In this chapter we will study the design of layer 2, the data link layer. This study deals with the algorithms for achieving reliable, efficient communication between two adjacent machines at the data link layer. By adjacent, we mean that the two machines are connected by a communication channel that acts conceptually like a wire (e.g., a coaxial cable, telephone line, or point-to-point wireless channel). The essential property of a channel that makes it “wire-like” is that the bits are delivered in exactly the same order in which they are sent.

At first you might think this problem is so trivial that there is no software to study—machine $A$ just puts the bits on the wire, and machine $B$ just takes them off. Unfortunately, communication circuits make errors occasionally. Furthermore, they have only a finite data rate, and there is a nonzero propagation delay between the time a bit is sent and the time it is received. These limitations have important implications for the efficiency of the data transfer. The protocols used for communications must take all these factors into consideration. These protocols are the subject of this chapter.

After an introduction to the key design issues present in the data link layer, we will start our study of its protocols by looking at the nature of errors, their causes, and how they can be detected and corrected. Then we will study a series of increasingly complex protocols, each one solving more and more of the problems present in this layer. Finally, we will conclude with an examination of protocol modeling and correctness and give some examples of data link protocols.
3.1 DATA LINK LAYER DESIGN ISSUES

The data link layer has a number of specific functions to carry out. These functions include

1. Providing a well-defined service interface to the network layer.
2. Dealing with transmission errors.
3. Regulating the flow of data so that slow receivers are not swamped by fast senders.

To accomplish these goals, the data link layer takes the packets it gets from the network layer and encapsulates them into frames for transmission. Each frame contains a frame header, a payload field for holding the packet, and a frame trailer, as illustrated in Fig. 3-1. Frame management forms the heart of what the data link layer does. In the following sections we will examine all the above-mentioned issues in detail.

Although this chapter is explicitly about the data link layer and the data link protocols, many of the principles we will study here, such as error control and flow control, are also found in transport and other protocols as well. In fact, in many networks, these functions are found only in the upper layers and not in the data link layer. However, no matter where they are found, the principles are pretty much the same, so it does not really matter where we study them. In the data link layer they often show up in their simplest and purest forms, making this a good place to examine them in detail.

3.1.1 Services Provided to the Network Layer

The function of the data link layer is to provide services to the network layer. The principal service is transferring data from the network layer on the source machine to the network layer on the destination machine. On the source machine there is an entity, call it a process, in the network layer that hands some bits to the

![Figure 3-1. Relationship between packets and frames.](image)
data link layer for transmission to the destination. The job of the data link layer is to transmit the bits to the destination machine, so they can be handed over to the network layer there, as shown in Fig. 3-2(a). The actual transmission follows the path of Fig. 3-2(b), but it is easier to think in terms of two data link layer processes communicating using a data link protocol. For this reason, we will implicitly use the model of Fig. 3-2(a) throughout this chapter.

![Figure 3-2.](image)

The data link layer can be designed to offer various services. The actual services offered can vary from system to system. Three reasonable possibilities that are commonly provided are

1. Unacknowledged connectionless service.
2. Acknowledged connectionless service.
3. Acknowledged connection-oriented service.

Let us consider each of these in turn.

Unacknowledged connectionless service consists of having the source machine send independent frames to the destination machine without having the destination machine acknowledge them. No logical connection is established beforehand or released afterward. If a frame is lost due to noise on the line, no attempt is made to detect the loss or recover from it in the data link layer. This class of service is appropriate when the error rate is very low so recovery is left to higher layers. It is also appropriate for real-time traffic, such as voice, in which
late data are worse than bad data. Most LANs use unacknowledged connectionless service in the data link layer.

The next step up in terms of reliability is acknowledged connectionless service. When this service is offered, there are still no logical connections used, but each frame sent is individually acknowledged. In this way, the sender knows whether or not a frame has arrived correctly. If it has not arrived within a specified time interval, it can be sent again. This service is useful over unreliable channels, such as wireless systems.

It is perhaps worth emphasizing that providing acknowledgements in the data link layer is just an optimization, never a requirement. The network layer can always send a packet and wait for it to be acknowledged. If the acknowledgement is not forthcoming before the timer expires, the sender can just send the entire message again. The trouble with this strategy is that frames usually have a strict maximum length imposed by the hardware and network layer packets do not. If the average packet is broken up into, say, 10 frames, and 20 percent of all frames are lost, it may take a very long time for the packet to get through. If individual frames are acknowledged and retransmitted, entire packets get through much faster. On reliable channels, such as fiber, the overhead of a heavyweight data link protocol may be unnecessary, but on wireless channels it is well worth the cost due to their inherent unreliability.

Getting back to our services, the most sophisticated service the data link layer can provide to the network layer is connection-oriented service. With this service, the source and destination machines establish a connection before any data are transferred. Each frame sent over the connection is numbered, and the data link layer guarantees that each frame sent is indeed received. Furthermore, it guarantees that each frame is received exactly once and that all frames are received in the right order. With connectionless service, in contrast, it is conceivable that a lost acknowledgement causes a frame to be sent several times and thus received several times. Connection-oriented service, in contrast, provides the network layer processes with the equivalent of a reliable bit stream.

When connection-oriented service is used, transfers have three distinct phases. In the first phase the connection is established by having both sides initialize variables and counters needed to keep track of which frames have been received and which ones have not. In the second phase, one or more frames are actually transmitted. In the third and final phase, the connection is released, freeing up the variables, buffers, and other resources used to maintain the connection.

Consider a typical example: a WAN subnet consisting of routers connected by point-to-point leased telephone lines. When a frame arrives at a router, the hardware checks it for errors (using techniques we will study late in this chapter), then passes the frame to the data link layer software (which might be embedded in a chip on the network interface board). The data link layer software checks to see if this is the frame expected, and if so, gives the packet contained in the payload field to the routing software. The routing software chooses the appropriate
outgoing line and passes the packet back down to the data link layer software, which then transmits it. The flow over two routers is shown in Fig. 3-3.

![Figure 3-3. Placement of the data link protocol.](image)

The routing code frequently wants the job done right, that is, reliable, sequenced connections on each of the point-to-point lines. It does not want to be bothered too often with packets that got lost on the way. It is up to the data link protocol, shown in the dotted rectangle, to make unreliable communication lines look perfect or, at least, fairly good. As an aside, although we have shown multiple copies of the data link layer software in each router, in fact, one copy handles all the lines, with different tables and data structures for each one.

### 3.1.2 Framing

In order to provide service to the network layer, the data link layer must use the service provided to it by the physical layer. What the physical layer does is accept a raw bit stream and attempt to deliver it to the destination. This bit stream is not guaranteed to be error free. The number of bits received may be less than, equal to, or more than the number of bits transmitted, and they may have different values. It is up to the data link layer to detect, and if necessary, correct errors.

The usual approach is for the data link layer to break the bit stream up into discrete frames and compute the checksum for each frame. (Checksum algorithms will be discussed later in this chapter.) When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it (e.g., discarding the bad frame and possibly also sending back an error report).
Breaking the bit stream up into frames is more difficult than it at first appears. One way to achieve this framing is to insert time gaps between frames, much like the spaces between words in ordinary text. However, networks rarely make any guarantees about timing, so it is possible these gaps might be squeezed out, or other gaps might be inserted during transmission.

Since it is too risky to count on timing to mark the start and end of each frame, other methods have been devised. In this section we will look at four methods:

1. Character count.
2. Flag bytes with byte stuffing.
3. Starting and ending flags, with bit stuffing.
4. Physical layer coding violations.

The first framing method uses a field in the header to specify the number of characters in the frame. When the data link layer at the destination sees the character count, it knows how many characters follow, and hence where the end of the frame is. This technique is shown in Fig. 3-4(a) for four frames of sizes 5, 5, 8, and 8 characters respectively.

![Character count](image)

**Figure 3-4.** A character stream. (a) Without errors. (b) With one error.

The trouble with this algorithm is that the count can be garbled by a transmission error. For example, if the character count of 5 in the second frame of Fig. 3-4(b) becomes a 7, the destination will get out of synchronization and will be unable to locate the start of the next frame. Even if the checksum is incorrect so the destination knows that the frame is bad, it still has no way of telling where the next frame starts. Sending a frame back to the source asking for a retransmission does not help either, since the destination does not know how many characters to
skip over to get to the start of the retransmission. For this reason, the character count method is rarely used anymore.

The second framing method gets around the problem of resynchronization after an error by having each frame start and end with special bytes. In the past, the starting and ending bytes were different, but in recent years most protocols have used the same byte, called a **flag byte** as both the starting and ending delimiter, as shown in Fig. 3-5(a) as FLAG. In this way, if the receiver ever loses synchronization, it can just search for the flag byte to find the end of the current frame. Two consecutive flag bytes indicate the end of one frame and start of the next one.

![Diagram](https://via.placeholder.com/150)

**Figure 3-5.** (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.

A serious problem occurs with this method when binary data, such as object programs or floating-point numbers, are being transmitted. It may easily happen that the flag byte’s bit pattern occurs in the data, which will interfere with the framing. One way to solve this problem is to have the sender’s data link layer insert a special escape byte (ESC) just before each “accidental” flag byte in the data. The data link layer on the receiving end removes the escape byte before the data are given to the network layer. This technique is called **byte stuffing** or **character stuffing**. Thus a framing flag byte can be distinguished from one in the data by the absence or presence of an escape byte before it.

Of course, the next question is what happens if an escape byte occurs in the middle of the data? The answer is that it, too, is stuffed with escape. Thus any single escape byte is part of an escape sequence, whereas a doubled one indicates a single escape occurred naturally in the data. Some examples are shown in
Fig. 3-5(b). In all cases, the byte sequence delivered after destuffing is exactly the same as the original byte sequence.

The byte-stuffing scheme depicted in Fig. 3-5 is a slight simplification of the one used in the PPP protocol that most home computers use to communicate with their Internet service provider. We will discuss PPP later in this chapter.

A major disadvantage of using this framing method is that it is closely tied to the use of 8-bit characters. Not all character codes use 8-bit characters (UNICODE uses 16-bit characters, for example). As networks developed, the disadvantages of embedding the character code length in the framing mechanism became more and more obvious so a new technique had to be developed to allow arbitrary sized characters.

The new technique allows data frames to contain an arbitrary number of bits and allows character codes with an arbitrary number of bits per character. It works like this. Each frame begins and ends with a special bit pattern, 01111110 (in fact, a flag byte). Whenever the sender’s data link layer encounters five consecutive ones in the data, it automatically stuffs a 0 bit into the outgoing bit stream. This bit stuffing is analogous to byte stuffing, in which an escape byte is stuffed into the outgoing character stream before a flag byte in the data.

When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit. Just as byte stuffing is completely transparent to the network layer in both computers, so is bit stuffing. If the user data contain the flag pattern, 01111110, this flag is transmitted as 01111010 but stored in the receiver’s memory as 01111110. Figure 3-6 gives an example of bit stuffing.

![Figure 3-6. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver’s memory after destuffing.](image)

With bit stuffing, the boundary between two frames can be unambiguously recognized by the flag pattern. Thus if the receiver loses track of where it is, all it has to do is scan the input for flag sequences, since they can only occur at frame boundaries and never within the data.

The last method of framing is only applicable to networks in which the encoding on the physical medium contains some redundancy. For example, some LANs encode 1 bit of data by using 2 physical bits. Normally, a 1 bit is a high-low pair and a 0 bit is a low-high pair. The scheme means that every data bit has a
transition in the middle, making it easy for the receiver to locate the bit boundaries. The combinations high-high and low-low are not used for data, but are used for delimiting frames in some protocols.

As a final note on framing, many data link protocols use a combination of a character count with one of the other methods for extra safety. When a frame arrives, the count field is used to locate the end of the frame. Only if the appropriate delimiter is present at that position and the checksum is correct, is the frame accepted as valid. Otherwise, the input stream is scanned for the next delimiter.

### 3.1.3 Error Control

Having solved the problem of marking the start and end of each frame, we come to the next problem: how to make sure all frames are eventually delivered to the network layer at the destination, and in the proper order. Suppose that the sender just kept outputting frames without regard to whether they were arriving properly. This might be fine for unacknowledged connectionless service but would most certainly not be fine for reliable, connection-oriented service.

The usual way to ensure reliable delivery is to provide the sender with some feedback about what is happening at the other end of the line. Typically the protocol calls for the receiver to send back special control frames bearing positive or negative acknowledgements about the incoming frames. If the sender receives a positive acknowledgement about a frame, it knows the frame has arrived safely. On the other hand, a negative acknowledgement means that something has gone wrong, and the frame must be transmitted again.

An additional complication comes from the possibility that hardware troubles may cause a frame to vanish completely (e.g., in a noise burst). In this case, the receiver will not react at all, since it has no reason to react. It should be clear that a protocol in which the sender transmits a frame and then waits for an acknowledgement, positive or negative, will hang forever if a frame is ever lost due to, for example, malfunctioning hardware.

This possibility is dealt with by introducing timers into the data link layer. When the sender transmits a frame, it generally also starts a timer. The timer is set to expire after an interval long enough for the frame to reach the destination, be processed there, and have the acknowledgement propagate back to the sender. Normally, the frame will be correctly received and the acknowledgement will get back before the timer runs out, in which case the timer will be canceled.

However, if either the frame or the acknowledgement is lost, the timer will go off, alerting the sender to a potential problem. The obvious solution is to just transmit the frame again. However, when frames may be transmitted multiple times there is a danger that the receiver will accept the same frame two or more times, and pass it to the network layer more than once. To prevent this from happening, it is generally necessary to assign sequence numbers to outgoing frames, so that the receiver can distinguish retransmissions from originals.
The whole issue of managing the timers and sequence numbers so as to ensure that each frame is ultimately passed to the network layer at the destination exactly once, no more and no less, is an important part of the data link layer’s duties. Later in this chapter, we will study in detail how this management is done by looking at a series of increasingly sophisticated examples.

3.1.4 Flow Control

Another important design issue that occurs in the data link layer (and higher layers as well) is what to do with a sender that systematically wants to transmit frames faster than the receiver can accept them. This situation can easily occur when the sender is running on a fast (or lightly loaded) computer and the receiver is running on a slow (or heavily loaded) machine. The sender keeps pumping the frames out at a high rate until the receiver is completely swamped. Even if the transmission is error free, at a certain point the receiver will simply be unable to handle the frames as they arrive and will start to lose some. Clearly, something has to be done to prevent this situation.

Two approaches are commonly used. In the first one, feedback-based flow control, the receiver sends back information to the sender giving it permission to send more data or at least telling the sender how the receiver is doing. In the second one, rate-based flow control, the protocol has a built-in mechanism that limits the rate at which senders may transmit data, without using feedback from the receiver. In this chapter we will study feedback-based flow control schemes because rate-based schemes are never used in the data link layer. We will look at rate-based schemes in Chap. 5.

Various feedback-based flow control schemes are known, but most of them use the same basic principle. The protocol contains well-defined rules about when a sender may transmit the next frame. These rules often prohibit frames from being sent until the receiver has granted permission, either implicitly or explicitly. For example, when a connection is set up, the receiver might say: “You may send me \( n \) frames now, but after they have been sent, do not send any more until I have told you to continue.” We will examine the details shortly.

3.2 ERROR DETECTION AND CORRECTION

As we saw in Chap. 2, the telephone system has three parts: the switches, the interoffice trunks, and the local loops. The first two are now almost entirely digital in most developed countries. The local loops are still analog twisted copper pairs everywhere and will continue to be so for years due to the enormous expense of replacing them. While errors are rare on the digital part, they are still common on the local loops. Furthermore, wireless communication is becoming more common, and the error rates here are orders of magnitude worse than on the interoffice...
fiber trunks. The conclusion is: transmission errors are going to be a fact of life for many years to come.

As a result of the physical processes that generate them, errors on some media (e.g., radio) tend to come in bursts rather than singly. Having the errors come in bursts has both advantages and disadvantages over isolated single-bit errors. On the advantage side, computer data are always sent in blocks of bits. Suppose that the block size is 1000 bits, and the error rate is 0.001 per bit. If errors were independent, most blocks would contain an error. If the errors came in bursts of 100 however, only one or two blocks in 100 would be affected, on the average. The disadvantage of burst errors is that they are much harder to correct than are isolated errors.

### 3.2.1 Error-Correcting Codes

Network designers have developed two basic strategies for dealing with errors. One way is to include enough redundant information along with each block of data sent to enable the receiver to deduce what the transmitted data must have been. The other way is to include only enough redundancy to allow the receiver to deduce that an error occurred, but not which error, and have it request a retransmission. The former strategy uses error-correcting codes and the latter uses error-detecting codes. The use of error-correcting codes is often referred to as forward error correction.

Each of these techniques occupies a different ecological niche. On channels that are highly reliable, such as fiber, it is cheaper to use an error detecting code and just retransmit the occasional block found to be faulty. However, on channels that are make many errors, such as wireless links, it is better to add enough redundancy to each block for the receiver to be able to figure out what the original block was, rather than relying on a retransmission, which itself may be in error.

To understand how errors can be handled, it is necessary to look closely at what an error really is. Normally, a frame consists of \( m \) data (i.e., message) bits and \( r \) redundant, or check bits. Let the total length be \( n \) (i.e., \( n = m + r \)). An \( n \)-bit unit containing data and checkbits is often referred to as an \( n \)-bit codeword.

Given any two codewords, say, 10001001 and 10110001, it is possible to determine how many corresponding bits differ. In this case, 3 bits differ. To determine how many bits differ, just EXCLUSIVE OR the two codewords, and count the number of 1 bits in the result, for example:

\[
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\end{array}
\]

The number of bit positions in which two codewords differ is called the Hamming distance (Hamming, 1950). Its significance is that if two codewords are a Hamming distance \( d \) apart, it will require \( d \) single-bit errors to convert one into
In most data transmission applications, all $2^m$ possible data messages are legal, but due to the way the check bits are computed, not all of the $2^n$ possible codewords are used. Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list find the two codewords whose Hamming distance is minimum. This distance is the Hamming distance of the complete code.

The error-detecting and error-correcting properties of a code depend on its Hamming distance. To detect $d$ errors, you need a distance $d + 1$ code because with such a code there is no way that $d$ single-bit errors can change a valid codeword into another valid codeword. When the receiver sees an invalid codeword, it can tell that a transmission error has occurred. Similarly, to correct $d$ errors, you need a distance $2d + 1$ code because that way the legal codewords are so far apart that even with $d$ changes, the original codeword is still closer than any other codeword, so it can be uniquely determined.

As a simple example of an error-detecting code, consider a code in which a single parity bit is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). For example, when 10110101 is sent in even parity by adding a bit at the end, it becomes 101101011, whereas 10110001 becomes 101100010 with even parity. A code with a single parity bit has a distance 2, since any single-bit error produces a codeword with the wrong parity. It can be used to detect single errors.

As a simple example of an error-correcting code, consider a code with only four valid codewords:

$0000000000, \ 0000011111, \ 1111100000, \text{and} \ 1111111111$

This code has a distance 5, which means that it can correct double errors. If the codeword 0000000111 arrives, the receiver knows that the original must have been 0000011111. If, however, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly.

Imagine that we want to design a code with $m$ message bits and $r$ check bits that will allow all single errors to be corrected. Each of the $2^m$ legal messages has $n$ illegal codewords at a distance 1 from it. These are formed by systematically inverting each of the $n$ bits in the $n$-bit codeword formed from it. Thus each of the $2^m$ legal messages requires $n + 1$ bit patterns dedicated to it. Since the total number of bit patterns is $2^n$, we must have $(n + 1)2^m \leq 2^n$. Using $n = m + r$, this requirement becomes $(m + r + 1) \leq 2^r$. Given $m$, this puts a lower limit on the number of check bits needed to correct single errors.

This theoretical lower limit can, in fact, be achieved using a method due to Hamming (1950). The bits of the codeword are numbered consecutively, starting with bit 1 at the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the $m$ data bits. Each check bit forces the parity of some collection
of bits, including itself, to be even (or odd). A bit may be included in several parity computations. To see which check bits the data bit in position \( k \) contributes to, rewrite \( k \) as a sum of powers of 2. For example, \( 11 = 1 + 2 + 8 \) and \( 29 = 1 + 4 + 8 + 16 \). A bit is checked by just those check bits occurring in its expansion (e.g., bit 11 is checked by bits 1, 2, and 8).

<table>
<thead>
<tr>
<th>Char.</th>
<th>ASCII</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>100100</td>
<td>00110010000</td>
</tr>
<tr>
<td>a</td>
<td>110001</td>
<td>10111001001</td>
</tr>
<tr>
<td>m</td>
<td>110110</td>
<td>11010101011</td>
</tr>
<tr>
<td>m</td>
<td>110110</td>
<td>11010101010</td>
</tr>
<tr>
<td>l</td>
<td>110100</td>
<td>01101011001</td>
</tr>
<tr>
<td>n</td>
<td>110110</td>
<td>01101010110</td>
</tr>
<tr>
<td>g</td>
<td>110011</td>
<td>01111100111</td>
</tr>
<tr>
<td>c</td>
<td>010000</td>
<td>10011000000</td>
</tr>
<tr>
<td>o</td>
<td>110111</td>
<td>10101011111</td>
</tr>
<tr>
<td>d</td>
<td>110010</td>
<td>11111001100</td>
</tr>
<tr>
<td>e</td>
<td>110011</td>
<td>00111000101</td>
</tr>
</tbody>
</table>

Order of bit transmission

**Figure 3-7.** Use of a Hamming code to correct burst errors.

When a codeword arrives, the receiver initializes a counter to zero. It then examines each check bit, \( k \) \((k = 1, 2, 4, 8, \ldots)\) to see if it has the correct parity. If not, it adds \( k \) to the counter. If the counter is zero after all the check bits have been examined (i.e., if they were all correct), the codeword is accepted as valid. If the counter is nonzero, it contains the number of the incorrect bit. For example, if check bits 1, 2, and 8 are in error, the inverted bit is 11, because it is the only one checked by bits 1, 2, and 8. Figure 3-7 shows some 7-bit ASCII characters encoded as 11-bit codewords using a Hamming code. Remember that the data are found in bit positions 3, 5, 6, 7, 9, 10, and 11.

Hamming codes can only correct single errors. However, there is a trick that can be used to permit Hamming codes to correct burst errors. A sequence of \( k \) consecutive codewords are arranged as a matrix, one codeword per row. Normally, the data would be transmitted one codeword at a time, from left to right. To correct burst errors, the data should be transmitted one column at a time, starting with the leftmost column. When all \( k \) bits have been sent, the second column is sent, and so on, as indicated in Fig. 3-7. When the frame arrives at the receiver, the matrix is reconstructed, one column at a time. If a burst error of length \( k \) occurs, at most 1 bit in each of the \( k \) codewords will have been affected, but the Hamming code can correct one error per codeword, so the entire block can be restored. This method uses \( kr \) check bits to make blocks of \( km \) data bits immune to a single burst error of length \( k \) or less.
3.2.2 Error-Detecting Codes

Error-correcting codes are widely used over wireless links, which are notoriously noisy and error prone. Without them, it would be hard to get anything through. However, over copper wire or fiber, the error rate is much lower, so error detection and retransmission is usually more efficient there for dealing with the occasional error.

As a simple example, consider a channel on which errors are isolated and the error rate is $10^{-6}$ per bit. Let the block size be 1000 bits. To provide error correction for 1000-bit blocks, 10 check bits are needed; a megabit of data would require 10,000 check bits. To merely detect a block with a single 1-bit error, one parity bit per block will suffice. Once every 1000 blocks an extra block (1001 bits) will have to be transmitted. The total overhead for the error detection + retransmission method is only 2001 bits per megabit of data, versus 10,000 bits for a Hamming code.

If a single parity bit is added to a block and the block is badly garbled by a long burst error, the probability that the error will be detected is only 0.5, which is hardly acceptable. The odds can be improved considerably by regarding each block to be sent as a rectangular matrix $n$ bits wide and $k$ bits high. A parity bit is computed separately for each column and affixed to the matrix as the last row. The matrix is then transmitted one row at a time. When the block arrives, the receiver checks all the parity bits. If any one of them is wrong, it requests a retransmission of the block.

This method can detect a single burst of length $n$, since only 1 bit per column will be changed. A burst of length $n + 1$ will pass undetected, however, if the first bit is inverted, the last bit is inverted, and all the other bits are correct. (A burst error does not imply that all the bits are wrong; it just implies that at least the first and last are wrong.) If the block is badly garbled by a long burst or by multiple shorter bursts, the probability that any of the $n$ columns will have the correct parity, by accident, is 0.5, so the probability of a bad block being accepted when it should not be is $2^{-n}$.

Although the above scheme may sometimes be adequate, in practice, another method is in widespread use: the polynomial code (also known as a cyclic redundancy code or CRC code). Polynomial codes are based upon treating bit strings as representations of polynomials with coefficients of 0 and 1 only. A $k$-bit frame is regarded as the coefficient list for a polynomial with $k$ terms, ranging from $x^{k-1}$ to $x^0$. Such a polynomial is said to be of degree $k - 1$. The high-order (left-most) bit is the coefficient of $x^{k-1}$; the next bit is the coefficient of $x^{k-2}$, and so on. For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients 1, 1, 0, 0, 0, and 1: $x^5 + x^4 + x^0$.

Polynomial arithmetic is done modulo 2, according to the rules of algebraic field theory. There are no carries for addition or borrows for subtraction. Both
addition and subtraction are identical to EXCLUSIVE OR. For example:

\[
\begin{array}{cccc}
10011011 & 00110011 & 11110000 & 01010101 \\
+ 11001010 & + 11001101 & - 10100110 & - 10101111 \\
\hline
01010001 & 11111110 & 01010110 & 11111010 \\
\end{array}
\]

Long division is carried out the same way as it is in binary except that the subtraction is done modulo 2, as above. A divisor is said “to go into” a dividend if the dividend has as many bits as the divisor.

When the polynomial code method is employed, the sender and receiver must agree upon a **generator polynomial**, \(G(x)\), in advance. Both the high- and low-order bits of the generator must be 1. To compute the **checksum** for some frame with \(m\) bits, corresponding to the polynomial \(M(x)\), the frame must be longer than the generator polynomial. The idea is to append a checksum to the end of the frame in such a way that the polynomial represented by the checksummed frame is divisible by \(G(x)\). When the receiver gets the checksummed frame, it tries dividing it by \(G(x)\). If there is a remainder, there has been a transmission error.

The algorithm for computing the checksum is as follows:

1. Let \(r\) be the degree of \(G(x)\). Append \(r\) zero bits to the low-order end of the frame, so it now contains \(m + r\) bits and corresponds to the polynomial \(x^rM(x)\).
2. Divide the bit string corresponding to \(G(x)\) into the bit string corresponding to \(x^rM(x)\) using modulo 2 division.
3. Subtract the remainder (which is always \(r\) or fewer bits) from the bit string corresponding to \(x^rM(x)\) using modulo 2 subtraction. The result is the checksummed frame to be transmitted. Call its polynomial \(T(x)\).

Figure 3-8 illustrates the calculation for a frame 1101011011 and \(G(x) = x^4 + x + 1\).

It should be clear that \(T(x)\) is divisible (modulo 2) by \(G(x)\). In any division problem, if you diminish the dividend by the remainder, what is left over is divisible by the divisor. For example, in base 10, if you divide 210,278 by 10,941, the remainder is 2399. By subtracting off 2399 from 210,278, what is left over (207,879) is divisible by 10,941.

Now let us analyze the power of this method. What kinds of errors will be detected? Imagine that a transmission error occurs, so that instead of the bit string for \(T(x)\) arriving, \(T(x) + E(x)\) arrives. Each 1 bit in \(E(x)\) corresponds to a bit that has been inverted. If there are \(k\) 1 bits in \(E(x)\), \(k\) single-bit errors have occurred. A single burst error is characterized by an initial 1, a mixture of 0s and 1s, and a final 1, with all other bits being 0.

Upon receiving the checksummed frame, the receiver divides it by \(G(x)\); that
Frame: 1101011011
Generator: 10011
Message after appending 4 zero bits: 11010110110000

Transmitted frame: 11010110111110

Figure 3-8. Calculation of the polynomial code checksum.

is, it computes \( [T(x) + E(x)]/G(x) \). \( T(x)/G(x) \) is 0, so the result of the computation is simply \( E(x)/G(x) \). Those errors that happen to correspond to polynomials containing \( G(x) \) as a factor will slip by; all other errors will be caught.

If there has been a single-bit error, \( E(x) = x^i \), where \( i \) determines which bit is in error. If \( G(x) \) contains two or more terms, it will never divide \( E(x) \), so all single-bit errors will be detected.

If there have been two isolated single-bit errors, \( E(x) = x^i + x^j \), where \( i > j \).
Alternatively, this can be written as $E(x) = x^i(x^{i-j} + 1)$. If we assume that $G(x)$ is not divisible by $x$, a sufficient condition for all double errors to be detected is that $G(x)$ does not divide $x^k + 1$ for any $k$ up to the maximum value of $i - j$ (i.e., up to the maximum frame length). Simple, low-degree polynomials that give protection to long frames are known. For example, $x^{15} + x^{14} + 1$ will not divide $x^k + 1$ for any value of $k$ below 32,768.

If there are an odd number of bits in error, $E(X)$ contains an odd number of terms (e.g., $x^5 + x^2 + 1$, but not $x^2 + 1$). Interestingly enough, there is no polynomial with an odd number of terms that has $x + 1$ as a factor in the modulo 2 system. By making $x + 1$ a factor of $G(x)$, we can catch all errors consisting of an odd number of inverted bits.

To see that no polynomial with an odd number of terms is divisible by $x + 1$, assume that $E(x)$ has an odd number of terms and is divisible by $x + 1$. Factor $E(x)$ into $(x + 1)Q(x)$. Now evaluate $E(1) = (1 + 1)Q(1)$. Since $1 + 1 = 0$ (modulo 2), $E(1)$ must be zero. If $E(x)$ has an odd number of terms, substituting 1 for $x$ everywhere will always yield 1 as the result. Thus no polynomial with an odd number of terms is divisible by $x + 1$.

Finally, and most important, a polynomial code with $r$ check bits will detect all burst errors of length $\leq r$. A burst error of length $k$ can be represented by $x^i(x^{k-1} + \ldots + 1)$, where $i$ determines how far from the right-hand end of the received frame the burst is located. If $G(x)$ contains an $x^0$ term, it will not have $x^i$ as a factor, so if the degree of the parenthesized expression is less than the degree of $G(x)$, the remainder can never be zero.

If the burst length is $r + 1$, the remainder of the division by $G(x)$ will be zero if and only if the burst is identical to $G(x)$. By definition of a burst, the first and last bits must be 1, so whether it matches depends on the $r - 1$ intermediate bits. If all combinations are regarded as equally likely, the probability of such an incorrect frame being accepted as valid is $\frac{1}{2^{r-1}}$.

It can also be shown that when an error burst longer than $r + 1$ bits occurs, or several shorter bursts occur, the probability of a bad frame getting through unnoticed is $\frac{1}{2^r}$ assuming that all bit patterns are equally likely.

Certain polynomials have become international standards. The one used in IEEE 802 is

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

It has the property that it detects all bursts of length 32 or less and all bursts affecting an odd number of bits, as well as other desirable properties.

Although the calculation required to compute the checksum may seem complicated, Peterson and Brown (1961) have shown that a simple shift register circuit can be constructed to compute and verify the checksums in hardware. In practice, this hardware is nearly always used.

For decades, it has been assumed that frames to be checksummed contain random bits. All analyses of checksum algorithms have been made under this
assumption. Inspection of real data has shown this assumption to be quite wrong. As a consequence, under some circumstances, undetected errors are much more common than had been previously thought (Partridge et al., 1995).

3.3 ELEMENTARY DATA LINK PROTOCOLS

To introduce the subject of protocols, we will begin by looking at three protocols of increasing complexity. For interested readers, a simulator for these and subsequent protocols is available via the WWW (see the preface). Before we look at the protocols, it is useful to make explicit some of the assumptions underlying the model of communication. To start with, we assume that in the physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth. In many cases, the physical and data link layer processes will be running on a processor inside a special network I/O chip and the network layer on the main CPU, but other implementations are also possible (e.g., three processes inside a single I/O chip; the physical and data link layers as procedures called by the network layer process, and so on). In any event, treating the three layers as separate processes makes the discussion conceptually cleaner and also serves to emphasize the independence of the layers.

Another key assumption is that machine A wants to send a long stream of data to machine B using a reliable, connection-oriented service. Later, we will consider the case where B also wants to send data to A simultaneously. A is assumed to have an infinite supply of data ready to send and never has to wait for data to be produced. Instead, when A’s data link layer asks for data, the network layer is always able to comply immediately. (This restriction, too, will be dropped later.)

We also assume that machines do not crash. That is, these protocols deal with communication errors, but not the problems caused by computers crashing and rebooting.

As far as the data link layer is concerned, the packet passed across the interface to it from the network layer is pure data, every bit of which is to be delivered to the destination’s network layer. The fact that the destination’s network layer may interpret part of the packet as a header is of no concern to the data link layer.

When the data link layer accepts a packet, it encapsulates the packet in a frame by adding a data link header and trailer to it (see Fig. 3-1). Thus a frame consists of an embedded packet, some control information (in the header), and a checksum (in the trailer). The frame is then transmitted to the other data link layer. We will assume that there exist suitable library procedures to_physical_layer to send a frame and from_physical_layer to receive a frame. The transmitting hardware computes and appends the checksum (thus creating the trailer), so that the datalink layer software need not worry about it. The polynomial algorithm discussed earlier in this chapter might be used, for example.

Initially, the receiver has nothing to do. It just sits around waiting for
something to happen. In the example protocols of this chapter we indicate that the data link layer is waiting for something to happen by the procedure call \texttt{wait\_for\_event(event)}. This procedure only returns when something has happened (e.g., a frame has arrived). Upon return, the variable \texttt{event} tells what happened. The set of possible events differs for the various protocols to be described and will be defined separately for each protocol. Note that in a more realistic situation, the data link layer will not sit in a tight loop waiting for an event, as we have suggested, but will receive an interrupt, which will cause it to stop whatever it was doing and go handle the incoming frame. Nevertheless, for simplicity we will ignore all the details of parallel activity within the data link layer and assume that it is dedicated full time to handling just our one channel.

When a frame arrives at the receiver, the hardware computes the checksum. If the checksum is incorrect (i.e., there was a transmission error), the data link layer is so informed (\texttt{event = cksum\_err}). If the inbound frame arrived undamaged, the data link layer is also informed (\texttt{event = frame\_arrival}), so it can acquire the frame for inspection using \texttt{from\_physical\_layer}. As soon as the receiving data link layer has acquired an undamaged frame, it checks the control information in the header, and if everything is all right, the packet portion is passed to the network layer. Under no circumstances is a frame header ever given to a network layer.

There is a good reason why the network layer must never be given any part of the frame header: to keep the network and data link protocols completely separate. As long as the network layer knows nothing at all about the data link protocol or the frame format, these things can be changed without requiring changes to the network layer’s software. Providing a rigid interface between network layer and data link layer greatly simplifies the software design, because communication protocols in different layers can evolve independently.

Figure 3-9 shows some declarations (in C) common to many of the protocols to be discussed later. Five data structures are defined there: \texttt{boolean}, \texttt{seq\_nr}, \texttt{packet}, \texttt{frame\_kind}, and \texttt{frame}. A \texttt{boolean} is an enumerated type and can take on the values \texttt{true} and \texttt{false}. A \texttt{seq\_nr} is a small integer used to number the frames, so we can tell them apart. These sequence numbers run from 0 up to and including \texttt{MAX\_SEQ}, which is defined in each protocol needing it. A \texttt{packet} is the unit of information exchanged between the network layer and the data link layer on the same machine, or between network layer peers. In our model it always contains \texttt{MAX\_PKT} bytes, but more realistically it would be of variable length.

A \texttt{frame} is composed of four fields: \texttt{kind}, \texttt{seq}, \texttt{ack}, and \texttt{info}, the first three of which contain control information, and the last of which may contain actual data to be transferred. These control fields are collectively called the \texttt{frame\_header}. The \texttt{kind} field tells whether or not there are any data in the frame, because some of the protocols distinguish frames containing exclusively control information from those containing data as well. The \texttt{seq} and \texttt{ack} fields are used for sequence numbers and acknowledgements, respectively; their use will be described in more
detail later. The *info* field of a data frame contains a single packet; the *info* field of a control frame is not used. A more realistic implementation would use a variable-length *info* field, omitting it altogether for control frames.

Again, it is important to realize the relationship between a packet and a frame. The network layer builds a packet by taking a message from the transport layer and adding the network layer header to it. This packet is passed to the data link layer for inclusion in the *info* field of an outgoing frame. When the frame arrives at the destination, the data link layer extracts the packet from the frame and passes the packet to the network layer. In this manner, the network layer can act as though machines can exchange packets directly.

A number of procedures are also listed in Fig. 3-9. These are library routines whose details are implementation-dependent and whose inner workings will not concern us further here. The procedure `wait_for_event` sits in a tight loop waiting for something to happen, as mentioned earlier. The procedures `to_network_layer` and `from_network_layer` are used by the data link layer to pass packets to the network layer and accept packets from the network layer, respectively. Note that `from_physical_layer` and `to_physical_layer` are used for passing frames between the data link and physical layers, whereas the procedures `to_network_layer` and `from_network_layer` are used for passing packets between the data link layer and network layer. In other words, `to_network_layer` and `from_network_layer` deal with the interface between layers 2 and 3, whereas `from_physical_layer` and `to_physical_layer` deal with the interface between layers 1 and 2.

In most of the protocols we assume an unreliable channel that loses entire frames upon occasion. To be able to recover from such calamities, the sending data link layer must start an internal timer or clock whenever it sends a frame. If no reply has been received within a certain predetermined time interval, the clock times out and the data link layer receives an interrupt signal.

In our protocols this is handled by allowing the procedure `wait_for_event` to return `event = timeout`. The procedures `start_timer` and `stop_timer` are used to turn the timer on and off, respectively. Timeouts are possible only when the timer is running. It is explicitly permitted to call `start_timer` while the timer is running; such a call simply resets the clock to cause the next timeout after a full timer interval has elapsed (unless it is reset or turned off in the meanwhile).

The procedures `start_ack_timer` and `stop_ack_timer` are used to control an auxiliary timer used to generate acknowledgements under certain conditions.

The procedures `enable_network_layer` and `disable_network_layer` are used in the more sophisticated protocols, where we no longer assume that the network layer always has packets to send. When the data link layer enables the network layer, the network layer is then permitted to interrupt when it has a packet to be sent. We indicate this with `event = network_layer_ready`. When a network layer is disabled, it may not cause such events. By being careful about when it enables and disables its network layer, the data link layer can prevent the network layer from swamping it with packets for which it has no buffer space.
SEC. 3.3 ELEMENTARY DATA LINK PROTOCOLS

#define MAX_PKT 1024 /* determines packet size in bytes */
typedef enum {false, true} boolean; /* boolean type */
typedef unsigned int seq nr; /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet; /* packet definition */
typedef enum {data, ack, nak} frame kind; /* frame kind definition */
typedef struct {
    frame kind kind; /* what kind of a frame is it? */
    seq nr seq; /* sequence number */
    seq nr ack; /* acknowledgement number */
    packet info; /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event type *event);
/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);
/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);
/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);
/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);
/* Start the clock running and enable the timeout event. */
void start_timer(seq nr k);
/* Stop the clock and disable the timeout event. */
void stop_timer(seq nr k);
/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);
/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);
/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);
/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);
/* Macro inc is expanded in-line: Increment k circularly. */
define inc(k) if (k < MAX_seq) k = k + 1; else k = 0

Figure 3-9. Some definitions needed in the protocols to follow. These definitions are located in the file protocol.h.
Frame sequence numbers are always in the range 0 to $MAX\_SEQ$ (inclusive), where $MAX\_SEQ$ is different for the different protocols. It is frequently necessary to advance a sequence number by 1 circularly (i.e., $MAX\_SEQ$ is followed by 0). The macro $inc$ performs this incrementing. It has been defined as a macro because it is used in-line within the critical path. As we will see later in this book, the factor limiting network performance is often protocol processing, so defining simple operations like this as macros does not affect the readability of the code, but does improve performance. Also, since $MAX\_SEQ$ will have different values in different protocols, by making it a macro, it becomes possible to include all the protocols in the same binary without conflict. This ability is useful for the simulator.

The declarations of Fig. 3-9 are part of each of the protocols to follow. To save space and to provide a convenient reference, they have been extracted and listed together, but conceptually they should be merged with the protocols themselves. In C, this merging is done by putting the definitions in a special header file, in this case $protocol.h$, and using the #include facility of the C preprocessor to include them in the protocol files.

### 3.3.1 An Unrestricted Simplex Protocol

As an initial example we will consider a protocol that is as simple as can be. Data are transmitted in one direction only. Both the transmitting and receiving network layers are always ready. Processing time can be ignored. Infinite buffer space is available. And best of all, the communication channel between the data link layers never damages or loses frames. This thoroughly unrealistic protocol, which we will nickname “utopia,” is shown in Fig. 3-10.

The protocol consists of two distinct procedures, a sender and a receiver. The sender runs in the data link layer of the source machine, and the receiver runs in the data link layer of the destination machine. No sequence numbers or acknowledgements are used here, so $MAX\_SEQ$ is not needed. The only event type possible is $frame\_arrival$ (i.e., the arrival of an undamaged frame).

The sender is in an infinite while loop just pumping data out onto the line as fast as it can. The body of the loop consists of three actions: go fetch a packet from the (always obliging) network layer, construct an outbound frame using the variable $s$, and send the frame on its way. Only the $info$ field of the frame is used by this protocol, because the other fields have to do with error and flow control, and there are no errors or flow control restrictions here.

The receiver is equally simple. Initially, it waits for something to happen, the only possibility being the arrival of an undamaged frame. Eventually, the frame arrives and the procedure $wait\_for\_event$ returns, with $event$ set to $frame\_arrival$ (which is ignored anyway). The call to $from\_physical\_layer$ removes the newly arrived frame from the hardware buffer and puts it in the variable $r$. Finally, the data portion is passed on to the network layer and the data link layer settles back
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
sender to receiver. The communication channel is assumed to be error free,
and the receiver is assumed to be able to process all the input infinitely quickly.
Consequently, the sender just sits in a loop pumping data out onto the line as
fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                  /* buffer for an outbound frame */
  packet buffer;            /* buffer for an outbound packet */

  while (true) {
    from network_layer(&buffer); /* go get something to send */
    s.info = buffer;            /* copy it into s for transmission */
    to physical_layer(&s);     /* send it on its way */
  }
}

void receiver1(void)
{
  frame r;
  event_type event;          /* filled in by wait, but not used here */

  while (true) {
    wait_for_event(&event);   /* only possibility is frame_arrival */
    from physical_layer(&r);  /* go get the inbound frame */
    to network_layer(&r.info); /* pass the data to the network layer */
  }
}

Figure 3-10. An unrestricted simplex protocol.

to wait for the next frame, effectively suspending itself until the frame arrives.

3.3.2 A Simplex Stop-and-Wait Protocol

Now we will drop the most unrealistic restriction used in protocol 1: the ability of the receiving network layer to process incoming data infinitely quickly (or equivalently, the presence in the receiving data link layer of an infinite amount of
buffer space in which to store all incoming frames while they are waiting their respective turns). The communication channel is still assumed to be error free however, and the data traffic is still simplex.

The main problem we have to deal with here is how to prevent the sender from flooding the receiver with data faster than the latter is able to process them. In essence, if the receiver requires a time $\Delta t$ to execute from physical layer plus to network layer, the sender must transmit at an average rate less than one frame per time $\Delta t$. Moreover, if we assume that there is no automatic buffering and queueing done within the receiver’s hardware, the sender must never transmit a new frame until the old one has been fetched by from physical layer, lest the new one overwrite the old one.

In certain restricted circumstances (e.g., synchronous transmission and a receiving data link layer fully dedicated to processing the one input line), it might be possible for the sender to simply insert a delay into protocol 1 to slow it down sufficiently to keep from swamping the receiver. However, more usually, each data link layer will have several lines to attend to, and the time interval between a frame arriving and its being processed may vary considerably. If the network designers can calculate the worst-case behavior of the receiver, they can program the sender to transmit so slowly that even if every frame suffers the maximum delay, there will be no overruns. The trouble with this approach is that it is too conservative. It leads to a bandwidth utilization that is far below the optimum, unless the best and worst cases are almost the same (i.e., the variation in the data link layer’s reaction time is small).

A more general solution to this dilemma is to have the receiver provide feedback to the sender. After having passed a packet to its network layer, the receiver sends a little dummy frame back to the sender which, in effect, gives the sender permission to transmit the next frame. After having sent a frame, the sender is required by the protocol to bide its time until the little dummy (i.e., acknowledgement) frame arrives. Using feedback from the receiver to let the sender know when it may send more data is an example of the flow control mentioned earlier.

Protocols in which the sender sends one frame and then waits for an acknowledgement before proceeding are called stop-and-wait. Figure 3-11 gives an example of a simplex stop-and-wait protocol.

Although data traffic in this example is simplex, going only from the sender to the receiver, frames do travel in both directions. Consequently, the communication channel between the two data link layers needs to be capable of bidirectional information transfer. However, this protocol entails a strict alternation of flow: first the sender sends a frame, then the receiver sends a frame, then the sender sends another frame, then the receiver sends another one, and so on. A half-duplex physical channel would suffice here.

As in protocol 1, the sender starts out by fetching a packet from the network layer, using it to construct a frame and sending it on its way. Only now, unlike in protocol 1, the sender must wait until an acknowledgement frame arrives before
Protocol 2 (stop-and-wait) also provides for a one-directional flow of data from 
sender to receiver. The communication channel is once again assumed to be error 
free, as in protocol 1. However, this time, the receiver has only a finite buffer 
capacity and a finite processing speed, so the protocol must explicitly prevent 
the sender from flooding the receiver with data faster than it can be handled. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
  frame s; /* buffer for an outbound frame */
  packet buffer; /* buffer for an outbound packet */
  event_type event; /* frame_arrival is the only possibility */

  while (true) {
    from_network_layer(&buffer); /* go get something to send */
    s.info = buffer; /* copy it into s for transmission */
    to_physical_layer(&s); /* bye bye little frame */
    wait_for_event(&event); /* do not proceed until given the go ahead */
  }
}

void receiver2(void)
{
  frame r, s; /* buffers for frames */
  event_type event; /* frame_arrival is the only possibility */

  while (true) {
    wait_for_event(&event); /* only possibility is frame_arrival */
    from_physical_layer(&r); /* go get the inbound frame */
    to_network_layer(&r.info); /* pass the data to the network layer */
    to_physical_layer(&s); /* send a dummy frame to awaken sender */
  }
}

Figure 3-11. A simplex stop-and-wait protocol.

looping back and fetching the next packet from the network layer. The sending 
data link layer need not even inspect the incoming frame: there is only one possi-
bility.

The only difference between receiver1 and receiver2 is that after delivering a 
packet to the network layer, receiver2 sends an acknowledgement frame back to 
the sender before entering the wait loop again. Because only the arrival of the 
frame back at the sender is important, not its contents, the receiver need not put 
any particular information in it.
3.3.3 A Simplex Protocol for a Noisy Channel

Now let us consider the normal situation of a communication channel that makes errors. Frames may be either damaged or lost completely. However, we assume that if a frame is damaged in transit, the receiver hardware will detect this when it computes the checksum. If the frame is damaged in such a way that the checksum is nevertheless correct, an exceedingly unlikely occurrence, this protocol (and all other protocols) can fail (i.e., deliver an incorrect packet to the network layer).

At first glance it might seem that a variation of protocol 2 would work: adding a timer. The sender could send a frame, but the receiver would only send an acknowledgement frame if the data were correctly received. If a damaged frame arrived at the receiver, it would be discarded. After a while the sender would time out and send the frame again. This process would be repeated until the frame finally arrived intact.

The above scheme has a fatal flaw in it. Think about the problem and try to discover what might go wrong before reading further.

To see what might go wrong, remember that it is the task of the data link layer processes to provide error free, transparent communication between network layer processes. The network layer on machine A gives a series of packets to its data link layer, which must ensure that an identical series of packets are delivered to the network layer on machine B by its data link layer. In particular, the network layer on B has no way of knowing that a packet has been lost or duplicated, so the data link layer must guarantee that no combination of transmission errors, no matter how unlikely, can cause a duplicate packet to be delivered to a network layer.

Consider the following scenario:

1. The network layer on A gives packet 1 to its data link layer. The packet is correctly received at B and passed to the network layer on B. B sends an acknowledgement frame back to A.

2. The acknowledgement frame gets lost completely. It just never arrives at all. Life would be a great deal simpler if the channel only mangled and lost data frames and not control frames, but sad to say, the channel is not very discriminating.

3. The data link layer on A eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.

4. The duplicate frame also arrives at the data link layer on B perfectly and is unwittingly passed to the network layer there. If A is sending a file to B, part of the file will be duplicated (i.e., the copy of the file made by B will be incorrect and the error will not have been
Clearly, what is needed is some way for the receiver to be able to distinguish a frame that it is seeing for the first time from a retransmission. The obvious way to achieve this is to have the sender put a sequence number in the header of each frame it sends. Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.

Since a small frame header is desirable, the question arises: What is the minimum number of bits needed for the sequence number? The only ambiguity in this protocol is between a frame, \( m \), and its direct successor, \( m + 1 \). If frame \( m \) is lost or damaged, the receiver will not acknowledge it, so the sender will keep trying to send it. Once it has been correctly received, the receiver will send an acknowledgement back to the sender. It is here that the potential trouble crops up. Depending upon whether the acknowledgement frame gets back to the sender correctly or not, the sender may try to send \( m \) or \( m + 1 \).

The event that triggers the sender to start sending \( m + 2 \) is the arrival of an acknowledgement for \( m + 1 \). But this implies that \( m \) has been correctly received, and furthermore that its acknowledgement has also been correctly received by the sender (otherwise, the sender would not have begun with \( m + 1 \), let alone \( m + 2 \)). As a consequence, the only ambiguity is between a frame and its immediate predecessor or successor, not between the predecessor and successor themselves.

A 1-bit sequence number (0 or 1) is therefore sufficient. At each instant of time, the receiver expects a particular sequence number next. Any arriving frame containing the wrong sequence number is rejected as a duplicate. When a frame containing the correct sequence number arrives, it is accepted, passed to the network layer, and the expected sequence number is incremented modulo 2 (i.e., 0 becomes 1 and 1 becomes 0).

An example of this kind of protocol is shown in Fig. 3-12. Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called PAR (Positive Acknowledgement with Retransmission) or ARQ (Automatic Repeat reQuest). Like protocol 2, this one also transmits data only in one direction.

Protocol 3 differs from its predecessors in that both sender and receiver have a variable whose value is remembered while the data link layer is in the wait state. The sender remembers the sequence number of the next frame to send in `next_frame_to_send`; the receiver remembers the sequence number of the next frame expected in `frame_expected`. Each protocol has a short initialization phase before entering the infinite loop.

After transmitting a frame, the sender starts the timer running. If it was already running, it will be reset to allow another full timer interval. The time interval should be chosen to allow enough time for the frame to get to the receiver, for the receiver to process it in the worst case, and for the acknowledgement frame to propagate back to the sender. Only when that time interval has elapsed...
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */
#define MAX_SEQ 1 /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send; /* seq number of next outgoing frame */
  frame s; /* scratch variable */
  packet buffer; /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0; /* initialize outbound sequence numbers */
  from_network_layer(&buffer); /* fetch first packet */

  while (true) {
    s.info = buffer; /* construct a frame for transmission */
    s.seq = next_frame_to_send; /* insert sequence number in frame */
    to_physical_layer(&s); /* send it on its way */
    start_timer(s.seq); /* if answer takes too long, time out */
    wait_for_event(&event); /* frame arrival, cksum err, timeout */
    if (event == frame_arrival) {
      from_physical_layer(&s);
      if (s.ack == next_frame_to_send) {
        stop_timer(s.ack); /* turn the timer off */
        inc(next_frame_to_send); /* invert next frame to send */
      }
    }
  }
}

void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  from_network_layer(&buffer); /* fetch first packet */

  while (true) {
    wait_for_event(&event); /* possibilities: frame_arrival, cksum_err */
    if (event == frame_arrival) { /* a valid frame has arrived. */
      from_physical_layer(&r);
      if (r.seq == frame_expected) { /* this is what we have been waiting for. */
        inc(frame_expected); /* next time expect the other sequence nr */
        send acknowledgement */
      }
    }"
is it safe to assume that either the transmitted frame or its acknowledgement has been lost, and to send a duplicate. If the timeout interval is set too short, the sender will transmit unnecessary frames. While these extra frames will not affect the correctness of the protocol, they will hurt performance.

After transmitting a frame and starting the timer, the sender waits for something exciting to happen. There are three possibilities: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires. If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet. It also advances the sequence number. If a damaged frame arrives or no frame at all arrives, neither the buffer nor the sequence number are changed, so that a duplicate can be sent.

When a valid frame arrives at the receiver, its sequence number is checked to see if it is a duplicate. If not, it is accepted, passed to the network layer, and an acknowledgement generated. Duplicates and damaged frames are not passed to the network layer.
3.4 SLIDING WINDOW PROTOCOLS

In the previous protocols, data frames were transmitted in one direction only. In most practical situations, there is a need for transmitting data in both directions. One way of achieving full-duplex data transmission is to have two separate communication channels and use each one for simplex data traffic (in different directions). If this is done, we have two separate physical circuits, each with a “forward” channel (for data) and a “reverse” channel (for acknowledgements). In both cases the bandwidth of the reverse channel is almost entirely wasted. In effect, the user is paying for two circuits but using only the capacity of one.

A better idea is to use the same circuit for data in both directions. After all, in protocols 2 and 3 it was already being used to transmit frames both ways, and the reverse channel has the same capacity as the forward channel. In this model the data frames from A to B are intermixed with the acknowledgement frames from A to B. By looking at the kind field in the header of an incoming frame, the receiver can tell whether the frame is data or acknowledgement.

Although interleaving data and control frames on the same circuit is an improvement over having two separate physical circuits, yet another improvement is possible. When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the network layer passes it the next packet. The acknowledgement is attached to the outgoing data frame (using the ack field in the frame header). In effect, the acknowledgement gets a free ride on the next outgoing data frame. The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame is known as piggybacking.

The principal advantage of using piggybacking over having distinct acknowledgement frames is a better use of the available channel bandwidth. The ack field in the frame header costs only a few bits, whereas a separate frame would need a header, the acknowledgement, and a checksum. In addition, fewer frames sent means fewer “frame arrived” interrupts, and perhaps fewer buffers in the receiver, depending on how the receiver’s software is organized. In the next protocol to be examined, the piggyback field costs only 1 bit in the frame header. It rarely costs more than a few bits.

However, piggybacking introduces a complication not present with separate acknowledgements. How long should the data link layer wait for a packet onto which to piggyback the acknowledgement? If the data link layer waits longer than the sender’s timeout period, the frame will be retransmitted, defeating the whole purpose of having acknowledgements. If the data link layer were an oracle and could foretell the future, it would know when the next network layer packet was going to come in, and could decide either to wait for it or send a separate acknowledgement immediately, depending on how long the projected wait was going to be. Of course, the data link layer cannot foretell the future, so it must resort to some ad hoc scheme, such as waiting a fixed number of milliseconds. If
a new packet arrives quickly, the acknowledgement is piggybacked onto it; otherwise, if no new packet has arrived by the end of this time period, the data link layer just sends a separate acknowledgement frame.

The next three protocols are bidirectional protocols that belong to a class called **sliding window** protocols. The three differ among themselves in terms of efficiency, complexity, and buffer requirements, as discussed later. In these, as in all sliding window protocols, each outbound frame contains a sequence number, ranging from 0 up to some maximum. The maximum is usually $2^n - 1$ so the sequence number fits exactly in an $n$-bit field. The stop-and-wait sliding window protocol uses $n = 1$, restricting the sequence numbers to 0 and 1, but more sophisticated versions can use arbitrary $n$.

The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send. These frames are said to fall within the **sending window**. Similarly, the receiver also maintains a **receiving window** corresponding to the set of frames it is permitted to accept. The sender’s window and the receiver’s window need not have the same lower and upper limits, or even have the same size. In some protocols they are fixed in size, but in others they can grow or shrink as frames are sent and received.

Although these protocols give the data link layer more freedom about the order in which it may send and receive frames, we have most emphatically not dropped the requirement that the protocol must deliver packets to the destination network layer in the same order that they were passed to the data link layer on the sending machine. Nor have we changed the requirement that the physical communication channel is “wire-like,” that is, it must deliver all frames in the order sent.

The sequence numbers within the sender’s window represent frames sent but as yet not acknowledged. Whenever a new packet arrives from the network layer, it is given the next highest sequence number, and the upper edge of the window is advanced by one. When an acknowledgement comes in, the lower edge is advanced by one. In this way the window continuously maintains a list of unacknowledged frames.

Since frames currently within the sender’s window may ultimately be lost or damaged in transit, the sender must keep all these frames in its memory for possible retransmission. Thus if the maximum window size is $n$, the sender needs $n$ buffers to hold the unacknowledged frames. If the window ever grows to its maximum size, the sending data link layer must forcibly shut off the network layer until another buffer becomes free.

The receiving data link layer’s window corresponds to the frames it may accept. Any frame falling outside the window is discarded without comment. When a frame whose sequence number is equal to the lower edge of the window is received, it is passed to the network layer, an acknowledgement is generated, and the window is rotated by one. Unlike the sender’s window, the receiver’s
window always remains at its initial size. Note that a window size of 1 means that the data link layer only accepts frames in order, but for larger windows this is not so. The network layer, in contrast, is always fed data in the proper order, regardless of the data link layer’s window size.

Figure 3-13 shows an example with a maximum window size of 1. Initially, no frames are outstanding, so the lower and upper edges of the sender’s window are equal, but as time goes on, the situation progresses as shown.

3.4.1 A One-Bit Sliding Window Protocol

Before tackling the general case, let us first examine a sliding window protocol with a maximum window size of 1. Such a protocol uses stop-and-wait, since the sender transmits a frame and waits for its acknowledgement before sending the next one.

Figure 3-14 depicts such a protocol. Like the others, it starts out by defining some variables. Next_frame_to_send tells which frame the sender is trying to send. Similarly, frame_expected tells which frame the receiver is expecting. In both cases, 0 and 1 are the only possibilities.

Under normal circumstances, one of the two data link layers goes first. In other words, only one of the data link layer programs should contain the to_physical_layer and start_timer procedure calls outside the main loop. In the
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1     /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void protocol4 (void)
{
    seq_nr next_frame_to_send; /* 0 or 1 only */
    seq_nr frame_expected;    /* 0 or 1 only */
    frame r, s;               /* scratch variables */
    packet buffer;            /* current packet being sent */
    event_type event;

    next_frame_to_send = 0;  /* next frame on the outbound stream */
    frame_expected = 0;      /* frame expected next */
    from_network_layer(&buffer); /* fetch a packet from the network layer */
    s.info = buffer;         /* prepare to send the initial frame */
    s.seq = next_frame_to_send; /* insert sequence number into frame */
    s.ack = 1 - frame_expected; /* piggybacked ack */
    to_physical_layer(&s);   /* transmit the frame */
    start_timer(s.seq);      /* start the timer running */

    while (true) {
        wait_for_event(&event);   /* frame_arrival, cksum_err, or timeout */
        if (event == frame_arrival) {
            from_physical_layer(&r); /* a frame has arrived undamaged. */
            if (r.seq == frame_expected) { /* handle inbound frame stream. */
                to_network_layer(&r.info);   /* pass packet to network layer */
                inc(frame_expected);        /* invert seq number expected next */
            }
        }
        if (r.ack == next_frame_to_send) { /* handle outbound frame stream. */
            stop_timer(r.ack);            /* turn the timer off */
            from_network_layer(&buffer); /* fetch new pkt from network layer */
            inc(next_frame_to_send);      /* invert sender's sequence number */
        }
    }

    s.info = buffer;            /* construct outbound frame */
    s.seq = next_frame_to_send; /* insert sequence number into it */
    s.ack = 1 - frame_expected; /* seq number of last received frame */
    to_physical_layer(&s);      /* transmit a frame */
    start_timer(s.seq);         /* start the timer running */
}

Figure 3-14. A 1-bit sliding window protocol.
event both data link layers start off simultaneously, a peculiar situation arises, which is discussed later. The starting machine fetches the first packet from its network layer, builds a frame from it, and sends it. When this (or any) frame arrives, the receiving data link layer checks to see if it is a duplicate, just as in protocol 3. If the frame is the one expected, it is passed to the network layer and the receiver’s window is slid up.

The acknowledgement field contains the number of the last frame received without error. If this number agrees with the sequence number of the frame the sender is trying to send, the sender knows it is done with the frame stored in buffer and can fetch the next packet from its network layer. If the sequence number disagrees, it must continue trying to send the same frame. Whenever a frame is received, a frame is also sent back.

Now let us examine protocol 4 to see how resilient it is to pathological scenarios. Assume that computer A is trying to send its frame 0 to computer B and that B is trying to send its frame 0 to A. Suppose that A sends a frame to B, but A’s timeout interval is a little too short. Consequently, A may time out repeatedly, sending a series of identical frames, all with seq = 0 and ack = 1.

When the first valid frame arrives at computer B, it will be accepted, and frame expected will be set to 1. All the subsequent frames will be rejected, because B is now expecting frames with sequence number 1, not 0. Furthermore, since all the duplicates have ack = 1 and B is still waiting for an acknowledgement of 0, B will not fetch a new packet from its network layer.

After every rejected duplicate comes in, B sends A a frame containing seq = 0 and ack = 0. Eventually, one of these arrives correctly at A, causing A to begin sending the next packet. No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, or to skip a packet, or to get into a deadlock.

However, a peculiar situation arises if both sides simultaneously send an initial packet. This synchronization difficulty is illustrated by Fig. 3-15. In part (a), the normal operation of the protocol is shown. In (b) the peculiarity is illustrated. If B waits for A’s first frame before sending one of its own, the sequence is as shown in (a), and every frame is accepted. However, if A and B simultaneously initiate communication, their first frames cross, and the data link layers then get into situation (b). In (a) each frame arrival brings a new packet for the network layer; there are no duplicates. In (b) half of the frames contain duplicates, even though there are no transmission errors. Similar situations can occur as a result of premature timeouts, even when one side clearly starts first. In fact, if multiple premature timeouts occur, frames may be sent three or more times.
3.4.2 A Protocol Using Go Back N

Until now we have made the tacit assumption that the transmission time required for a frame to arrive at the receiver plus the transmission time for the acknowledgement to come back is negligible. Sometimes this assumption is clearly false. In these situations the long round-trip time can have important implications for the efficiency of the bandwidth utilization. As an example, consider a 50-kbps satellite channel with a 500-msec round-trip propagation delay. Let us imagine trying to use protocol 4 to send 1000-bit frames via the satellite. At $t = 0$ the sender starts sending the first frame. At $t = 20$ msec the frame has been completely sent. Not until $t = 270$ msec has the frame fully arrived at the receiver, and not until $t = 520$ msec has the acknowledgement arrived back at the sender, under the best of circumstances (no waiting in the receiver and a short acknowledgement frame). This means that the sender was blocked during 500/520 or 96 percent of the time. In other words, only 4 percent of the available bandwidth was used. Clearly, the combination of a long transit time, high bandwidth, and short frame length is disastrous in terms of efficiency.

The problem described above can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame. If we relax that restriction, much better efficiency can be achieved. Basically the solution lies in allowing the sender to transmit up to $w$ frames before blocking, instead of just 1. With an appropriate choice of $w$ the sender will be able to continuously transmit frames for a time equal to the round-trip transit time without
filling up the window. In the example above, \( w \) should be at least 26. The sender begins sending frame 0 as before. By the time it has finished sending 26 frames, at \( t = 520 \), the acknowledgement for frame 0 will have just arrived. Thereafter, acknowledgements will arrive every 20 msec, so the sender always gets permission to continue just when it needs it. At all times, 25 or 26 unacknowledged frames are outstanding. Put in other terms, the sender’s maximum window size is 26.

The need for a large window on the sending side occurs whenever the product of bandwidth \( \times \) roundtrip-delay is large. If the bandwidth is high, even for a moderate delay, the sender will exhaust its window quickly unless it has a large window. If the delay is high (e.g., a geostationary satellite channel), the sender will exhaust its window even for a moderate bandwidth. The product of these two factors basically tells that the capacity of the pipe is, and the sender needs the ability to fill it without stopping to operate at peak efficiency.

This technique is known as **pipelining**. If the channel capacity is \( b \) bits/sec,
the frame size $l$ bits, and the round-trip propagation time $R$ sec, the time required to transmit a single frame is $l/b$ sec. After the last bit of a data frame has been sent, there is a delay of $R/2$ before that bit arrives at the receiver, and another delay of at least $R/2$ for the acknowledgement to come back, for a total delay of $R$. In stop-and-wait the line is busy for $l/b$ and idle for $R$, giving

$$\text{line utilization} = \frac{l}{l + bR}$$

If $l < bR$ the efficiency will be less than 50 percent. Since there is always a nonzero delay for the acknowledgement to propagate back, in principle pipelining can be used to keep the line busy during this interval, but if the interval is small, the additional complexity is not worth the trouble.

Pipelining frames over an unreliable communication channel raises some serious issues. First, what happens if a frame in the middle of a long stream is damaged or lost? Large numbers of succeeding frames will arrive at the receiver before the sender even finds out that anything is wrong. When a damaged frame arrives at the receiver, it obviously should be discarded, but what should the receiver do with all the correct frames following it? Remember that the receiving data link layer is obligated to hand packets to the network layer in sequence.

There are two basic approaches to dealing with errors in the presence of pipelining. One way, called go back $n$, is for the receiver simply to discard all subsequent frames, sending no acknowledgements for the discarded frames. This strategy corresponds to a receive window of size 1. In other words, the data link layer refuses to accept any frame except the next one it must give to the network layer. If the sender’s window fills up before the timer runs out, the pipeline will begin to empty. Eventually, the sender will time out and retransmit all unacknowledged frames in order, starting with the damaged or lost one. This approach can waste a lot of bandwidth if the error rate is high.

In Fig. 3-16(a) we see go back $n$ for the case where the receiver’s window is large. Frames 0 and 1 are correctly received and acknowledged. Frame 2, however, is damaged or lost. The sender, unaware of this problem, continues to send frames until the timer for frame 2 expires. Then it backs up to frame 2 and starts all over with it, sending 2, 3, 4, etc. all over again.

The other general strategy for handling errors when frames are pipelined is called selective repeat. When it is used, if a bad frame is received it is discarded, but good frames received after it are buffered. When the sender times out, only the oldest unacknowledged frame is retransmitted. At that frame arrives correctly, the receiver can deliver all the frames it has buffered to the network layer in sequence. Selective repeat is often combined with having the receiver send a negative acknowledgement (NAK) when it detects an error, for example when it receives a checksum error or a frame out of sequence. NAKs stimulate retransmission before the corresponding timer expires and thus improve performance.

In Fig. 3-16(b), frames 0 and 1 are again correctly received and acknowledged
and frame 2 is lost. When frame 3 arrives at the receiver, the data link layer there notices that it has missed a frame, so it sends back a NAK for 2 but buffers 3. When frames 4 and 5 arrive, they, too, are buffered by the data link layer instead of being passed to the network layer. Eventually, the NAK 2 gets back to the sender, which immediately resends frame 2. When that arrives, the data link layer now has 2, 3, 4, and 5 and can pass all of them to the network layer in the correct order. It can also acknowledge all frames up to and including 5, as shown in the figure. If the NAK should get lost, eventually the sender will time out for frame 2 and send it (and only it) of its own accord, but that may be a quite a while later. In effect, the NAK speeds up the retransmission of one specific frame.

Selective repeat corresponds to a receiver window larger than 1. Any frame within the window may be accepted and buffered until all the preceding ones have been passed to the network layer. This approach can require large amounts of data link layer memory if the window is large.

These two alternative approaches are trade-offs between bandwidth and data link layer buffer space. Depending on which resource is more valuable, one or the other can be used. Figure 3-17 shows a pipelining protocol in which the receiving data link layer only accepts frames in order; frames following an error are discarded. In this protocol, for the first time, we have now dropped the assumption that the network layer always has an infinite supply of packets to send. When the network layer has a packet it wants to send, it can cause a network_layer_ready event to happen. However, in order to enforce the flow control rule of no more than MAX_SEQ unacknowledged frames outstanding at any time, the data link layer must be able to prohibit the network layer from bothering it with more work. The library procedures enable_network_layer and disable_network_layer perform this function.

Note that a maximum of MAX_SEQ frames and not MAX_SEQ + 1 frames may be outstanding at any instant, even though there are MAX_SEQ + 1 distinct sequence numbers: 0, 1, 2, ..., MAX_SEQ. To see why this restriction is needed, consider the following scenario with MAX_SEQ = 7.

1. The sender sends frames 0 through 7.
2. A piggybacked acknowledgement for frame 7 eventually comes back to the sender.
3. The sender sends another eight frames, again with sequence numbers 0 through 7.
4. Now another piggybacked acknowledgement for frame 7 comes in.

The question is: Did all eight frames belonging to the second batch arrive successfully, or did all eight get lost (counting discards following an error as lost)? In both cases the receiver would be sending frame 7 as the acknowledgement. The sender has no way of telling. For this reason the maximum number of outstanding
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
  to MAX_SEQ frames without waiting for an ack. In addition, unlike the previous protocols,
  the network layer is not assumed to have a new packet all the time. Instead, the
  network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7 /* should be 2^n – 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
    /* Return true if a <= b < c circularly; false otherwise. */
    if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return(true);
    else
        return(false);
}

static void send_data(seq_nr frameNr, seq_nr frame_expected, packet buffer[])
{
    /* Construct and send a data frame. */
    frame s; /* scratch variable */
    s.info = buffer[frameNr]; /* insert packet into frame */
    s.seq = frameNr; /* insert sequence number into frame */
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
    to_physical_layer(&s); /* transmit the frame */
    start_timer(frameNr); /* start the timer running */
}

void protocol5(void)
{
    seq_nr next_frame_to_send; /* MAX_SEQ > 1; used for outbound stream */
    seq_nr ack_expected; /* oldest frame as yet unacknowledged */
    seq_nr frame_expected; /* next frame expected on inbound stream */
    frame r; /* scratch variable */
    packet buffer[MAX_SEQ + 1]; /* buffers for the outbound stream */
    seq_nr nbuffered; /* # output buffers currently in use */
    seq_nr i; /* used to index into the buffer array */
    event_type event;

    enable_network_layer(); /* allow network_layer_ready events */
    ack_expected = 0; /* next ack expected inbound */
    next_frame_to_send = 0; /* next frame going out */
    frame_expected = 0; /* number of frame expected inbound */
    nbuffered = 0; /* initially no packets are buffered */
while (true) {
    wait_for_event(&event); /* four possibilities: see event_type above */

    switch(event) {
        case network_layer_ready: /* the network layer has a packet to send */
            /* Accept, save, and transmit a new frame. */
            from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
            nbuffered = nbuffered + 1; /* expand the sender's window */
            send_data(next_frame_to_send, frame_expected, buffer); /* transmit the frame */
            inc(next_frame_to_send); /* advance sender's upper window edge */
            break;

        case frame_arrival: /* a data or control frame has arrived */
            from_physical_layer(&r); /* get incoming frame from physical layer */
            if (r.seq == frame_expected) {
                /* Frames are accepted only in order. */
                to_network_layer(&r.info); /* pass packet to network layer */
                inc(frame_expected); /* advance lower edge of receiver's window */
            }
            /* Ack n implies n - 1, n - 2, etc. Check for this. */
            while (between(ack_expected, r.ack, next_frame_to_send)) {
                /* Handle piggybacked ack. */
                nbuffered = nbuffered - 1; /* one frame fewer buffered */
                stop_timer(ack_expected); /* frame arrived intact; stop timer */
                inc(ack_expected); /* contract sender's window */
            }
            break;

        case cksum_err: break; /* just ignore bad frames */

        case timeout: /* trouble; retransmit all outstanding frames */
            next_frame_to_send = ack_expected; /* start retransmitting here */
            for (i = 1; i <= nbuffered; i++) {
                send_data(next_frame_to_send, frame_expected, buffer); /* resend 1 frame */
                inc(next_frame_to_send); /* prepare to send the next one */
            }
    }

    if (nbuffered < MAX_SEQ)
        enable_network_layer();
    else
        disable_network_layer();
}
frames must be restricted to \textit{MAX_SEQ}.

Although protocol 5 does not buffer the frames arriving after an error, it does not escape the problem of buffering altogether. Since a sender may have to retransmit all the unacknowledged frames at a future time, it must hang on to all transmitted frames until it knows for sure that they have been accepted by the receiver. When an acknowledgement comes in for frame \( n \), frames \( n - 1 \), \( n - 2 \), and so on, are also automatically acknowledged. This property is especially important when some of the previous acknowledgement-bearing frames were lost or garbled. Whenever any acknowledgement comes in, the data link layer checks to see if any buffers can now be released. If buffers can be released (i.e., there is some room available in the window), a previously blocked network layer can now be allowed to cause more \textit{network_layer_ready} events.

For this protocol, we are assuming that there is always reverse traffic on which to piggyback acknowledgements. If there is no reverse traffic, no acknowledgements can be sent. Protocol 4 does not need this assumption since it sends back one frame every time it receives a frame, even if it has just already sent that frame. In the next protocol we will solve the problem of one-way traffic in an elegant way.

Because protocol 5 has multiple outstanding frames, it logically needs multiple timers, one per outstanding frame. Each frame times out independently of all the other ones. All of these timers can easily be simulated in software, using a single hardware clock that causes interrupts periodically. The pending timeouts form a linked list, with each node of the list telling how many clock ticks until the timer expires, the frame being timed, and a pointer to the next node.

\begin{center}
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3-18}
\caption{Simulation of multiple timers in software.}
\end{figure}
\end{center}

As an illustration of how the timers could be implemented, consider the example of Fig. 3-19. Assume that the clock ticks once every 100 msec. Initially the real time is 10:00:00.0 and there are three timeouts pending, at 10:00:00.5, 10:00:01.3, and 10:00:01.9. Every time the hardware clock ticks, the real time is
updated and the tick counter at the head of the list is decremented. When the tick counter becomes zero, a timeout is caused and the node removed from the list, as shown in Fig. 3-19(b). Although this organization requires the list to be scanned when start_timer or stop_timer is called, it does not require much work per tick. In protocol 5, both of these routines have been given a parameter, indicating which frame is to be timed.

3.4.3 A Protocol Using Selective Repeat

Protocol 5 works well if errors are rare, but if the line is poor it wastes a lot of bandwidth on retransmitted frames. An alternative strategy for handling errors is to allow the receiver to accept and buffer the frames following a damaged or lost one. Such a protocol does not discard frames merely because an earlier frame was damaged or lost.

In this protocol, both sender and receiver maintain a window of acceptable sequence numbers. The sender’s window size starts out at 0 and grows to some predefined maximum, MAX_SEQ. The receiver’s window, in contrast, is always fixed in size and equal to MAX_SEQ. The receiver has a buffer reserved for each sequence number within its window. Associated with each buffer is a bit (arrived) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function between to see if it falls within the window. If so, and if it has not already been received, it is accepted and stored. This action is taken without regard to whether or not it contains the next packet expected by the network layer. Of course, it must be kept within the data link layer and not passed to the network layer until all the lower numbered frames have already been delivered to the network layer in the correct order. A protocol using this algorithm is given in Fig. 3-19.

Nonsequential receive introduces certain problems not present in protocols in which frames are only accepted in order. We can illustrate the trouble most easily with an example. Suppose that we have a 3-bit sequence number, so that the sender is permitted to transmit up to seven frames before being required to wait for an acknowledgement. Initially the sender and receiver’s windows are as shown in Fig. 3-20(a). The sender now transmits frames 0 through 6. The receiver’s window allows it to accept any frame with sequence number between 0 and 6 inclusive. All seven frames arrive correctly, so the receiver acknowledges them and advances its window to allow receipt of 7, 0, 1, 2, 3, 4, or 5, as shown in Fig. 3-20(b). All seven buffers are marked empty.

It is at this point that disaster strikes in the form of a lightning bolt hitting the telephone pole and wiping out all the acknowledgements. The sender eventually times out and retransmits frame 0. When this frame arrives at the receiver, a check is made to see if it is within the receiver’s window. Unfortunately, in Fig. 3-20(b) frame 0 is within the new window, so it will be accepted. The receiver sends a piggybacked acknowledgement for frame 6, since 0 through 6
/* Protocol 6 (nonsequential receive) accepts frames out of order, but passes packets to the
network layer in order. Associated with each outstanding frame is a timer. When the timer
expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7 /* should be 2^n − 1 */
#define NR_BUFS (MAX_SEQ + 1)/2

typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"

boolean no_nak = true; /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1; /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
    return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame/frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
    /* Construct and send a data, ack, or nak frame. */
    frame s; /* scratch variable */
    s.kind = fk; /* kind == data, ack, or nak */
    if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
    s.seq = frame_nr; /* only meaningful for data frames */
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    if (fk == nak) no_nak = false; /* one nak per frame, please */
    to_physical_layer(&s); /* transmit the frame */
    if (fk == data) start_timer(frame_nr % NR_BUFS);
    stop_ack_timer(); /* no need for separate ack frame */
}

void protocol6(void)
{
    seq_nr ack_expected; /* lower edge of sender’s window */
    seq_nr next_frame_to_send; /* upper edge of sender’s window + 1 */
    seq_nr frame_expected; /* lower edge of receiver’s window */
    seq_nr too_far; /* upper edge of receiver’s window + 1 */
    int i; /* index into buffer pool */
    frame r; /* scratch variable */
    packet out_buf[NR_BUFS]; /* buffers for the outbound stream */
    packet in_buf[NR_BUFS]; /* buffers for the inbound stream */
    boolean arrived[NR_BUFS]; /* inbound bit map */
    seq_nr nbuffers; /* how many output buffers currently used */
    event.type event;

    enable_network_layer(); /* initialize */
    ack_expected = 0; /* next ack expected on the inbound stream */
    next_frame_to_send = 0; /* number of next outgoing frame */
    frame_expected = 0;
    too_far = NR_BUFS;
    nbuffers = 0; /* initially no packets are buffered */
    for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
while (true) {
    wait_for_event(&event); /* five possibilities: see event_type above */
    switch(event) {
        case network_layer_ready: /* accept, save, and transmit a new frame */
            nbuffered = nbuffered + 1; /* expand the window */
            from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
            send_frame(data, next_frame_to_send, frame_expected, out_buf); /* transmit the frame */
            inc(next_frame_to_send); /* advance upper window edge */
            break;
        case frame_arrival: /* a data or control frame has arrived */
            from_physical_layer(&r); /* fetch incoming frame from physical layer */
            if (r.kind == data) {
                /* An undamaged frame has arrived. */
                if ((r.seq != frame_expected) && no_nak)
                    send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
                if (between(frame_expected, r.seq, too_far) && (arrived[frame_expected % NR_BUFS] == false)) {
                    /* Frames may be accepted in any order. */
                    arrived[frame_expected % NR_BUFS] = true; /* mark buffer as full */
                    in_buf[frame_expected % NR_BUFS] = r.info; /* insert data into buffer */
                    while (arrived[frame_expected % NR_BUFS]) {
                        /* Pass frames and advance window. */
                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc(frame_expected); /* advance lower edge of receiver's window */
                        inc(too_far); /* advance upper edge of receiver's window */
                        start_ack_timer(); /* to see if a separate ack is needed */
                    }
                }
            } else if ((r.kind == nak) && between(ack_expected, (r.ack+1) % (MAX_SEQ+1), next_frame_to_send))
                send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
        case cksum_err:
            if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* damaged frame */
            break;
        case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
            break;
        case ack_timeout:
            send_frame(ack, 0, frame_expected, out_buf); /* ack timer expired; send ack */
    }
    if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
}

Figure 3-19. A sliding window protocol using selective repeat.
have been received.

The sender is happy to learn that all its transmitted frames did actually arrive correctly, so it advances its window and immediately sends frames 7, 0, 1, 2, 3, 4, and 5. Frame 7 will be accepted by the receiver and its packet will be passed directly to the network layer. Immediately thereafter, the receiving data link layer checks to see if it has a valid frame 0 already, discovers that it does, and passes the embedded packet to the network layer. Consequently, the network layer gets an incorrect packet, and the protocol fails.

The essence of the problem is that after the receiver advanced its window, the new range of valid sequence numbers overlapped the old one. The following batch of frames might be either duplicates (if all the acknowledgements were lost) or new ones (if all the acknowledgements were received). The poor receiver has no way of distinguishing these two cases.

The way out of this dilemma lies in making sure that after the receiver has advanced its window, there is no overlap with the original window. To ensure that there is no overlap, the maximum window size should be at most half the range of the sequence numbers, as is done in Fig. 3-20(c) and Fig. 3-20(d). For example, if 4 bits are used for sequence numbers, these will range from 0 to 15. Only eight unacknowledged frames should be outstanding at any instant. That way, if the receiver has just accepted frames 0 through 7 and advanced its window to permit acceptance of frames 8 through 15, it can unambiguously tell if subsequent frames are retransmissions (0 through 7) or new ones (8 through 15). In general, the window size for protocol 6 will be $\frac{MAX\_SEQ}{2}$.

An interesting question is: How many buffers must the receiver have? Under no conditions will it ever accept frames whose sequence numbers are below the lower edge of the window or frames whose sequence numbers are above the upper edge of the window. Consequently, the number of buffers needed is equal to the window size, not the range of sequence numbers. In the above example of a 4-bit sequence number, eight buffers, numbered 0 through 7, are needed. When frame $i$ arrives, it is put in buffer $i \mod 8$. Notice that although $i$ and $(i + 8) \mod 8$ are “competing” for the same buffer, they are never within the window at the same time, because that would imply a window size of at least 9.

For the same reason, the number of timers needed is equal to the number of buffers, not the size of the sequence space. Effectively, there is a timer associated with each buffer. When the timer runs out, the contents of the buffer are retransmitted.

In protocol 5, there is an implicit assumption that the channel is heavily loaded. When a frame arrives, no acknowledgement is sent immediately. Instead, the acknowledgement is piggybacked onto the next outgoing data frame. If the reverse traffic is light, the acknowledgement will be held up for a long period of time. If there is a lot of traffic in one direction and no traffic in the other direction, only $MAX\_SEQ$ packets are sent, and then the protocol blocks, which is why we had to assume there was always some reverse traffic.
In protocol 6 this problem is fixed. After an in-sequence data frame arrives, an auxiliary timer is started by \textit{start.ack.timer}. If no reverse traffic has presented itself before this timer expires, a separate acknowledgement frame is sent. An interrupt due to the auxiliary timer is called an \textit{ack.timeout} event. With this arrangement, one-directional traffic flow is now possible, because the lack of reverse data frames onto which acknowledgements can be piggybacked is no longer an obstacle. Only one auxiliary timer exists, and if \textit{start.ack.timer} is called while the timer is running, it is reset to a full acknowledgement timeout interval.

It is essential that the timeout associated with the auxiliary timer be appreciably shorter than the timer used for timing out data frames. This condition is required to make sure a correctly received frame is acknowledged early enough that the frame's retransmission timer does not expire and retransmit the frame.

Protocol 6 uses a more efficient strategy than protocol 5 for dealing with errors. Whenever the receiver has reason to suspect that an error has occurred, it sends a negative acknowledgement (NAK) frame back to the sender. Such a frame is a request for retransmission of the frame specified in the NAK. There are two cases when the receiver should be suspicious: a damaged frame has arrived or a frame other than the expected one arrived (potential lost frame). To avoid making multiple requests for retransmission of the same lost frame, the receiver should keep track of whether a NAK has already been sent for a given frame. The variable \textit{no.nak} in protocol 6 is true if no NAK has been sent yet for \textit{frame.expected}. If the NAK gets mangled or lost, no real harm is done, since the sender will eventually time out and retransmit the missing frame anyway. If the wrong frame arrives after a NAK has been sent and lost, \textit{no.nak} will be true and the auxiliary timer will be started. When it expires, an ACK will be sent to resynchronize the sender to the receiver's current status.

\begin{figure}[h]
\centering
\begin{tabular}{cccc}
\text{Sender} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{Receiver} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{tabular}
\caption{(a) Initial situation with a window of size seven. (b) After seven frames have been sent and received but not acknowledged. (c) Initial situation with a window size of four. (d) After four frames have been sent and received but not acknowledged.}
\end{figure}

In some situations, the time required for a frame to propagate to the destination, be processed there, and have the acknowledgement come back is (nearly)
constant. In these situations, the sender can adjust its timer to be just slightly larger than the normal time interval expected between sending a frame and receiving its acknowledgement. However, if this time is highly variable, the sender is faced with the choice of either setting the interval to a small value and risking unnecessary retransmissions, thus wasting bandwidth, or setting it to a large value, going idle for a long period after an error, thus also wasting bandwidth. If the reverse traffic is sporadic, the time before acknowledgement will be irregular, being shorter when there is reverse traffic and longer when there is not. Variable processing time within the receiver can also be a problem here. In general, whenever the standard deviation of the acknowledgement interval is small compared to the interval itself, the timer can be set “tight” and NAKs are not useful. Otherwise, the timer must be set “loose,” and NAKs can appreciably speed up retransmission of lost or damaged frames.

Closely related to the matter of timeouts and NAKs is the question of determining which frame caused a timeout. In protocol 5 it is always $\text{ack\_expected}$, because it is always the oldest. In protocol 6, there is no trivial way to determine who timed out. Suppose that frames 0 through 4 have been transmitted, meaning that the list of outstanding frames is 01234, in order from oldest to youngest. Now imagine that 0 times out, 5 (a new frame) is transmitted, 1 times out, 2 times out, and 6 (another new frame) is transmitted. At this point the list of outstanding frames is 3405126, from oldest to youngest. If all inbound traffic is lost for a while, the seven outstanding frames will time out in that order. To keep the example from getting even more complicated than it already is, we have not shown the timer administration. Instead, we just assume that the variable $\text{oldest\_frame}$ is set upon timeout to indicate which frame timed out.