonurgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including a value of `UrgPtr` bytes into the segment. The `PUSH` flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact. We discuss these last two features more in a later section. Finally, the `RESET` flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

Finally, the `Checksum` field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header. The checksum is required for TCP in both IPv4 and IPv6. Also, since the TCP header is of variable length (options can be attached after the mandatory fields), a `HdrLen` field is included that gives the length of the header in 32-bit words. This field is also known as the `Offset` field, since it measures the offset from the start of the packet to the start of the data.

Connection Establishment and Termination

A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection. (Recall from Chapter 1 that a party wanting to initiate a connection performs an active open, while a party willing to accept a connection does a passive open.) Only after this connection establishment phase is over do the two sides begin sending data. Likewise, as soon as a participant is done sending data, it closes one direction of the connection, which causes TCP to initiate a round of connection termination messages. Notice that, while connection setup is an asymmetric activity (one side does a passive open and the other side does an active open), connection teardown is symmetric (each side has to close the connection independently). Therefore, it is possible for one side to have done a close, meaning that it can no longer send data, but for the other side to keep the other half of the bidirectional connection open and to continue sending data.

> To be more precise, connection setup can be symmetric, with both sides trying to open the connection at the same time, but the common case is for one side to do an active open and the other side to do a passive open.

Three-Way Handshake

The algorithm used by TCP to establish and terminate a connection is called a *three-way handshake*. We first describe the basic algorithm and then show how it is used by TCP. The three-way handshake involves the exchange of three messages between the client and the server, as illustrated by the timeline given in Figure 4.

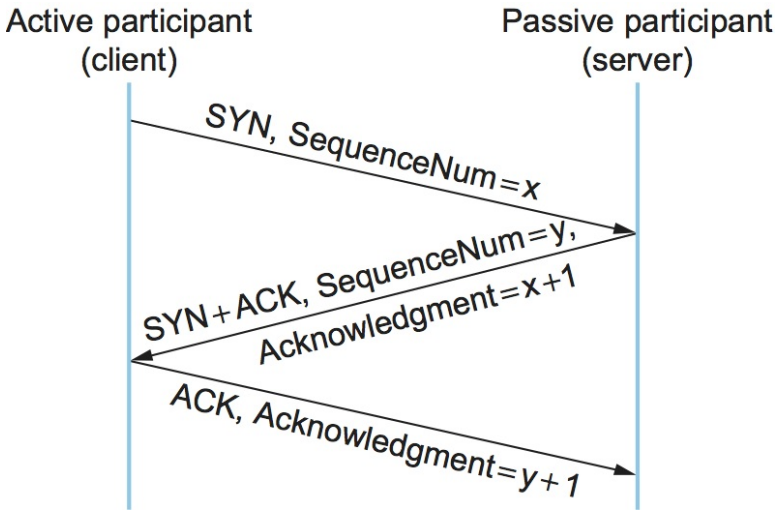

Figure 4. Timeline for three-way handshake algorithm.

The idea is that two parties want to agree on a set of parameters, which, in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective byte streams. In general, the parameters might be any facts that each side wants the other to know about. First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (`Flags` = `SYN` , `SequenceNum` = x). The server then responds with a single segment that both acknowledges the client's sequence number (`Flags = ACK, Ack = x + 1`) and states its own beginning sequence number (`Flags = SYN, SequenceNum = y`). That is, both the `SYN` and `ACK` bits are set in the `Flags` field of this second message. Finally, the client responds with a third segment that acknowledges the server's sequence number (`Flags = ACK, Ack = y + 1`). The reason why each side acknowledges a sequence number that is one larger than the one sent is that the `Acknowledgement` field actually identifies the "next sequence number expected," thereby implicitly acknowledging all earlier sequence numbers. Although not shown in this timeline, a timer is scheduled for each of the first two segments, and if the expected response is not received the segment is retransmitted.

You may be asking yourself why the client and server have to exchange starting sequence numbers with each other at connection setup time. It would be simpler if each side simply started at some "well-known" sequence number, such as 0. In fact, the TCP specification requires that each side of a connection select an initial starting sequence number at random. The reason for this is to protect against two incarnations of the same connection reusing the same sequence numbers too soon—that is, while there is still a chance that a segment from an earlier incarnation of a connection might interfere with a later incarnation of the connection.

State-Transition Diagram

TCP is complex enough that its specification includes a state-transition diagram. A copy of this diagram is given in Figure 5. This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED). Everything that goes on while a connection is open—that is, the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.

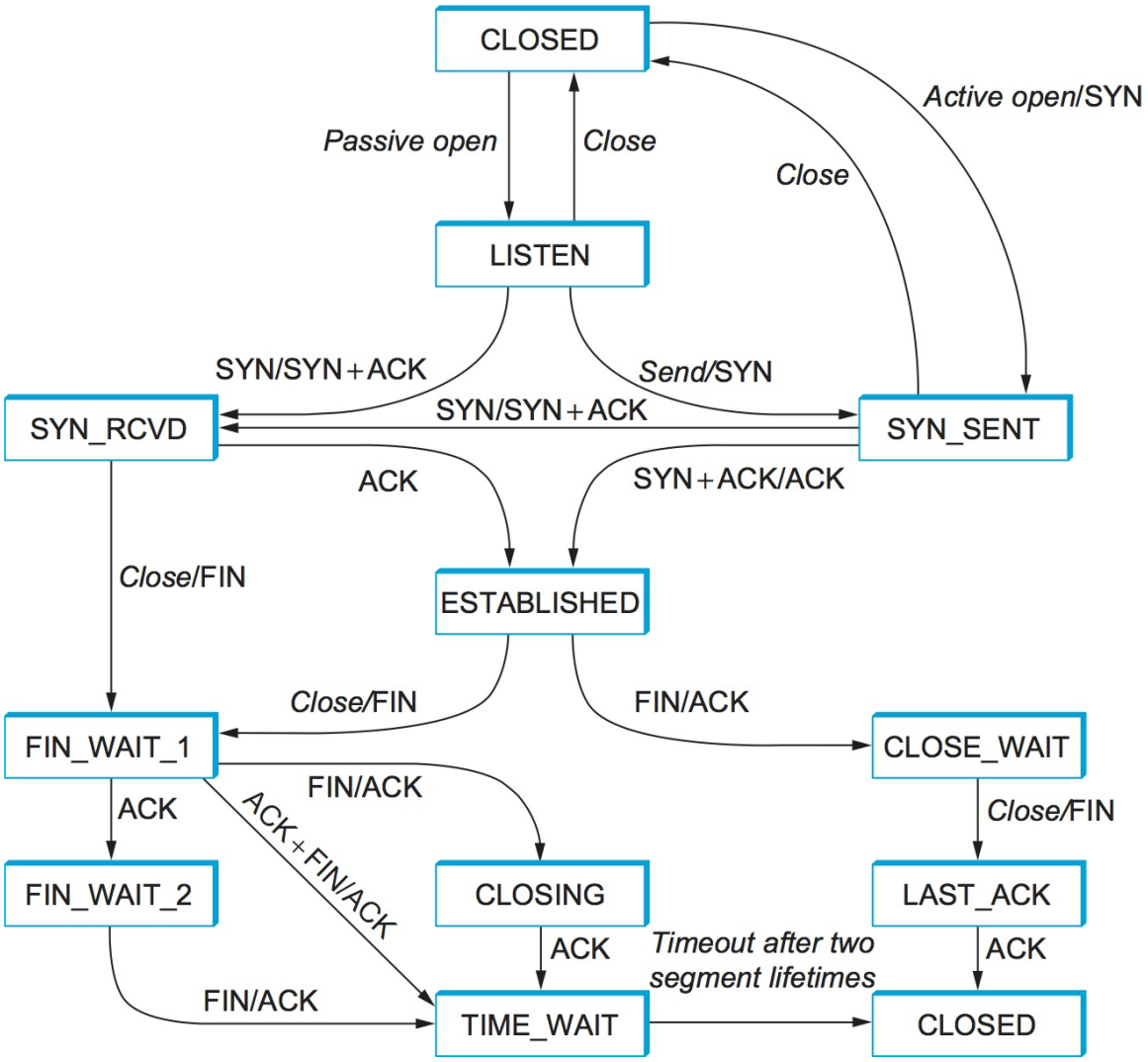

Figure 5. TCP state-transition diagram.

TCP's state-transition diagram is fairly easy to understand. Each box denotes a state that one end of a TCP connection can find itself in. All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event/action*. Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the `SYN` flag set), the connection makes a transition to the SYN_RCVD state and takes the action of replying with an `ACK+SYN` segment.

Notice that two kinds of events trigger a state transition: (1) a segment arrives from the peer (e.g., the event on the arc from LISTEN to SYN_RCVD), or (2) the local application process invokes an operation on TCP (e.g., the *active open* event on the arc from CLOSED to SYN_SENT). In other words, TCP's state-transition diagram effectively defines the *semantics* of both its peer-to-peer interface and its service interface. The *syntax* of these two interfaces is given by the segment format (as illustrated in Figure 2) and by some application programming interface, such as the socket API, respectively.

Now let's trace the typical transitions taken through the diagram in Figure 5. Keep in mind that at each end of the connection, TCP makes different transitions from state to state. When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state. At some later time, the client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN_SENT state. When the SYN segment arrives at the server, it moves to the SYN_RCVD state and responds with

a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.

There are three things to notice about the connection establishment half of the state-transition diagram. First, if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the `ACK` flag set, and the correct value in the `Acknowledgement` field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP—every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.

The second thing to notice about the state-transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then for it to change its mind about waiting for the other side and instead actively establish the connection. To the best of our knowledge, this is a feature of TCP that no application process actually takes advantage of.

The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be present if the expected response does not happen. These retransmissions are not depicted in the state-transition diagram. If after several tries the expected response does not arrive, TCP gives up and returns to the CLOSED state.

Turning our attention now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection. If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side. This complicates the state-transition diagram because it must account for the possibility that the two sides invoke the *close* operator at the same time, as well as the possibility that first one side invokes close and then, at some later time, the other side invokes close. Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:

- This side closes first: ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.
- The other side closes first: ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED.
- Both sides close at the same time: ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.

There is actually a fourth, although rare, sequence of transitions that leads to the CLOSED state; it follows the arc from FIN_WAIT_1 to TIME_WAIT. We leave it as an exercise for you to figure out what combination of circumstances leads to this fourth possibility.

The main thing to recognize about connection teardown is that a connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that, while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

Sliding Window Revisited

We are now ready to discuss TCP's variant of the sliding window algorithm, which serves several purposes: (1) it guarantees the reliable delivery of data, (2) it ensures that data is delivered in order, and (3) it enforces flow control between the sender and the receiver. TCP's use of the sliding window algorithm is the same as at the link level in the case of the first two of these three functions. Where TCP differs from the link-level algorithm is that it folds the flow-control function in as well. In particular, rather than having a fixed-size sliding window, the receiver *advertises* a window size to the sender. This is done using the `AdvertisedWindow` field in the TCP header. The sender is then limited to having no more than a value of `AdvertisedWindow` bytes of unacknowledged data at any given time. The receiver selects a suitable value for `AdvertisedWindow` based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from over-running the receiver's buffer. We discuss this at greater length below.

Reliable and Ordered Delivery

To see how the sending and receiving sides of TCP interact with each other to implement reliable and ordered delivery, consider the situation illustrated in Figure 6. TCP on the sending side maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application but not transmitted. On the receiving side, TCP maintains a receive buffer. This buffer holds data that arrives out of order, as well as data that is in the correct order (i.e., there are no missing bytes earlier in the stream) but that the application process has not yet had the chance to read.

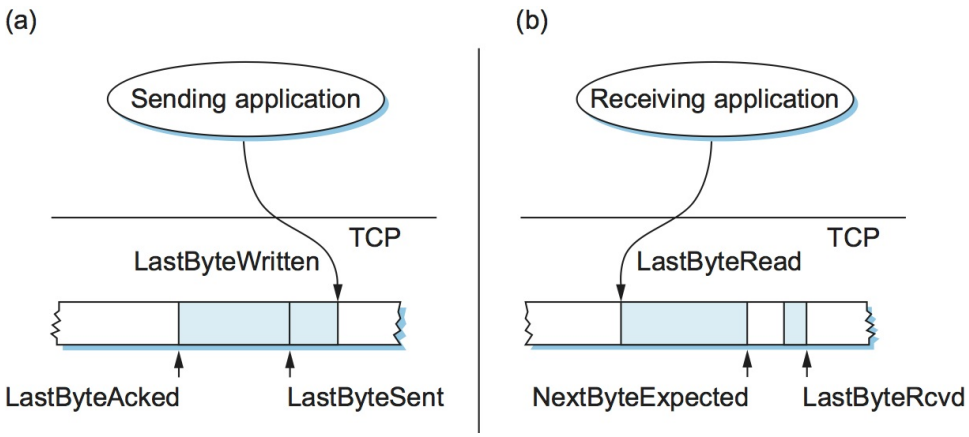

Figure 6. Relationship between TCP send buffer (a) and receive buffer (b).

To make the following discussion simpler to follow, we initially ignore the fact that both the buffers and the sequence numbers are of some finite size and hence will eventually wrap around. Also, we do not distinguish between a pointer into a buffer where a particular byte of data is stored and the sequence number for that byte.

Looking first at the sending side, three pointers are maintained into the send buffer, each with an obvious meaning: `LastByteAcked`, `LastByteSent`, and `LastByteWritten`. Clearly,

```
LastByteAcked <= LastByteSent
```

since the receiver cannot have acknowledged a byte that has not yet been sent, and

```
LastByteSent <= LastByteWritten
```

since TCP cannot send a byte that the application process has not yet written. Also note that none of the bytes to the left of `LastByteAcked` need to be saved in the buffer because they have already been acknowledged, and none of the bytes to the right of `LastByteWritten` need to be buffered because they have not yet been generated.

A similar set of pointers (sequence numbers) are maintained on the receiving side: `LastByteRead`, `NextByteExpected`, and `LastByteRcvd`. The inequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

```
LastByteRead < NextByteExpected
```

is true because a byte cannot be read by the application until it is received *and* all preceding bytes have also been received. `NextByteExpected` points to the byte immediately after the latest byte to meet this criterion. Second,

```
NextByteExpected <= LastByteRcvd + 1
```

since, if data has arrived in order, `NextByteExpected` points to the byte after `LastByteRcvd`, whereas if data has arrived out of order, then `NextByteExpected` points to the start of the first gap in the data, as in Figure 6. Note that bytes to the left of `LastByteRead` need not be buffered because they have already been read by the local application process, and bytes to the right of `LastByteRcvd` need not be buffered because they have not yet arrived.

Flow Control

Most of the above discussion is similar to that found in the standard sliding window algorithm; the only real difference is that this time we elaborated on the fact that the sending and receiving application processes are filling and emptying their local buffer, respectively. (The earlier discussion glossed over the fact that data arriving from an upstream node was filling the send buffer and data being transmitted to a downstream node was emptying the receive buffer.)

You should make sure you understand this much before proceeding because now comes the point where the two algorithms differ more significantly. In what follows, we reintroduce the fact that both buffers are of some finite size, denoted `MaxSendBuffer` and `MaxRcvBuffer`, although we don't worry about the details of how they are implemented. In other words, we are only interested in the number of bytes being buffered, not in where those bytes are actually stored.

Recall that in a sliding window protocol, the size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer. Observe that TCP on the receive side must keep

```
LastByteRcvd - LastByteRead <= MaxRcvBuffer
```

to avoid overflowing its buffer. It therefore advertises a window size of

```
AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)
```

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding bytes have also arrived. In addition, `LastByteRcvd` moves to the right (is incremented), meaning that the advertised window potentially shrinks. Whether or not it shrinks depends on how fast the local application process is consuming data. If the local process is reading data just as fast as it arrives (causing `LastByteRead` to be incremented at the same rate as `LastByteRcvd`), then the advertised window stays open (i.e., `AdvertisedWindow = MaxRcvBuffer`). If, however, the receiving process falls behind, perhaps because it performs a very expensive operation on each byte of data that it reads, then the advertised window grows smaller with every segment that arrives, until it eventually goes to 0.

TCP on the send side must then adhere to the advertised window it gets from the receiver. This means that at any given time, it must ensure that

```
LastByteSent - LastByteAcked <= AdvertisedWindow
```

Said another way, the sender computes an *effective* window that limits how much data it can send:

```
EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
```

Clearly, `EffectiveWindow` must be greater than 0 before the source can send more data. It is possible, therefore, that a segment arrives acknowledging x bytes, thereby allowing the sender to increment `LastByteAcked` by x, but because the receiving process was not reading any data, the advertised window is now x bytes smaller than the time before. In such a situation, the sender would be able to free buffer space, but not to send any more data.

All the while this is going on, the send side must also make sure that the local application process does not overflow the send buffer—that is,

```
LastByteWritten - LastByteAcked <= MaxSendBuffer
```

If the sending process tries to write y bytes to TCP, but

```
(LastByteWritten - LastByteAcked) + y > MaxSendBuffer
```

then TCP blocks the sending process and does not allow it to generate more data.

It is now possible to understand how a slow receiving process ultimately stops a fast sending process. First, the receive buffer fills up, which means the advertised window shrinks to 0. An advertised window of 0 means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As soon as the receiving process starts to read data again, the receive-side TCP is able to open its window back up, which allows the send-side TCP to transmit data out of its buffer. When this data is eventually acknowledged, `LastByteAcked` is incremented, the buffer space holding this acknowledged data becomes free, and the sending process is unblocked and allowed to proceed.

There is only one remaining detail that must be resolved—how does the sending side know that the advertised window is no longer 0? As mentioned above, TCP *always* sends a segment in response to a received data segment, and this response contains the latest values for the `Acknowledge` and `AdvertisedWindow` fields, even if these values have not changed since the last time they were sent. The problem is this. Once the receive side has advertised a window size of 0, the sender is not permitted to send any more data, which means it has no way to discover that the advertised window is no longer 0 at some time in the future. TCP on the receive side does not spontaneously send nondata segments; it only sends them in response to an arriving data segment.

TCP deals with this situation as follows. Whenever the other side advertises a window size of 0, the sending side persists in sending a segment with 1 byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments triggers a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a nonzero advertised window.

Note that the reason the sending side periodically sends this probe segment is that TCP is designed to make the receive side as simple as possible—it simply responds to segments from the sender, and it never initiates any activity on its own. This is an example of a well-recognized (although not universally applied) protocol design rule, which, for lack of a better name, we call the *smart sender/ dumb receiver* rule. Recall that we saw another example of this rule when we discussed the use of NAKs in sliding window algorithm.

Protecting against Wraparound

This subsection and the next consider the size of the `SequenceNum` and `AdvertisedWindow` fields and the implications of their sizes on TCP's correctness and performance. TCP's `SequenceNum` field is 32 bits long, and its `AdvertisedWindow` field is 16 bits long, meaning that TCP has easily satisfied the requirement of the sliding window algorithm that the

sequence number space be twice as big as the window size: $2^{32} >> 2 \times 2^{16}$. However, this requirement is not the interesting thing about these two fields. Consider each field in turn.

The relevance of the 32-bit sequence number space is that the sequence number used on a given connection might wrap around—a byte with sequence number S could be sent at one time, and then at a later time a second byte with the same sequence number S might be sent. Once again, we assume that packets cannot survive in the Internet for longer than the recommended MSL. Thus, we currently need to make sure that the sequence number does not wrap around within a 120-second period of time. Whether or not this happens depends on how fast data can be transmitted over the Internet—that is, how fast the 32-bit sequence number space can be consumed. (This discussion assumes that we are trying to consume the sequence number space as fast as possible, but of course we will be if we are doing our job of keeping the pipe full.) Table 1 shows how long it takes for the sequence number to wrap around on networks with various bandwidths.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-48 (2.5 Gbps)	14 seconds
OC-192 (10 Gbps)	3 seconds
10GigE (10 Gbps)	3 seconds

Table 1. Time Until 32-Bit Sequence Number Space Wraps Around.

As you can see, the 32-bit sequence number space is adequate at modest bandwidths, but given that OC-192 links are now common in the Internet backbone, and that most servers now come with 10Gig Ethernet (or 10 Gbps) interfaces, we're now well-past the point where 32 bits is too small. Fortunately, the IETF has worked out an extension to TCP that effectively extends the sequence number space to protect against the sequence number wrapping around. This and related extensions are described in a later section.

Keeping the Pipe Full

The relevance of the 16-bit `AdvertisedWindow` field is that it must be big enough to allow the sender to keep the pipe full. Clearly, the receiver is free to not open the window as large as the `AdvertisedWindow` field allows; we are interested in the situation in which the receiver has enough buffer space to handle as much data as the largest possible `AdvertisedWindow` allows.

In this case, it is not just the network bandwidth but the delay $\times$ bandwidth product that dictates how big the `AdvertisedWindow` field needs to be—the window needs to be opened far enough to allow a full delay $\times$ bandwidth product's worth of data to be transmitted. Assuming an RTT of 100 ms (a typical number for a cross-country connection in the United States), Table 2 gives the delay $\times$ bandwidth product for several network technologies.

Bandwidth	Delay $\times$ Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB

OC-3 (155 Mbps)	1.8 MB
OC-48 (2.5 Gbps)	29.6 MB
OC-192 (10 Gbps)	118.4 MB
10GigE (10 Gbps)	118.4 MB

Table 2. Required Window Size for 100-ms RTT.

As you can see, TCP's `AdvertisedWindow` field is in even worse shape than its `SequenceNum` field—it is not big enough to handle even a T3 connection across the continental United States, since a 16-bit field allows us to advertise a window of only 64 KB. The very same TCP extension mentioned above provides a mechanism for effectively increasing the size of the advertised window.

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 7 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 7(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 create larger containers, as in Figure 7(b). This scenario was discovered when early implementations of TCP regularly found themselves filling the network with tiny segments.

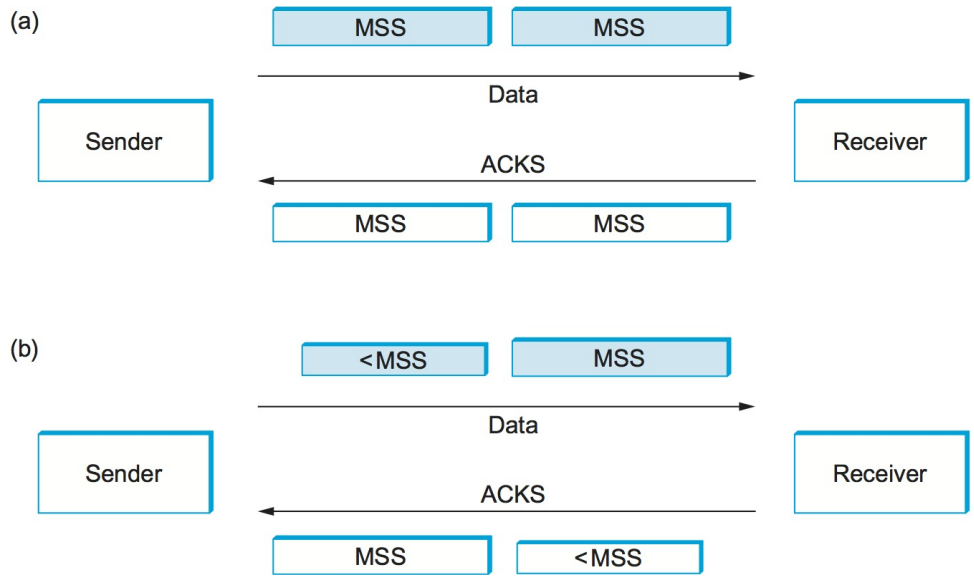

Figure 7. Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small "container" enters the system and continues to circulate.

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.

Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We now describe this mechanism and how it has evolved over time as the Internet community has gained more experience using TCP.

Original Algorithm

We begin with a simple algorithm for computing a timeout value between a pair of hosts. This is the algorithm that was originally described in the TCP specification—and the following description presents it in those terms—but it could be used by any end-to-end protocol.

The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again, and then takes the difference between these two times as a `SampleRTT`. TCP then computes an `EstimatedRTT` as a weighted average between the previous estimate and this new sample. That is,

```
EstimatedRTT = alpha x EstimatedRTT + (1 - alpha) x SampleRTT
```

The parameter `alpha` is selected to *smooth* the `EstimatedRTT`. A small `alpha` tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations. On the other hand, a large `alpha` is more stable but perhaps not quick enough to adapt to real changes. The original TCP specification recommended a setting of `alpha` between 0.8 and 0.9. TCP then uses `EstimatedRTT` to compute the timeout in a rather conservative way:

```
TimeOut = 2 x EstimatedRTT
```

Karn/Partridge Algorithm

After several years of use on the Internet, a rather obvious flaw was discovered in this simple algorithm. The problem was that an ACK does not really acknowledge a transmission; it actually acknowledges the receipt of data. In other words, whenever a segment is retransmitted and then an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment for the purpose of measuring the sample RTT. It is necessary to know which transmission to associate it with so as to compute an accurate `SampleRTT`. As illustrated in Figure 8, if you assume that the ACK is for the original transmission but it was really for the second, then the `SampleRTT` is too large (a); if you assume that the ACK is for the second transmission but it was actually for the first, then the `SampleRTT` is too small (b).

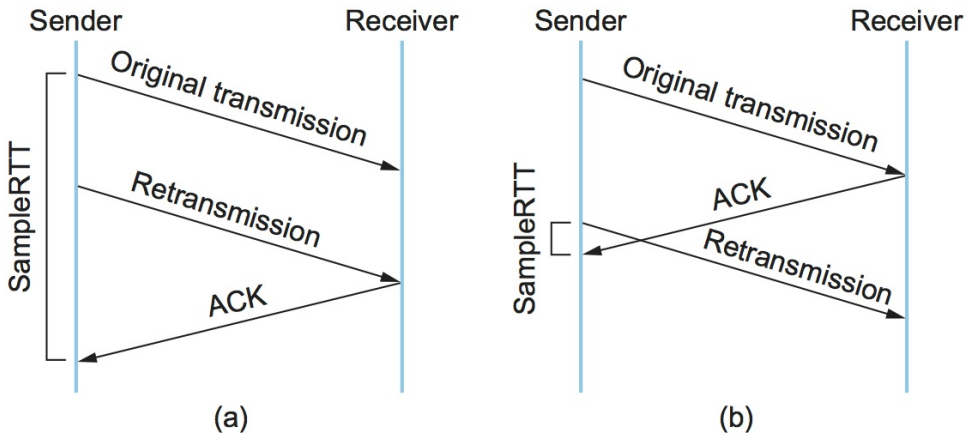

Figure 8. Associating the ACK with (a) original transmission versus (b) retransmission.

The solution, which was proposed in 1987, is surprisingly simple. Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures `SampleRTT` for segments that have been sent only once. This solution is known as the Karn/Partridge algorithm, after its inventors. Their proposed fix also includes a second small change to TCP's timeout mechanism. Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last `EstimatedRTT`. That is, Karn and Partridge proposed that TCP use exponential backoff, similar to what the Ethernet does. The motivation for using exponential backoff is simple: Congestion is the most likely cause of lost segments, meaning that the TCP source should not react too aggressively to a timeout. In fact, the more times the connection times out, the more cautious the source should become. We will see this idea again, embodied in a much more sophisticated mechanism, in the next chapter.

Jacobson/Karels Algorithm

The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion. Their approach was designed to fix some of the causes of that congestion, but, although it was an improvement, the congestion was not eliminated. The following year (1988), two other researchers—Jacobson and Karels—proposed a more drastic change to TCP to battle congestion. The bulk of that proposed change is described in the next chapter. Here, we focus on the aspect of that proposal that is related to deciding when to time out and retransmit a segment.

As an aside, it should be clear how the timeout mechanism is related to congestion—if you time out too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network. The other reason for needing an accurate timeout value is that a timeout is taken to imply congestion, which triggers a congestion-control mechanism. Finally, note that there is nothing about the Jacobson/Karels timeout computation that is specific to TCP. It could be used by any end-to-end protocol.

The main problem with the original computation is that it does not take the variance of the sample RTTs into account. Intuitively, if the variation among samples is small, then the `EstimatedRTT` can be better trusted and there is no reason for multiplying this estimate by 2 to compute the timeout. On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the `EstimatedRTT`.

In the new approach, the sender measures a new `SampleRTT` as before. It then folds this new sample into the timeout calculation as follows:

```
Difference = SampleRTT - EstimatedRTT
EstimatedRTT = EstimatedRTT + (delta x Difference)
Deviation = Deviation + delta (|Difference| - Deviation)
```

where `delta` is a fraction between 0 and 1. That is, we calculate both the mean RTT and the variation in that mean.

TCP then computes the timeout value as a function of both `EstimatedRTT` and `Deviation` as follows:

```
TimeOut = mu x EstimatedRTT + phi x Deviation
```

where based on experience, `mu` is typically set to 1 and `phi` is set to 4. Thus, when the variance is small, `TimeOut` is close to `EstimatedRTT`; a large variance causes the `Deviation` term to dominate the calculation.

Implementation

There are two items of note regarding the implementation of timeouts in TCP. The first is that it is possible to implement the calculation for `EstimatedRTT` and `Deviation` without using floating-point arithmetic. Instead, the whole calculation is scaled by 2^n, with `delta` selected to be $1/2^n$. This allows us to do integer arithmetic, implementing multiplication and division using shifts, thereby achieving higher performance. The resulting calculation is given by the following code fragment, where n=3 (i.e., `delta = 1/8`). Note that `EstimatedRTT` and `Deviation` are stored in their scaled-up forms, while the value of `SampleRTT` at the start of the code and of `TimeOut` at the end are real, unscaled values. If you find the code hard to follow, you might want to try plugging some real numbers into it and verifying that it gives the same results as the equations above.

```
{
    SampleRTT -= (EstimatedRTT >> 3);
    EstimatedRTT += SampleRTT;
    if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
    SampleRTT -= (Deviation >> 3);
    Deviation += SampleRTT;
    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
}
```

The second point of note is that the Jacobson/Karels algorithm is only as good as the clock used to read the current time. On typical Unix implementations at the time, the clock granularity was as large as 500 ms, which is significantly larger than the average cross-country RTT of somewhere between 100 and 200 ms. To make matters worse, the Unix implementation of TCP only checked to see if a timeout should happen every time this 500-ms clock ticked and would only take a sample of the round-trip time once per RTT. The combination of these two factors could mean that a timeout would happen 1 second after the segment was transmitted. Once again, the extensions to TCP include a mechanism that makes this RTT calculation a bit more precise.

All of the retransmission algorithms we have discussed are based on acknowledgment timeouts, which indicate that a segment has probably been lost. Note that a timeout does not, however, tell the sender whether any segments it sent after the lost segment were successfully received. This is because TCP acknowledgments are cumulative; they identify only the last segment that was received without any preceding gaps. The reception of segments that occur after a gap grows more frequent as faster networks lead to larger windows. If ACKs also told the sender which subsequent segments, if any, had been received, then the sender could be more intelligent about which segments it retransmits, draw better conclusions about the state of congestion, and make better RTT estimates. A TCP extension supporting this is described in a later section.

Record Boundaries

Since TCP is a byte-stream protocol, the number of bytes written by the sender are not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side the application reads 5 bytes at a time inside a loop that iterates 6