for UHF and VHF TV broadcast, for FM broadcast, and many specialized applications. Packet radio networks, discussed in Section 4.6, use this frequency band. Typical data rates in this band are highly variable; the DARPA (U.S. Department of Defense Advanced Research Projects Agency) packet radio network, for example, uses 100,000 and 400,000 bps.

Below 30 MHz, long-distance propagation beyond line-of-sight is possible by reflection from the ionosphere. Ironically, the 3 to 30 MHz band is called the high-frequency (HF) band, the terminology coming from the early days of radio. This band is very noisy, heavily used (*e.g.*, by ham radio), and subject to fading. Fading can be viewed as a channel filtering phenomenon with the frequency response changing relatively rapidly in time; this is caused by time-varying multiple propagation paths from source to destination. Typical data rates in this band are 2400 bps and less.

Microwave links (above 1000 MHz) must use line-of-sight paths. The antennas (usually highly directional dishes) yield typical path lengths of 10 to 200 km. Longer paths than this can be achieved by the use of repeaters. These links can carry 1000 Mbps or so and are usually multiplexed between long-distance telephony, TV program distribution, and data.

Satellite links use microwave frequencies with a satellite as a repeater. They have similar data rates and uses as microwave links. One satellite repeater can receive signals from many ground stations and broadcast back in another frequency band to all those ground stations. The satellite can contain multiple antenna beams, allowing it to act as a switch for multiple microwave links. In Chapter 4, multiaccess techniques for sharing individual frequency bands between different ground stations are studied.

This section has provided a brief introduction to physical channels and their use in data transmission. A link in a subnet might use any of these physical channels, or might share such a channel on a TDM or FDM basis with many other uses. Despite the complexity of this subject (which we have barely touched), these links can be regarded simply as unreliable bit pipes by higher layers.

2.3 ERROR DETECTION

The subject of the next four sections is data link control. This section treats the detection of transmission errors, and the next section treats retransmission requests. Assume initially that the receiving data link control (DLC) module knows where frames begin and end. The problem then is to determine which of those frames contain errors. From the layering viewpoint, the packets entering the DLC are arbitrary bit strings (*i.e.*, the function of the DLC layer is to provide error-free packets to the next layer up, no matter what the packet bit strings are). Thus, at the receiving DLC, any bit string is acceptable as a packet and errors cannot be detected by analysis of the packet itself. Note that a transformation on packets of K bits into some other representation of length K cannot help; there are 2^K possible packets and all possible bit strings of length K must be used to represent all possible packets. The conclusion is that extra bits must be appended to a packet to detect errors.

2.3.1 Single Parity Checks

The simplest example of error detection is to append a single bit, called a *parity check*, to a string of data bits. This parity check bit has the value 1 if the number of 1's in the bit string is odd, and has the value 0 otherwise (see Fig. 2.13). In other words, the parity check bit is the sum, modulo 2, of the bit values in the original bit string (k modulo j, for integer k and positive integer j, is the integer m, $0 \leq m < j$, such that $k - m$ is divisible by j).

In the ASCII character code, characters are mapped into strings of seven bits and then a parity check is appended as an eighth bit. One can also visualize appending a parity check to the end of a packet, but it will soon be apparent that this is not a sufficiently reliable way to detect errors.

Note that the total number of 1's in an encoded string (*i.e.*, the original bit string plus the appended parity check) is always even. If an encoded string is transmitted and a single error occurs in transmission, then, whether a 1 is changed to a 0 or a 0 to a 1, the resulting number of 1's in the string is odd and the error can be detected at the receiver. Note that the receiver cannot tell which bit is in error, nor how many errors occurred; it simply knows that errors occurred because of the odd number of 1's.

It is rather remarkable that for bit strings of any length, a single parity check enables the detection of any single error in the encoded string. Unfortunately, two errors in an encoded string always leave the number of 1's even so that the errors cannot be detected. In general, any odd number of errors are detected and any even number are undetected.

Despite the appealing simplicity of the single parity check, it is inadequate for reliable detection of errors; in many situations, it only detects errors in about half of the encoded strings where errors occur. There are two reasons for this poor behavior. The first is that many modems map several bits into a single sample of the physical channel input (see Section 2.2.5), and an error in the reception of such a sample typically causes several bit errors. The second reason is that many kinds of noise, such as lightning and temporarily broken connections, cause long bursts of errors. For both these reasons, when one or more errors occur in an encoded string, an even number of errors is almost as likely as an odd number and a single parity check is ineffective.

2.3.2 Horizontal and Vertical Parity Checks

Another simple and intuitive approach to error detection is to arrange a string of data bits in a two-dimensional array (see Fig. 2.14) with one parity check for each row and one for each column. The parity check in the lower right corner can be viewed as a parity check on the row parity checks, on the column parity checks, or on the data array. If an even number of errors are confined to a single row, each of them can be detected by the

s_1	s_2	s_3	s_4	s_5	s_6	s_7	c
1	0	1	1	0	0	0	1

Figure 2.13 Single parity check. Final bit c is the modulo 2 sum of s_1 to s_k, where $k = 7$ here.

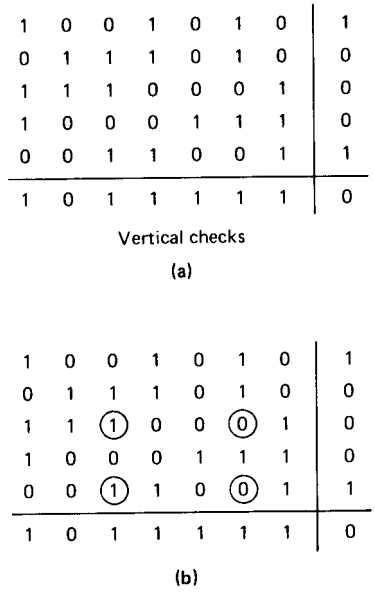

Figure 2.14 Horizontal and vertical parity checks. Each horizontal parity check checks its own row, and each column parity check checks its own column. Note, in part (b), that if each circled bit is changed, all parity checks are still satisfied.

corresponding column parity checks; similarly, errors in a single column can be detected by the row parity checks. Unfortunately, any pattern of four errors confined to two rows and two columns [*i.e.*, forming a rectangle as indicated in Fig. 2.14(b)] is undetectable.

The most common use of this scheme is where the input is a string of ASCII encoded characters. Each encoded character can be visualized as a row in the array of Fig. 2.14; the row parity check is then simply the last bit of the encoded character. The column parity checks can be trivially computed by software or hardware. The major weakness of this scheme is that it can fail to detect rather short bursts of errors (*e.g.*, two adjacent bits in each of two adjacent rows). Since adjacent errors are quite likely in practice, the likelihood of such failures is undesirably high.

2.3.3 Parity Check Codes

The nicest feature about horizontal and vertical parity checks is that the underlying idea generalizes immediately to arbitrary parity check codes. The underlying idea is to start with a bit string (the array of data bits in Fig. 2.14) and to generate parity checks on various subsets of the bits (the rows and columns in Fig. 2.14). The transformation from the string of data bits to the string of data bits and parity checks is called a *parity check code* or *linear code*. An example of a parity check code (other than the horizontal and vertical case) is given in Fig. 2.15. A parity check code is defined by the particular collection of subsets used to generate parity checks. Note that the word *code* refers to the transformation itself; we refer to an encoded bit string (data plus parity checks) as a *code word*.

Let K be the length of the data string for a given parity check code and let L be the number of parity checks. For the frame structure in Fig. 2.2, one can view the data

s_1	s_2	s_3	c_1	c_2	c_3	c_4	
1	0	0	1	1	1	0	
0	1	0	0	1	1	1	$c_1 = s_1 + s_3$
0	0	1	1	1	0	1	$c_2 = s_1 + s_2 + s_3$
1	1	0	1	0	0	1	$c_3 = s_1 + s_2$
1	0	1	0	0	1	1	$c_4 = s_2 + s_3$
1	1	1	0	1	0	0	
0	0	0	0	0	0	0	
0	1	1	1	0	1	0	

Figure 2.15 Example of a parity check code. Code words are listed on the left, and the rule for generating the parity checks is given on the right.

string as the header and packet, and view the set of parity checks as the trailer. Note that it is important to detect errors in the control bits of the header as well as to detect errors in the packets themselves. Thus, $K + L$ is the frame length, which for the present is regarded as fixed. For a given code, each of the possible 2^K data strings of length K is mapped into a frame (*i.e.*, code word) of length $K + L$. In an error-detection system, the frame is transmitted and the receiving DLC module determines if each of the parity checks is still the modulo 2 sum of the corresponding subset of data bits. If so, the frame is regarded by the receiver as error-free, and if not, the presence of errors is detected. If errors on the link convert one code word into another, the frame is regarded by the receiver as error-free, and undetectable errors are said to have occurred in the frame.

Given any particular code, one would like to be able to predict the probability of undetectable errors in a frame for a particular link. Unfortunately, this is very difficult. First, errors on links tend to be dependent and to occur in bursts; there are no good models for the length or intensity of these bursts, which vary widely among links of the same type. Second, for any reasonable code, the frequency of undetectable errors is very small and is thus both difficult to measure experimentally and dependent on rare, difficult-to-model events. The literature contains many calculations of the probability of undetectable errors, but these calculations are usually based on the assumption of independent errors; the results are usually orders of magnitude away from reality.

As a result of these difficulties, the effectiveness of a code for error detection is usually measured by three parameters: (1) the minimum distance of the code, (2) the burst-detecting capability, and (3) the probability that a completely random string will be accepted as error-free. The minimum distance of a code is defined as the smallest number of errors that can convert one code word into another. As we have seen, the minimum distance of a code using a single parity check is 2, and the minimum distance of a code with horizontal and vertical parity checks is 4.

The length of a burst of errors in a frame is the number of bits from the first error to the last, inclusive. The burst-detecting capability of a code is defined as the largest integer B such that a code can detect all bursts of length B or less. The burst-detecting capability of the single parity check code is 1, whereas the burst-detecting capability of

a code with horizontal and vertical parity checks is 1 plus the length of a row (assuming that rows are sent one after the other).

By a completely random string of length $K + L$ is meant that each such string is received with probability 2^{-K-L}. Since there are 2^K code words, the probability of an undetected error is the probability that the random string is one of the code words; this occurs with probability 2^{-L} (the possibility that the received random string happens to be the same as the transmitted frame is ignored). This is usually a good estimate of the probability of undetectable errors given that both the minimum distance and the burst-detecting capability of the code are greatly exceeded by the set of errors on a received frame.

Parity check codes can be used for error correction rather than just for error detection. For example, with horizontal and vertical parity checks, any single error can be corrected simply by finding the row and column with odd parity. It is shown in Problem 2.10 that a code with minimum distance d can be used to correct any combination of fewer than $d/2$ errors. Parity check codes (and convolutional codes, which are closely related but lack the frame structure) are widely used for error correction at the physical layer. The modern approach to error correction is to view it as part of the modulation and demodulation process, with the objective of creating a virtual bit pipe with relatively low error rate.

Error correction is generally not used at the DLC layer, since the performance of an error correction code is heavily dependent on the physical characteristics of the channel. One needs error detection at the DLC layer, however, to detect rare residual errors from long noisy periods.

2.3.4 Cyclic Redundancy Checks

The parity check codes used for error detection in most DLCs today are cyclic redundancy check (CRC) codes. The parity check bits are called the CRC. Again, let L be the length of the CRC (*i.e.*, the number of check bits) and let K be the length of the string of data bits (*i.e.*, the header and packet of a frame). It is convenient to denote the data bits as $s_{K-1}, s_{K-2}, \ldots, s_1, s_0$, and to represent the string as a polynomial $s(D)$ with coefficients $s_{K-1}, \ldots, s_0$,

$$s(D) = s_{K-1}D^{K-1} + s_{K-2}D^{K-2} + \cdots + s_0 \tag{2.14}$$

The powers of the indeterminate D can be thought of as keeping track of which bit is which; high-order terms are viewed as being transmitted first. The CRC is represented as another polynomial,

$$c(D) = c_{L-1}D^{L-1} + \cdots + c_1 D + c_0 \tag{2.15}$$

The entire frame of transmitted information and CRC can then be represented as $x(D) = s(D)D^L + c(D)$, that is, as

$$x(D) = s_{K-1}D^{L+K-1} + \cdots + s_0 D^L + c_{L-1}D^{L-1} + \cdots + c_0 \tag{2.16}$$

The CRC polynomial $c(D)$ is a function of the information polynomial $s(D)$, defined in terms of a *generator polynomial* $g(D)$; this is a polynomial of degree L with binary coefficients that specifies the particular CRC code to be used.

$$g(D) = D^L + g_{L-1}D^{L-1} + \cdots + g_1 D + 1 \tag{2.17}$$

For a given $g(D)$, the mapping from the information polynomial to the CRC polynomial $c(D)$ is given by

$$c(D) = \text{ Remainder } \left[\frac{s(D)D^L}{g(D)}\right] \tag{2.18}$$

The polynomial division above is just ordinary long division of one polynomial by another, except that the coefficients are restricted to be binary and the arithmetic on coefficients is performed modulo 2. Thus, for example, $(1+1)$ modulo $2 = 0$ and $(0-1)$ modulo $2 = 1$. Note that subtraction using modulo 2 arithmetic is the same as addition. As an example of the operation in Eq. (2.18),

$$\begin{array}{r} D^2 + D \\ D^3 + D^2 + 1 \overline{)\, D^5 + \qquad D^3} \\ \underline{D^5 + D^4 + \qquad D^2} \\ \qquad D^4 + D^3 + D^2 \\ \underline{D^4 + D^3 + \qquad D} \\ \qquad D^2 + D = \text{ Remainder} \end{array}$$

Since $g(D)$ is a polynomial of degree at most L, the remainder is of degree at most $L-1$. If the degree of $c(D)$ is less than $L-1$, the corresponding leading coefficients $c_{L-1}, \ldots$, in Eq. (2.18) are taken as zero.

This long division can be implemented easily in hardware by the feedback shift register circuit shown in Fig. 2.16. By comparing the circuit with the long division above, it can be seen that the successive contents of the shift register cells are just the coefficients of the partial remainders in the long division. In practice, CRCs are usually calculated by VLSI chips, which often perform the other functions of DLC as well.

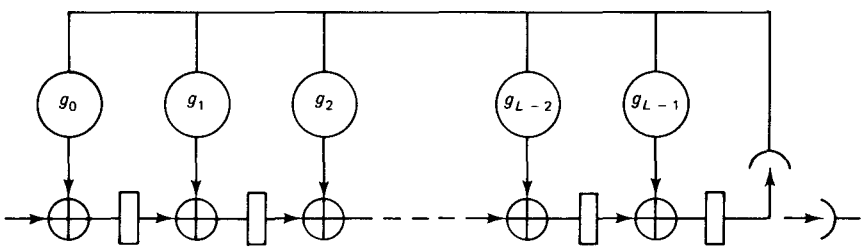

Figure 2.16 Shift register circuit for dividing polynomials and finding the remainder. Each rectangle indicates a storage cell for a single bit and the preceding circles denote modulo 2 adders. The large circles at the top indicate multiplication by the value of g_i. Initially, the register is loaded with the first L bits of $s(D)$ with s_{K-1} at the right. On each clock pulse, a new bit of $s(D)$ comes in at the left and the register reads in the corresponding modulo 2 sum of feedback plus the contents of the previous stage. After K shifts, the switch at the right moves to the horizontal position and the CRC is read out.

Let $z(D)$ be the quotient resulting from dividing $s(D)D^L$ by $g(D)$. Then, $c(D)$ can be represented as

$$s(D)D^L = g(D)z(D) + c(D) \tag{2.19}$$

Subtracting $c(D)$ (modulo 2) from both sides of this equation and recognizing that modulo 2 subtraction and addition are the same, we obtain

$$x(D) = s(D)D^L + c(D) = g(D)z(D) \tag{2.20}$$

Thus, all code words are divisible by $g(D)$, and all polynomials divisible by $g(D)$ are code words. It has not yet been shown that the mapping from $s(D)$ to $x(D)$ corresponds to a parity check code. This is demonstrated in Problem 2.15 but is not necessary for the subsequent development.

Now suppose that $x(D)$ is transmitted and that the received sequence is represented by a polynomial $y(D)$, where $x(D)$ and $y(D)$ differ because of the errors on the communication link. If the error sequence is represented as a polynomial $e(D)$, then $y(D) = x(D) + e(D)$, where, as throughout this section, + means modulo 2 addition; each error in the frame corresponds to a nonzero coefficient in $e(D)$ [*i.e.*, a coefficient in which $y(D)$ and $x(D)$ differ]. At the receiver, Remainder$[y(D)/g(D)]$ can be calculated by essentially the same circuit as that above. Since it has been shown that $x(D)$ is divisible by $g(D)$,

$$\text{Remainder}\left[\frac{y(D)}{g(D)}\right] = \text{Remainder}\left[\frac{e(D)}{g(D)}\right] \tag{2.21}$$

If no errors occur, then $e(D) = 0$ and the remainder above will be 0. The rule followed by the receiver is to decide that the frame is error- free if this remainder is 0 and to decide that there are errors otherwise. When errors actually occur [*i.e.*, $e(D) \neq 0$], the receiver fails to detect the errors only if this remainder is 0; this occurs only if $e(D)$ is itself some code word. In other words, $e(D) \neq 0$ is undetectable if and only if

$$e(D) = g(D)z(D) \tag{2.22}$$

for some nonzero polynomial $z(D)$. We now explore the conditions under which undetected errors can occur.

First, suppose that a single error occurs, say $e_i = 1$, so that $e(D) = D^i$. Since $g(D)$ has at least two nonzero terms (*i.e.*, D^L and 1), $g(D)z(D)$ must also have at least two nonzero terms for any nonzero $z(D)$ (see Problem 2.13). Thus $g(D)z(D)$ cannot equal D^i; since this is true for all i, all single errors are detectable. By the same type of argument, since the highest-order and lowest-order terms in $g(D)$ (*i.e.*, D^L and 1, respectively) differ by L, the highest-order and lowest-order terms in $g(D)z(D)$ differ by at least L for all nonzero $z(D)$. Thus, if $e(D)$ is a code word, the burst length of the errors is at least $L + 1$ (the +1 arises from the definition of burst length as the number of positions from the first error to the last error *inclusive*).

Next, suppose that a double error occurs, say in positions i and j, so that

$$e(D) = D^i + D^j = D^j(D^{i-j} + 1), \quad i > j \tag{2.23}$$

From the argument above, D^j is not divisible by $g(D)$ or by any factor of $g(D)$; thus, $e(D)$ fails to be detected only if $D^{i-j}+1$ is divisible by $g(D)$. For any binary polynomial $g(D)$ of degree L, there is some smallest n for which $D^n + 1$ is divisible by $g(D)$. It is known from the theory of finite fields that this smallest n can be no larger than $2^L - 1$; moreover, for all $L > 0$, there are special L-degree polynomials, called *primitive polynomials*, such that this smallest n is equal to $2^L - 1$. Thus, if $g(D)$ is chosen to be such a primitive polynomial of degree L, and if the frame length is restricted to be at most $2^L - 1$, then $D^{i-j} + 1$ cannot be divisible by $g(D)$; thus, all double errors are detected.

In practice, the generator polynomial $g(D)$ is usually chosen to be the product of a primitive polynomial of degree $L-1$ times the polynomial $D+1$. It is shown in Problem 2.14 that a polynomial $e(D)$ is divisible by $D + 1$ if and only if $e(D)$ contains an even number of nonzero coefficients. This ensures that all odd numbers of errors are detected, and the primitive polynomial ensures that all double errors are detected (as long as the block length is less than 2^{L-1}). Thus, any code of this form has a minimum distance of at least 4, a burst-detecting capability of at least L, and a probability of failing to detect errors in completely random strings of 2^{-L}. There are two standard CRCs with length $L = 16$ (denoted CRC-16 and CRC-CCITT). Each of these CRCs is the product of $D+1$ times a primitive $(L-1)$- degree polynomial, and thus both have the foregoing properties. There is also a standard CRC with $L = 32$. It is a 32-degree primitive polynomial, and has been shown to have a minimum distance of 5 for block lengths less than 3007 and 4 for block lengths less than 12,145 [FKL86]. These polynomials are as follows:

$$g(D) = D^{16} + D^{15} + D^2 + 1 \qquad \text{for CRC-16}$$

$$g(D = D^{16} + D^{12} + D^5 + 1 \qquad \text{for CRC-CCITT}$$

$$g(D) = D^{32} + D^{26} + D^{23} + D^{22} + D^{16} + D^{12} + D^{11} + D^{10} + D^8 + D^7 + D^5 + D^4 + D^2 + D^1 + 1$$

2.4 ARQ: RETRANSMISSION STRATEGIES

The general concept of *a*utomatic *r*epeat re*q*uest (ARQ) is to detect frames with errors at the receiving DLC module and then to request the transmitting DLC module to repeat the information in those erroneous frames. Error detection was discussed in the preceding section, and the problem of requesting retransmissions is treated in this section. There are two quite different aspects of retransmission algorithms or protocols. The first is that of correctness: Does the protocol succeed in releasing each packet, once and only once, without errors, from the receiving DLC? The second is that of efficiency: How much of the bit-transmitting capability of the bit pipe is wasted by unnecessary waiting and by sending unnecessary retransmissions? First, several classes of protocols are developed and shown to be correct (in a sense to be defined more precisely later). Later, the effect that the various parameters in these classes have on efficiency is considered.

Recall from Fig. 2.2 that packets enter the DLC layer from the network layer. The DLC module appends a header and trailer to each packet to form a frame, and the frames are transmitted on the virtual bit pipe (*i.e.*, are sent to the physical layer for transmission). When errors are detected in a frame, a new frame containing the old packet is transmitted. Thus, the first transmitted frame might contain the first packet, the next frame the second packet, the third frame a repetition of the first packet, and so forth. When a packet is repeated, the frame header and trailer might or might not be the same as in the earlier version.

Since framing will not be discussed until the next section, we continue to assume that the receiving DLC knows when frames start and end; thus a CRC (or any other technique) may be used for detecting errors. We also assume, somewhat unrealistically, that *all* frames containing transmission errors are detected. The reason for this is that we want to prove that ARQ works correctly except when errors are undetected. This is the best that can be hoped for, since error detection cannot work with perfect reliability and bounded delay; in particular, any code word can be changed into another code word by some string of transmission errors. This can cause erroneous data to leave the DLC or, perhaps worse, can cause some control bits to be changed. In what follows, we refer to frames without transmission errors as *error-free frames* and those with transmission errors as *error frames*. We are assuming, then, that the receiver can always disinguish error frames from error-free frames.

Finally, we need some assumptions about the bit pipes over which these frames are traveling. The reason for these assumptions will be clearer when framing is studied; in effect, these assumptions will allow us to relax the assumption that the receiving DLC has framing information. Assume first that each transmitted frame is delayed by an arbitrary and variable time before arriving at the receiver, and assume that some frames might be "lost" and never arrive. Those frames that arrive, however, are assumed to arrive in the same order as transmitted, with or without errors. Figure 2.17 illustrates this behavior.

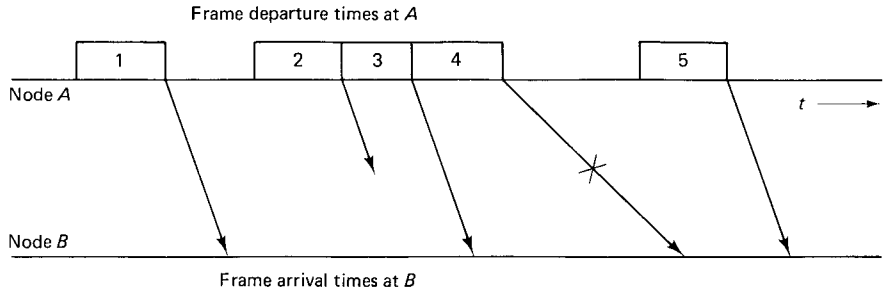

Figure 2.17 Model of frame transmissions: frame 2 is lost and never arrives; frame 4 contains errors; the frames have variable transmission delay, but those that arrive do so in the order transmitted. The rectangles at the top of the figure indicate the duration over which the frame is being transmitted. Each arrow indicates the propagation of a frame from A to B, and the arrowhead indicates the time at which the frame is completely received; at this time the CRC can be recomputed and the frame accepted or not.

2.4.1 Stop-and-Wait ARQ

The simplest type of retransmission protocol is called *stop-and-wait*. The basic idea is to ensure that each packet has been received correctly before initiating transmission of the next packet. Thus, in transmitting packets from point A to B, the first packet is transmitted in the first frame, and then the sending DLC waits. If the frame is error-free, B sends an acknowledgment (called an ack) back to A; if the frame is an error frame, B sends a negative acknowledgment (called a nak) back to A. Since errors can occur from B to A as well as from A to B, the ack or nak is protected with a CRC.

If an error-free frame is received at B, and the corresponding ack frame to A is error-free, then A can start to send the next packet in a new frame. Alternatively, detected errors can occur either in the transmission of the frame or the return ack or nak, and in either case A resends the old packet in a new frame. Finally, if either the frame or the return ack or nak is lost, A must eventually time-out and resend the old packet.

Figure 2.18 illustrates a potential malfunction in such a strategy. Since delays are arbitrary, it is possible for node A to time-out and resend a packet when the first transmission and/or the corresponding ack is abnormally delayed. If B receives both transmissions of the given packet correctly, B has no way of knowing whether the second transmission is a new packet or a repetition of the old packet. One might think that B could simply compare the packets to resolve this issue, but as far as the DLC layer is concerned, packets are arbitrary bit strings and the first and second packets could be identical; it would be a violation of the principle of layering for the DLC layer to rely on higher layers to ensure that successive packets are different.

The simplest solution to this problem is for the sending DLC module (at A) to use a sequence number in the frame header to identify successive packets. Unfortunately, even the use of sequence numbers from A to B is not quite enough to ensure correct operation. The problem is that acks can get lost on the return channel, and thus when B gets the same packet correctly twice in a row, it has to send a new ack for the second reception (see Fig. 2.19). After transmitting the packet twice but receiving only one ack,

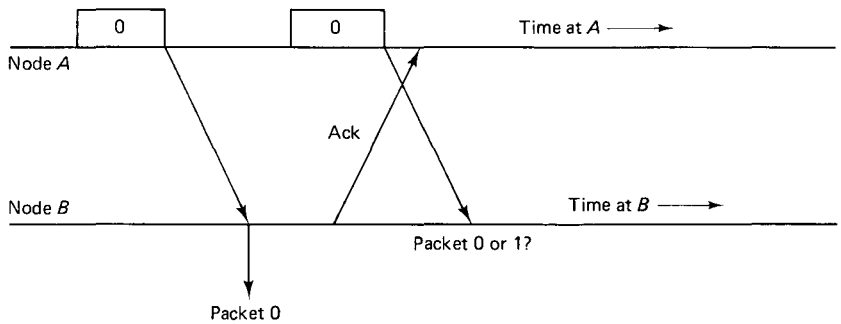

Figure 2.18 The trouble with unnumbered packets. If the transmitter at A times-out and sends packet 0 twice, the receiver at B cannot tell whether the second frame is a retransmission of packet 0 or the first transmission of packet 1.

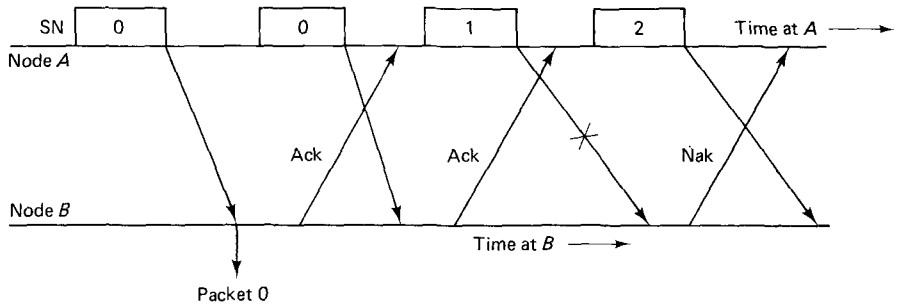

Figure 2.19 The trouble with unnumbered acks. If the transmitter at A times-out and sends packet 0 twice, node B can use the sequence numbers to recognize that packet 0 is being repeated. It must send an ack for both copies, however, and (since acks can be lost) the transmitter cannot tell whether the second ack is for packet 0 or 1.

node A could transmit the next packet in sequence, and then on receiving the second ack, could interpret that as an ack for the new packet, leading to a potential failure of the system.

To avoid this type of problem, the receiving DLC (at B), instead of returning ack or nak on the reverse link, returns the number of the next packet awaited. This provides all the information of the ack/nak, but avoids ambiguities about which frame is being acked. An equivalent convention would be to return the number of the packet just accepted, but this is not customary. Node B can request this next awaited packet upon the receipt of each packet, at periodic intervals, or at an arbitrary selection of times. In many applications, there is another stream of data from B to A, and in this case, the frames from B to A carrying requests for new A to B packets must be interspersed with data frames carrying data from B to A. It is also possible to "piggyback" these requests for new packets into the headers of the data frames from B to A (see Fig. 2.20), but as shown in Problem 2.38, this is often counterproductive for stop-and-wait ARQ. Aside from its effect on the timing of the requests from node B to A, the traffic from B to A does not affect the stop-and-wait strategy from A to B; thus, in what follows, we ignore this reverse traffic (except for the recognition that requests might be delayed). Figure 2.21 illustrates the flow of data from A to B and the flow of requests in the opposite direction.

We next specify the stop-and-wait strategy more precisely and then show that it works correctly. The correctness might appear to be self-evident, but the methodology of the demonstration will be helpful in understanding subsequent distributed algorithms. It

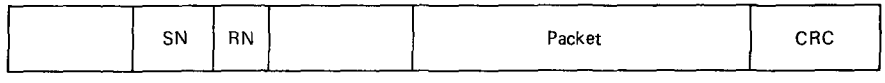

Figure 2.20 The header of a frame contains a field carrying the sequence number, SN, of the packet being transmitted. If piggybacking is being used, it also contains a field carrying the request number, RN, of the next packet awaited in the opposite direction.

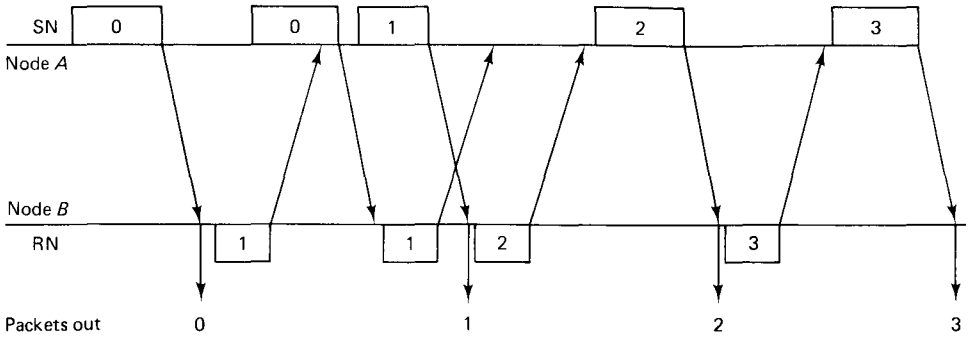

Figure 2.21 Example of use of sequence and request numbers for stop-and-wait transmission from A to B. Note that packet 0 gets repeated, presumably because node A times-out too soon. Note also that node A delays repeating packet 1 on the second request for it. This has no effect on the correctness of the protocol, but avoids unnecessary retransmissions.

will be seen that what is specified is not a single algorithm, but rather a class of algorithms in which the timing of frame transmissions is left unspecified. Showing that all algorithms in such a class are correct then allows one to specify the timing as a compromise between simplicity and efficiency without worrying further about correctness.

Assume that when the strategy is first started on the link, nodes A and B are correctly initialized in the sense that no frames are in transit on the link and that the receiver at B is looking for a frame with the same sequence number as the first frame to be transmitted from A. It makes no difference what this initial sequence number SN is as long as A and B agree on it, so we assume that $SN = 0$, since this is the conventional initial value.

The algorithm at node A for A-to-B transmission:

1. Set the integer variable SN to 0.
2. Accept a packet from the next higher layer at A; if no packet is available, wait until it is; assign number SN to the new packet.
3. Transmit the SNth packet in a frame containing SN in the sequence number field.
4. If an error-free frame is received from B containing a request number RN greater than SN, increase SN to RN and go to step 2. If no such frame is received within some finite delay, go to step 3.

The algorithm at node B for A-to-B transmission:

1. Set the integer variable RN to 0 and then repeat steps 2 and 3 forever.
2. Whenever an error-free frame is received from A containing a sequence number SN equal to RN, release the received packet to the higher layer and increment RN.
3. At arbitrary times, but within bounded delay after receiving any error-free data frame from A, transmit a frame to A containing RN in the request number field.

There are a number of conventional ways to handle the arbitrary delays between subsequent transmissions in the algorithm above. The usual procedure for node A (recall that we are discussing only the stop-and-wait strategy for A-to-B traffic) is to set a timer when a frame transmission begins. If the timer expires before receipt of a request for the next packet from B, the timer is reset and the packet is resent in a new frame. If a request for a new packet is received from B before the timer expires, the timer is disabled until A transmits the next packet. The same type of timer control can be used at node B. Alternatively, node B could send a frame containing a request for the awaited packet each time that it receives a frame from A. Also, if B is piggybacking its request numbers on data frames, it could simply send the current value of *RN* in each such frame. Note that this is not sufficient in itself, since communication from A to B would then halt in the absence of traffic from B to A; thus, node B must send nondata frames containing the *RN* values when there is no B-to-A data traffic. The important point is that whatever timing strategy is being used, it must guarantee that the intervening interval between repetitions in each direction is bounded.

Correctness of stop and wait We now go through an informal proof that this class of algorithms is correct in the sense that a never-ending stream of packets can be accepted from the higher layer at A and delivered to the higher layer at B in order and without repetitions or deletions. We continue to assume that all error frames are detected by the CRC. We also assume that there is some $q > 0$ such that each frame is received error-free with probability at least q. Finally, we recall the assumption that the link is initially empty, that the first packet from A has $SN = 0$, and that node B is initially awaiting the packet with $SN = 0$.

Proofs of this type usually break into two parts, characterized as *safety* and *liveness*. An algorithm is safe if it never produces an incorrect result, which in this case means never releasing a packet out of the correct order to the higher layer at B. An algorithm is live if it can continue forever to produce results (*i.e.*, if it can never enter a deadlock condition from which no further progress is possible). In this case liveness means the capability to continue forever to accept new packets at A and release them at B.

The safety property is self-evident for this algorithm; that is, node B is initially awaiting packet 0, and the only packet that can be released is packet 0. Subsequently (using induction if one wants to be formal), node B has released all the packets in order, up to, but not including, the current value of RN; packet RN is the only packet that it can next accept and release. When an error-free frame containing packet RN is received and released, the value of RN is incremented and the situation above is repeated with the new value of RN.

To see that the liveness property is satisfied, assume that node A first starts to transmit a given packet i at time t_1 (see Fig. 2.22). Let t_2 be the time at which this packet is received error-free and released to the higher layer at node B; let $t_2 = \infty$ if this event never occurs. Similarly, let t_3 be the time at which the sequence number at A is increased to $i + 1$, and let $t_3 = \infty$ if this never occurs. We will show that $t_1 < t_2 < t_3$ and that t_3 is finite. This is sufficient to demonstrate liveness, since using the argument repeatedly for $i = 0$, then $i = 1$, and so on, shows that each packet is transmitted with

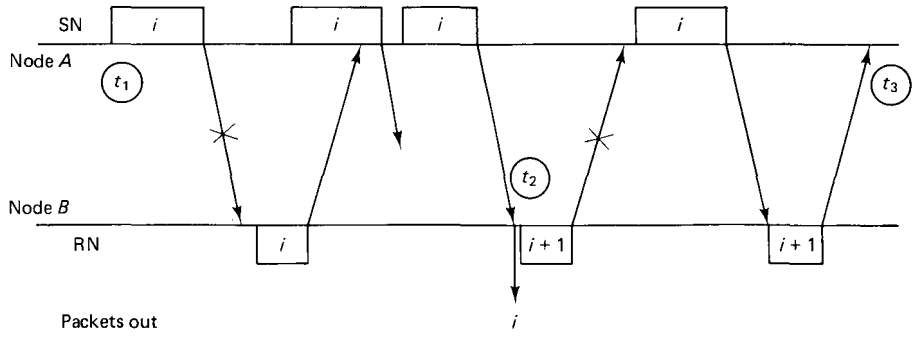

Figure 2.22 Times t_1, t_2, and t_3 when a packet is first placed in a frame for transmission at A, first received and released to the higher layer at B, and first acknowledged at A.

finite delay. Note that we cannot guarantee that the higher layer at A will always supply packets within finite delay, so that the notion of liveness here can only require finite delay given available packets to send.

Let $RN(t)$ be the value of the variable RN at node B as a function of time t and let $SN(t)$ be the corresponding value of SN at node A. It is seen directly from the algorithm statement that $SN(t)$ and $RN(t)$ are nondecreasing in t. Also, since $SN(t)$ is the largest request number received from B up to time t, $SN(t) \leq RN(t)$. By assumption, packet i has never been transmitted before time t_1, so (using the safety property) $RN(t_1) \leq i$. Since $SN(t_1) = i$, it follows that $SN(t_1) = RN(t_1) = i$. By definition of t_2 and t_3, $RN(t)$ is incremented to $i + 1$ at t_2 and $SN(t)$ is incremented to $i + 1$ at t_3. Using the fact that $SN(t) \leq RN(t)$, it follows that $t_2 < t_3$.

We have seen that node A transmits packet i repeatedly, with finite delay between successive transmissions, from t_1 until it is first received error-free at t_2. Since there is a probability $q > 0$ that each retransmission is received correctly, and retransmissions occur within finite intervals, an error-free reception eventually occurs and t_2 is finite. Node B then transmits frames carrying $RN = i+1$ from time t_2 until received error-free at time t_3. Since node A is also transmitting frames in this interval, the delay between subsequent transmissions from B is finite, and, since $q > 0$, t_3 eventually occurs; thus the interval from t_1 to t_3 is finite and the algorithm is live.

One trouble with the stop-and-wait strategy developed above is that the sequence and request numbers become arbitrarily large with increasing time. Although one could simply use a very large field for sending these numbers, it is nicer to send these numbers modulo some integer. Given our assumption that frames travel in order on the link, it turns out that a modulus of 2 is sufficient.

To understand why it is sufficient to send sequence numbers modulo 2, we first look more carefully at what happens in Fig. 2.22 when ordinary integers are used. Note that after node B receives packet i at time t_2, the subsequent frames received at B must all have sequence numbers i or greater (since frames sent before t_1 cannot arrive after t_2). Similarly, while B is waiting for packet $i + 1$, no packet greater than $i + 1$ can be sent [since $SN(t) \leq RN(t)$]. Thus, in the interval while $RN(t) = i + 1$, the received

frames all carry sequence numbers equal to i or $i + 1$. Sending the sequence number modulo 2 is sufficient to resolve this ambiguity. The same argument applies for all i. By the same argument, in the interval t_1 to t_3, while $SN(t)$ is equal to i, the request numbers received at A must be either i or $i + 1$, so again sending *RN* modulo 2 is sufficient to resolve the ambiguity. Finally, since *SN* and *RN* need be transmitted only modulo 2, it is sufficient to keep track of them at the nodes only modulo 2.

Using modulo 2 values for *SN* and *RN*, we can view nodes A and B as each having two states (for purposes of A to B traffic), corresponding to the binary value of SN at node A and RN at node B. Thus, A starts in state 0; a transition to state 1 occurs upon receipt of an error-free request for packet 1 modulo 2. Note that A has to keep track of more information than just this state, such as the contents of the current packet and the time until time-out, but the binary state above is all that is of concern here.

Node B similarly is regarded as having two possible states, 0 and 1, corresponding to the number modulo 2 of the packet currently awaited. When a packet of the desired number modulo 2 is received, the DLC at B releases that packet to the higher layer and changes state, awaiting the next packet (see Fig. 2.23). The combined state of A and B is then initially (0,0); when the first packet is received error-free, the state of B changes to 1, yielding a combined state (0, 1). When A receives the new *RN* value (*i.e.*, 1), the state of A changes to 1 and the combined state becomes (1,1). Note that there is a fixed sequence for these combined states, (0,0), (0,1), (1,1), (1,0), (0,0), and so on, and that A and B alternate in changing states. It is interesting that at the instant of the transition from (0,0) to (0,1), B knows the combined state, but subsequently, up until the transition later from (1,1) to (1,0), it does not know the combined state (*i.e.*, B never knows that A has received the ack information until the next packet is received). Similarly, A knows the combined state at the instant of the transition from (0,1) to (1,1) and of the transition from (1,0) to (0,0). The combined state is always unknown to either A or B, and is frequently unknown to both. The situation here is very similar to that in the three-army problem discussed in Section 1.4. Here, however, information is transmitted even though the combined state information is never jointly known, whereas in the three-army problem, it is impossible to coordinate an attack because of the impossibility of obtaining this joint knowledge.

The stop-and-wait strategy is not very useful for modern data networks because of its highly inefficient use of communication links. In particular, it should be possible to do something else while waiting for an ack. There are three common strategies for extending the basic idea of stop-and-wait ARQ so as to achieve higher efficiency: go back n ARQ, selective repeat ARQ, and finally, the ARPANET ARQ.

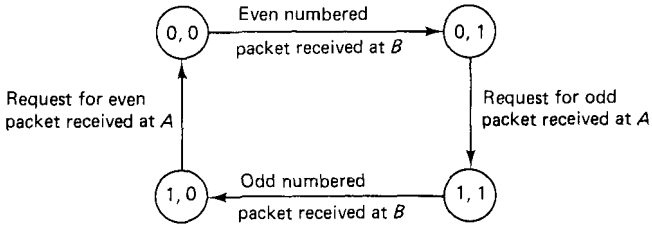

Figure 2.23 State transition diagram for stop-and-wait ARQ. The state is (*SN* mod 2 at A, *RN* mod 2 at B).

2.4.2 Go Back *n* ARQ

Go back n ARQ is the most widely used type of ARQ protocol; it appears in the various standard DLC protocols, such as HDLC, SDLC, ADCCP, and LAPB. It is not elucidating to know the meaning of these acronyms, but in fact, these standards are almost the same. They are discussed in Section 2.6, and some of their differences are mentioned there. Go back n is also the basis of most error recovery procedures at the transport layer.

The basic idea of go back n is very simple. Incoming packets to a transmitting DLC module for a link from A to B are numbered sequentially, and this sequence number SN is sent in the header of the frame containing the packet. In contrast to stop-and-wait ARQ, several successive packets can be sent without waiting for the next packet to be requested. The receiving DLC at B operates in essentially the same way as stop-and-wait ARQ. It accepts packets only in the correct order and sends request numbers RN back to A; the effect of a given request RN is to acknowledge all packets prior to RN and to request transmission of packet RN.

The go back number $n \geq 1$ in a go back n protocol is a parameter that determines how many successive packets can be sent in the absence of a request for a new packet. Specifically, node A is not allowed to send packet $i+n$ before i has been acknowledged (*i.e.*, before $i+1$ has been requested). Thus, if i is the most recently received request from node B, there is a "window" of n packets, from i to $i+n-1$, that the transmitter is allowed to send. As successively higher-numbered requests are received from B, this window slides upward; thus go back n protocols are often called *sliding window* ARQ protocols.

Figure 2.24 illustrates the operation of go back 7 ARQ when piggybacking of request numbers is being used and when there are no errors and a constant supply of traffic. Although the figure portrays data traffic in both directions, the flow of sequence numbers is shown in one direction and request numbers in the other. Note that when the first frame from A (containing packet 0) is received at B, node B is already in the middle of transmitting its second frame. The piggybacked request number at node B is traditionally in the frame header, and the frame is traditionally completely assembled before transmission starts. Thus when packet 0 is received from A, it is too late for node B to request packet 1 in the second frame, so that $RN = 1$ does not appear until the third frame from B. When this frame is completely received at A, the window at A "slides up" from $[0, 6]$ to $[1, 7]$.

Note that even in the absence of transmission errors, there are several sources of delay between the time that a packet is first assembled into a frame at A and the time when A receives an acknowledgment of the packet. First there is the transmission time of the frame, then the propagation delay, then the wait until the frame in transmission at B is completed, then the transmission time of the frame carrying the acknowledgment, and finally, the B-to-A propagation delay; the effect of these delays is discussed later.

Figure 2.25 shows the effect of error frames on go back 4 ARQ. The second frame from A, carrying packet 1, is received in error at node B. Node B continues to look for packet 1 and to transmit $RN = 1$ in frames from B to A. Packets 2, 3, and 4 from A

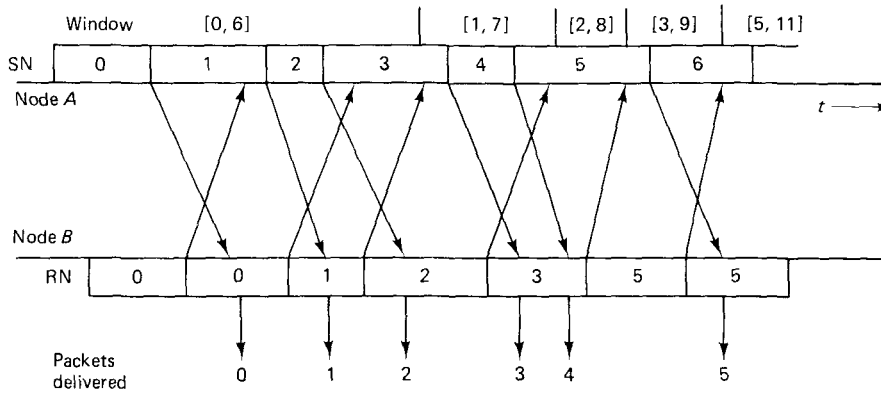

Figure 2.24 Example of go back 7 protocol for A-to-B traffic. Both nodes are sending data, but only the sequence numbers are shown for the frames transmitted from A and only the request numbers are shown for the frames from B. When packet 0 is completely received at B, it is delivered to the higher layer, as indicated at the lower left of the figure. At this point, node B wants to request packet 1, so it sends $RN = 1$ in the next outgoing frame. When that outgoing frame is completely received at A, node A updates its window from [0,6] to [1,7]. Note that when packets 3 and 4 are both received at B during the same frame transmission at B, node B awaits packet 5 and uses $RN = 5$ in the next frame from B to acknowledge both packets 3 and 4.

all arrive at B in error-free frames but are not accepted since node B is looking only for packet 1. One might think that it would be more efficient for node B to buffer packets 2, 3, and 4, thus avoiding the necessity for A to retransmit them after packet 1 is finally retransmitted. Such a buffering strategy is indeed possible and is called *selective repeat* ARQ; this is discussed in Section 2.4.3, but, by definition, go back n does not include this possibility.

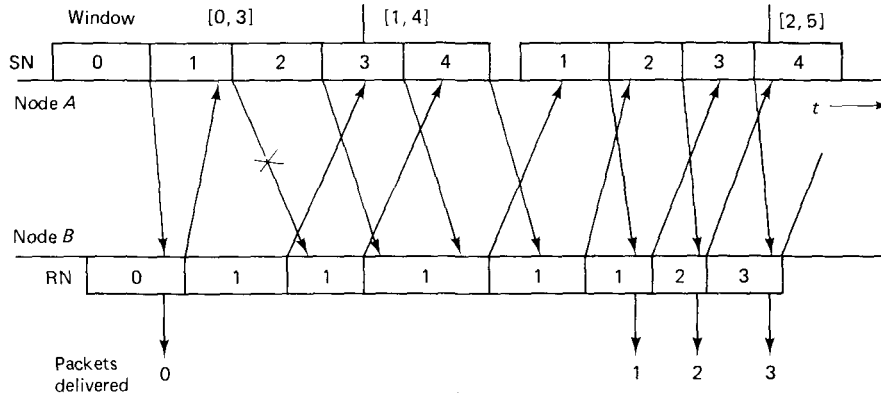

Figure 2.25 Effect of a transmission error on go back 4. Packet 1 is received in error at B, and node B continues to request packet 1 in each reverse frame until node A transmits its entire window, times-out, and goes back to packet 1.

After node A has completed transmission of the frame containing packet 4 (see Fig. 2.25), it has exhausted its window, which is still [1, 4]; node A then goes back and retransmits packet 1. The figure shows the retransmission of packet 1 after a time-out. There are many possible strategies for the timing of retransmissions within the go back n protocol, and for this reason, go back n is really an entire class of algorithms (just like stop and wait). In go back n, however, not only is the timing of transmissions unspecified, but also the selection of a packet within the window. The class of algorithms will be specified precisely after going through several more examples.

Figure 2.26 illustrates the effect of error frames in the reverse direction (from node B to A). Such an error frame need not slow down transmission in the A-to-B direction, since a subsequent error-free frame from B to A can acknowledge the packet and perhaps some subsequent packets (*i.e.*, the third frame from B to A in the figure acknowledges both packets 1 and 2). On the other hand, with a small window and long frames from B to A, an error frame from B to A can delay acknowledgments until after all the packets in the window are transmitted, thus causing A to either wait or to go back. Note that when the delayed acknowledgments get through, node A can jump forward again (from packet 3 to 5 in Fig. 2.26).

Finally, Fig. 2.27 shows that retransmissions can occur even in the absence of any transmission errors. This happens particularly in the case of short frames in one direction and long frames in the other. We discuss the impact of this phenomenon on the choice of window size and frame length later.

Rules followed by transmitter and receiver in go back *n* We now specify the precise rules followed by the class of go back n protocols and then, in the next

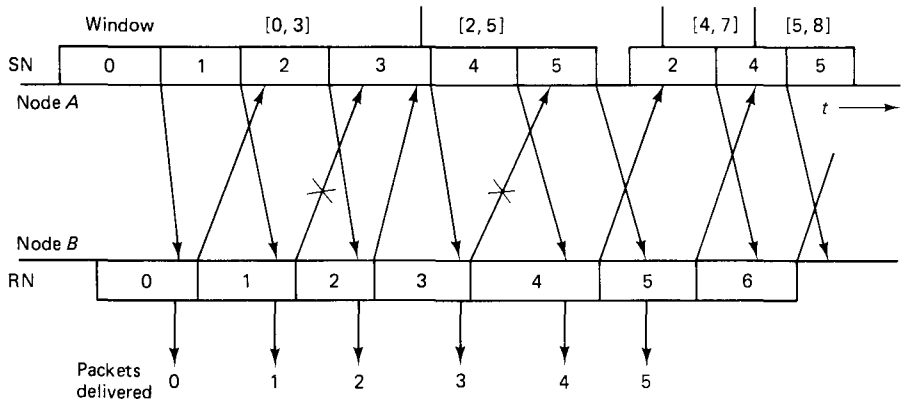

Figure 2.26 Effect of transmission errors in the reverse direction for go back 4. The first error frame, carrying $RN = 1$, causes no problem since it is followed by an error-free frame carrying $RN = 2$ and this frame reaches A before packet number 3, (*i.e.*, the last packet in the current window at A) has completed transmission and before a time-out occurs. The second error frame, carrying $RN = 3$, causes retransmissions since the following reverse frame is delayed until after node A sends its entire window and times-out. This causes A to go back and retransmit packet 2.

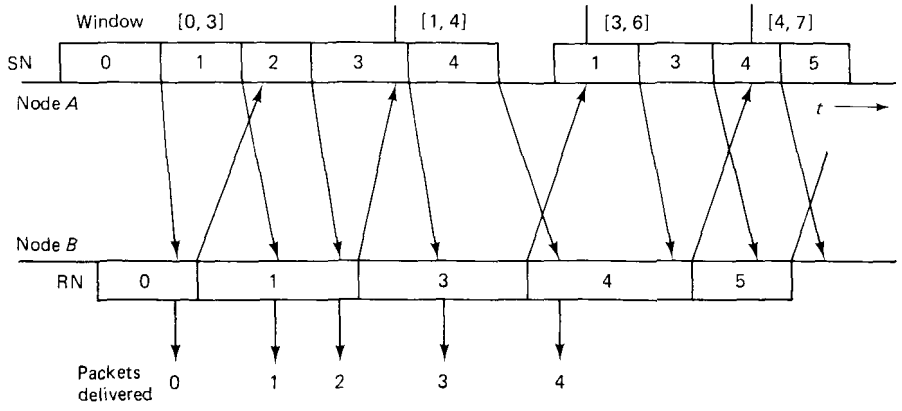

Figure 2.27 Effect of delayed feedback for go back 4. The frames in the B-to-A direction are much longer than those in the A-to-B direction, thus delaying the request numbers from getting back to A. The request for packet 1 arrives in time to allow packet 4 to be sent, but after sending packet 4, node A times-out and goes back to packet 1.

subsection, demonstrate that the algorithms in the class work correctly. We assume here that the sequence numbers and request numbers are integers that can increase without bound. The more practical case in which SN and RN are taken modulo some integer m is considered later.

The rules given here do not treat the initialization of the protocol. We simply assume that initially there are no frames in transit on the link, that node A starts with the transmission of packet number 0, and that node B is initially awaiting packet number 0. How to achieve such an initialization is discussed in Section 2.7.

The transmitter uses two integer variables, SN_{min} and SN_{max}, to keep track of its operations. SN_{min} denotes the smallest-numbered packet that has not yet been acknowledged (*i.e.*, the lower end of the window). SN_{max} denotes the number of the next packet to be accepted from the higher layer. Thus the DLC layer is attempting to transmit packets SN_{min} to $SN_{max} - 1$. Conceptually we can visualize the DLC layer as storing these packets, but it makes no difference where they are stored physically as long as they are maintained somewhere at the node for potential retransmission.

The go back n algorithm at node A for A-to-B transmission:

1. Set the integer variables SN_{min} and SN_{max} to 0.
2. Do steps 3, 4, and 5 repeatedly in any order. There can be an arbitrary but bounded delay between the time when the conditions for a step are satisfied and when the step is executed.
3. If $SN_{max} < SN_{min} + n$, and if a packet is available from the higher layer, accept a new packet into the DLC, assign number SN_{max} to it, and increment SN_{max}.
4. If an error-free frame is received from B containing a request number RN greater than SN_{min}, increase SN_{min} to RN.

5. If $SN_{min} < SN_{max}$ and no frame is currently in transmission, choose some number SN, $SN_{min} \leq SN < SN_{max}$; transmit the SNth packet in a frame containing SN in the sequence number field. At most a bounded delay is allowed between successive transmissions of packet SN_{min} over intervals when SN_{min} does not change.

The go back n algorithm at node B for A-to-B transmission:

1. Set the integer variable RN to 0 and repeat steps 2 and 3 forever.
2. Whenever an error-free frame is received from A containing a sequence number SN equal to RN, release the received packet to the higher layer and increment RN.
3. At arbitrary times, but within bounded delay after receiving any error-free data frame from A, transmit a frame to A containing RN in the request number field.

There are many conventional ways of handling the timing and ordering of the various operations in the algorithm above. Perhaps the simplest is for node A to set a timer whenever a packet is transmitted. If the timer expires before that packet is acknowledged (*i.e.*, before SN_{min} increases beyond that packet number), the packet is retransmitted. Sometimes when this approach is used, the transmitter, after going back and retransmitting SN_{min}, simply retransmits subsequent packets in order up to $SN_{max} - 1$, whether or not subsequent request numbers are received. For example, at the right-hand edge of Fig. 2.26, the transmitter might have followed packet 3 with 4 rather than 5. In terms of the algorithm as stated, this corresponds to the transmitter delaying the execution of step 4 while in the process of retransmitting a window of packets. Another possibility is for node A to cycle back whenever all the available packets in the window have been transmitted. Also, A might respond to a specific request from B for retransmission; such extra communication between A and B can be considered as part of this class of protocols in the sense that it simply guides the available choices within the algorithm.

Perhaps the simplest approach to timing at node B is to piggyback the current value of RN in each data frame going from B to A. When there are no data currently going from B to A, a nondata frame containing RN should be sent from B to A whenever a data frame is received from A.

Correctness of go back *n* We first demonstrate the correctness of this class of algorithms under our current assumptions that SN and RN are integers; we then show that correctness is maintained if SN and RN are integers modulo m, for m strictly greater than the go back number n. The correctness demonstration when SN and RN are integers is almost the same as the demonstration in Section 2.4.1 for stop and wait. In particular, we start by assuming that all frames with transmission errors are detected by the CRC, that there is some $q > 0$ such that each frame is received error-free with probability at least q, and that the system is correctly initialized in the sense that there

are no frames on the link and that nodes A and B both start at step 1 of their respective algorithms.

The safety property of the go back n algorithm is exactly the same as for stop and wait. In particular, node B releases packets to the higher layer in order, using the variable RN to track the next packet awaited. To verify the liveness property, assume that i is the value of SN_{min} at node A at a given time t_1 (see Fig. 2.28). Let t_2 be the time at which packet i is received error-free and released to the higher layer at node B; let $t_2 = \infty$ if this event never occurs. Similarly, let t_3 be the time at which SN_{min} is increased beyond i and let $t_3 = \infty$ if this never occurs. We will show that t_3 is finite and that $t_1 < t_3$ and $t_2 < t_3$. This is sufficient to demonstrate liveness, since using the argument for each successive value of SN_{min} shows that each packet is transmitted with finite delay.

Let $RN(t)$ be the value of the variable RN at node B as a function of time t and let $SN_{min}(t)$ be the corresponding value of SN_{min} at node A. It is seen directly from the algorithm statement that $SN_{min}(t)$ and $RN(t)$ are nondecreasing in t. Also, since $SN_{min}(t)$ is the largest request number (if any) received from B up to time t, $SN_{min}(t) \leq RN(t)$. By definition of t_2 and t_3, $RN(t)$ is incremented to $i + 1$ at t_2 and $SN_{min}(t)$ is increased beyond i at t_3. Using the fact that $SN_{min}(t) \leq RN(t)$, it follows that $t_2 < t_3$. Note that it is possible that $t_2 < t_1$, since packet i might have been received error-free and released at B before time t_1 and even before SN_{min} became equal to i.

From the algorithm statement, node A transmits packet i repeatedly, with finite delay between successive transmissions, from t_1 until t_3. If $t_1 < t_2$, then $RN(t) = i$ for $t_1 \leq t \leq t_2$, so the first error-free reception of packet i after t_1 will be accepted and released to the higher layer at B. Since $t_2 < t_3$, node A will retransmit packet i until this happens. Since there is a probability $q > 0$ that each retransmission is received correctly, and retransmissions occur within finite intervals, the time from t_1 to t_2 is finite. Node

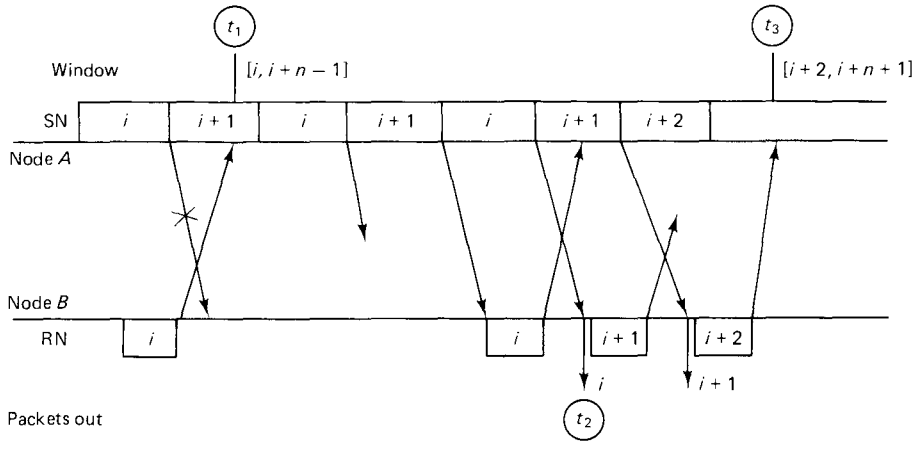

Figure 2.28 SN_{min} is i at time t_1. Packet i is then released to the higher layer at B at time t_2 and the window is increased at A at time t_3.

B (whether $t_1 < t_2$, or vice versa) transmits frames carrying $RN \geq i + 1$ from time t_2 until some such frame is received error-free at A at time t_3. Since node A is also transmitting frames in this interval, the delay between subsequent transmissions from B is finite, and, since $q > 0$, the interval from t_2 to t_3 is finite. Thus the interval from t_1 to t_3 is finite and the algorithm is live.

It will be observed that no assumption was made in the demonstration above about the frames traveling in order on the links, and the algorithm above operates correctly even when frames get out of order. When we look at error recovery at the transport layer, the role of a link will be played by a subnetwork and the role of a frame will be played by a packet. For datagram networks, packets can get out of order, so this generality will be useful.

Go back *n* with modulus *m* > *n*. It will now be shown that if the sequence number SN and the request number RN are sent modulo m, for some m strictly greater than the go back number n, the correctness of go back n is maintained as long as we reimpose the condition that frames do not get out of order on the links. To demonstrate this correctness, we first look more carefully at the ordering of events when ordinary integers are used for SN and RN.

Consider the transmission of an arbitrary frame from node A to B. Suppose that the frame is generated at time t_1 and received at t_2 (see Fig. 2.29). The sequence number SN of the frame must lie in node A's window at time t_1, so

$$SN_{min}(t_1) \leq SN \leq SN_{min}(t_1) + n - 1 \tag{2.24}$$

Also, as shown in the figure,

$$SN_{min}(t_1) \leq RN(t_2) \leq SN_{min}(t_1) + n \tag{2.25}$$

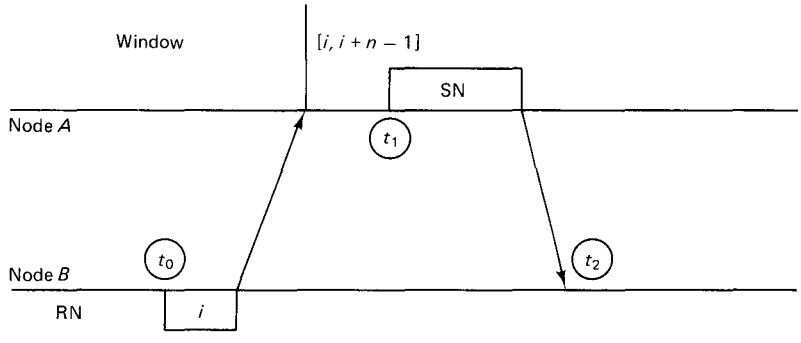

Figure 2.29 Let t_1 and t_2 be the times at which a given frame is generated at A and received at B respectively. The sequence number in the frame satisfies $SN_{min}(t_1) \leq SN \leq SN_{min}(t_1) + n - 1$. The value i of $SN_{min}(t_1)$ is equal to the last received value of RN, which is $RN(t_0) \leq RN(t_2)$. Thus $SN_{min}(t_1) \leq RN(t_2)$, which is the left side of Eq. (2.25). Conversely, no frame with sequence number $SN_{min}(t_1) + n$ can have been sent before t_1 since this value is beyond the upper limit of the window. Since frames travel in order on the link, no frame with this number has arrived at B before t_2, and $RN(t_2) \leq SN_{min}(t_1) + n$, which is the right side of Eq. (2.25).

We see from Eqs. (2.24) and (2.25) that SN and $RN(t_2)$ are both contained in the interval from $SN_{min}(t_1)$ to $SN_{min}(t_1) + n$, and thus must satisfy

$$|RN(t_2) - SN| \leq n \tag{2.26}$$

Now suppose that when packet number SN is sent, the accompanying sequence number is sent modulo m, and let sn denote SN mod m. Step 3 of the algorithm at node B must then be modified to: If an error-free frame is received from A containing a sequence number sn equal to RN mod m, release the received packet to the higher layer and increment RN. Since $m > n$ by assumption, we see from Eq. (2.26) that $sn = RN$ mod m will be satisfied if and only if the packet number SN is equal to RN; thus, the algorithm still works correctly.

Next consider the ordering of arriving request numbers (using ordinary integers) relative to the window at node A. From Fig. 2.30, we see that

$$SN_{min}(t_2) \leq RN \leq SN_{min}(t_2) + n \tag{2.27}$$

Now suppose that RN is sent modulo m, and let $rn = RN$ mod m. Step 4 of the algorithm at node A must then be modified to: If an error-free frame is received from B containing $rn \neq SN_{min}$ mod m, then increment SN_{min} until $rn = SN_{min}$ mod m. Because of the range of RN in Eq. (2.27), we see that this new rule is equivalent to the old rule, and it is sufficient to send request numbers modulo m. At this point, however, we see that it is unnecessary for SN_{min}, SN_{max}, and RN to be saved at nodes A and B as ordinary integers; everything can be numbered modulo m, and the algorithm has been demonstrated to work correctly for $m > n$.

For completeness, we restate the algorithm for operation modulo m; since all numbers are taken modulo m, we use capital letters for these numbers.

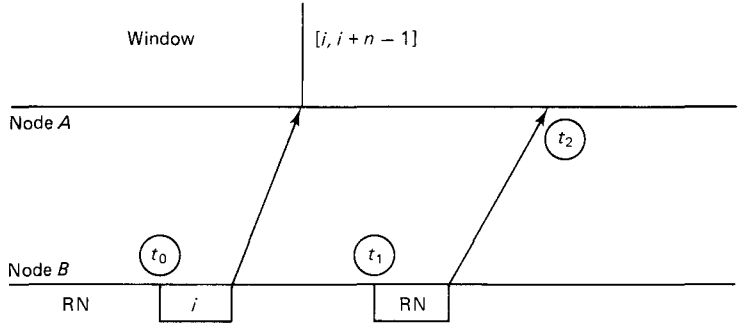

Figure 2.30 Let t_1 and t_2 be the times at which a given frame with request number RN is generated at B and received at A, respectively. Let $SN_{min}(t_2) = i$ be the lower edge of the window at t_2 and let t_0 be the generation time of the frame from B that caused the window at A to move to $[i, i + n - 1]$. Since the frames travel in order on the link, $t_0 < t_1$, so $i \leq RN$ and thus $SN_{min}(t_2) \leq RN$. Similarly, node A cannot have sent packet $i + n$ before t_2, so it certainly cannot have been received before t_1. Thus $RN \leq SN_{min}(t_2) + n$.

The go back n algorithm at node A for modulo m operation, $m > n$:

1. Set the modulo m variables SN_{min} and SN_{max} to 0.
2. Do steps 3, 4, and 5 repeatedly in any order. There can be an arbitrary but bounded delay between the time when the conditions for a step are satisfied and when the step is executed.
3. If $(SN_{max} - SN_{min}) \bmod m < n$, and if a packet is available from the higher layer, accept a new packet into the DLC, assign number SN_{max} to it, and increment SN_{max} to $(SN_{max} + 1) \bmod m$.
4. If an error-free frame is received from B containing a request number RN, and $(RN - SN_{min}) \bmod m \leq (SN_{max} - SN_{min}) \bmod m$, set SN_{min} to equal RN.
5. If $SN_{min} \neq SN_{max}$ and no frame is currently in transmission, choose some number SN such that $(SN - SN_{min}) \bmod m < (SN_{max} - SN_{min}) \bmod m$; transmit packet SN in a frame containing SN in the sequence number field.

The go back n algorithm at node B for modulo m operation, $m > n$:

1. Set the modulo m variable RN to 0.
2. Whenever an error-free frame is received from A containing a sequence number SN equal to RN, release the received packet to the higher layer and increment RN to $(RN + 1) \bmod m$.
3. At arbitrary times, but within bounded delay after receiving any error-free data frame from A, transmit a frame to A containing RN in the request number field.

Efficiency of go back *n* implementations Retransmissions, or delays waiting for time-outs, occur in go back n ARQ for the following three reasons: first, errors in the forward direction, second, errors in the feedback direction, and third, longer frames in the feedback than in the forward direction. These will be discussed in reverse order.

The likelihood of retransmissions caused by long reverse frames can be reduced by increasing the go back number n. Unfortunately, the normal value of modulus in the standard protocols is $m = 8$, which constrains n to be at most 7. Figure 2.31 illustrates that even in the absence of propagation delay, reverse frames more than three times the length of forward frames can cause retransmissions for $n = 7$. Problem 2.23 also shows that if frames have lengths that are exponentially distributed, with the same distribution

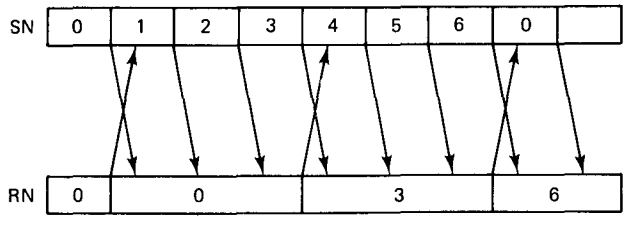

Figure 2.31 Go back 7 ARQ with long frames in the reverse direction. Note that the ack for packet 1 has not arrived at the sending side by the time packet 6 finishes transmission, thereby causing a retransmission of packet 0.

in each direction, the probability p that a frame is not acked by the time the window is exhausted is given by

$$p = (1 + n)2^{-n} \tag{2.28}$$

For $n = 7$, p is equal to 1/16. Frames normally have a maximum permissible length (and a minimum length because of the control overhead), so in practice p is somewhat smaller. However, links sometimes carry longer frames in one direction than in the other, and in this case, there might be considerable waste in link utilization for $n = 7$. When propagation delay is large relative to frame length (as, for example, on high-speed links and satellite links), this loss of utilization can be quite serious. Fortunately, the standard protocols have an alternative choice of modulus as $m = 128$.

When errors occur in a reverse frame, the acknowledgment information is postponed for an additional reverse frame. This loss of line utilization in one direction due to errors in the other can also be avoided by choosing n sufficiently large.

Finally, consider the effect of errors in the forward direction. If n is large enough to avoid retransmissions or delays due to large propagation delay and long frames or errors in the reverse direction, and if the sending DLC waits to exhaust its window of n packets before retransmitting, a large number of packets are retransmitted for each forward error. The customary solution to this problem is the use of time-outs. In its simplest version, if a packet is not acknowledged within a fixed time-out period after transmission, it is retransmitted. This time-out should be chosen long enough to include round-trip propagation and processing delay plus transmission time for two maximum-length packets in the reverse direction (one for the frame in transit when a packet is received, and one to carry the new RN). In a more sophisticated version, the sending DLC, with knowledge of propagation and processing delays, can determine which reverse frame should carry the ack for a given packet; it can go back if that frame is error free and fails to deliver the ack.

In a more fundamental sense, increasing link utilization and decreasing delay is achieved by going back quickly when a forward error occurs, but avoiding retransmissions caused by long frames and errors in the reverse direction. One possibility here is for the receiving DLC to send back a short supervisory frame upon receiving a frame in error. This allows the sending side to go back much sooner than if RN were simply piggybacked on a longer reverse data frame. Another approach is to insert RN in the trailer of the reverse frame, inserting it before the CRC and inserting it at the last moment, after the packet part of the frame has been sent. This cannot be done in the standard DLC protocols, but would have the effect of reducing the feedback delay by almost one frame length.

It is not particularly difficult to invent new ways of reducing both feedback delay and control overhead in ARQ strategies. One should be aware, however, that it is not trivial to ensure the correctness of such strategies. Also, except in special applications, improvements must be substantial to outweigh the advantages of standardization.

2.4.3 Selective Repeat ARQ

Even if unnecessary retransmissions are avoided, go back n protocols must retransmit at least one round-trip-delay worth of frames when a single error occurs in an awaited

packet. In many situations, the probability of one or more errors in a frame is 10^{-4} or less, and in this case, retransmitting many packets for each frame in error has little effect on efficiency. There are some communication links, however, for which small error probabilities per frame are very difficult to achieve, even with error correction in the modems. For other links (*e.g.*, high-speed links and satellite links), the number of frames transmitted in a round- trip delay time is very large. In both these cases, selective repeat ARQ can be used to increase efficiency.

The basic idea of selective repeat ARQ for data on a link from A to B is to accept out-of-order packets and to request retransmissions from A only for those packets that are not correctly received. There is still a go back number, or window size, n, specifying how far A can get ahead of RN, the lowest-numbered packet not yet correctly received at B.

Note that whatever ARQ protocol is used, only error-free frames can deliver packets to B, and thus, if p is the probability of frame error, the expected number η of packets delivered to B per frame from A to B is bounded by

$$\eta \leq 1 - p \tag{2.29}$$

As discussed later, this bound can be approached (in principle) by selective repeat ARQ; thus $1 - p$ is sometimes called the throughput of ideal selective repeat. Ideal go back n ARQ can similarly be defined as a protocol that retransmits the packets in one round-trip delay each time that a frame carrying the packet awaited by the receiving DLC is corrupted by errors. The throughput of this ideal is shown in Problem 2.26 to be

$$\eta \leq \frac{1 - p}{1 + p\beta} \tag{2.30}$$

where β is the expected number of frames in a round-trip delay interval. This indicates that the increase in throughput available with selective repeat is significant only when $p\beta$ is appreciable relative to 1.

The selective repeat algorithm at node A, for traffic from A to B, is the same as the go back n algorithm, except that (for reasons soon to be made clear) the modulus m must satisfy $m \geq 2n$. At node B, the variable RN has the same significance as in go back n; namely, it is the lowest-numbered packet (or the lowest-numbered packet modulo m) not yet correctly received. In the selective repeat algorithm, node B accepts packets anywhere in the range RN to $RN + n - 1$. The value of RN must be sent back to A as in go back n, either piggybacked on data frames or sent in separate frames. Usually, as discussed later, the feedback from node B to A includes not only the value of RN but also additional information about which packets beyond RN have been correctly received. In principle, the DLC layer still releases packets to the higher layer in order, so that the accepted out-of-order packets are saved until the earlier packets are accepted and released.

We now assume again that frames do not get out of order on the links and proceed to see why a larger modulus is required for selective repeat than for go back n. Assume that a frame is received at node B at some given time t_2 and that the frame was generated at node A at time t_1. If SN and RN are considered as integers, Eqs. (2.24) and (2.25)

are still valid, and we can conclude from them that the sequence number SN in the received frame must satisfy

$$RN(t_2) - n \leq SN \leq RN(t_2) + n - 1 \tag{2.31}$$

If sequence numbers are sent mod m, and if packets are accepted in the range $RN(t_2)$ to $RN(t_2) + n - 1$, it is necessary for node B to distinguish values of SN in the entire range of Eq. (2.31). This means that the modulus m must satisfy

$$m \geq 2n, \qquad \text{for selective repeat} \tag{2.32}$$

With this change, the correctness of this class of protocols follows as before. The real issue with selective repeat, however, is using it efficiently to achieve throughputs relatively close to the ideal $1 - p$. Note first that using RN alone to provide acknowledgment information is not very efficient, since if several frame errors occur in one round-trip delay period, node A does not find out about the second frame error until one round-trip delay after the first error is retransmitted. There are several ways of providing the additional acknowledgment information required by A. One is for B to send back the lowest j packet numbers that it has not yet received; j should be larger than $p\beta$ (the expected number of frame errors in a round-trip delay time), but is limited by the overhead required by the feedback. Another possibility is to send RN plus an additional k bits (for some constant k), one bit giving the ack/nak status of each of the k packets after RN.

Assume now that the return frames carry sufficient information for A to determine, after an expected delay of β frames, whether or not a packet was successfully received. The typical algorithm for A is then to repeat packets as soon as it is clear that the previous transmission contained errors; if A discovers multiple errors simultaneously, it retransmits them in the order of their packet numbers. When there are no requested retransmissions, A continues to send new packets, up to $SN_{max} - 1$. At this limit, node A can wait for some time-out or immediately go back to SN_{min} to transmit successive unacknowledged packets.

Node A acts like an ideal selective repeat system until it is forced to wait or go back from $SN_{max} - 1$. When this happens, however, the packet numbered SN_{min} must have been transmitted unsuccessfully about n/β times. Thus, by making n large enough, the probability of such a go back can be reduced to negligible value. There are two difficulties with very large values of n (assuming that packets must be reordered at the receiving DLC). The first is that storage must be provided at B for all of the accepted packets beyond RN. The second is that the large number of stored packets are all delayed waiting for RN.

The amount of storage provided can be reduced to $n - \beta$ without much harm, since whenever a go back from $SN_{max} - 1$ occurs, node A can send no new packets beyond $SN_{max} - 1$ for a round-trip delay; thus, it might as well resend the yet-unacknowledged packets from to $SN_{max} - \beta$ to $SN_{max} - 1$; this means that B need not save these packets. The value of n can also be reduced, without increasing the probability of go back, by retransmitting a packet several times in succession if the previous transmission contained errors. For example, Fig. 2.32 compares double retransmissions with single retransmissions for $n = 2\beta + 2$. Single retransmissions fail with probability p and cause β extra retrans-

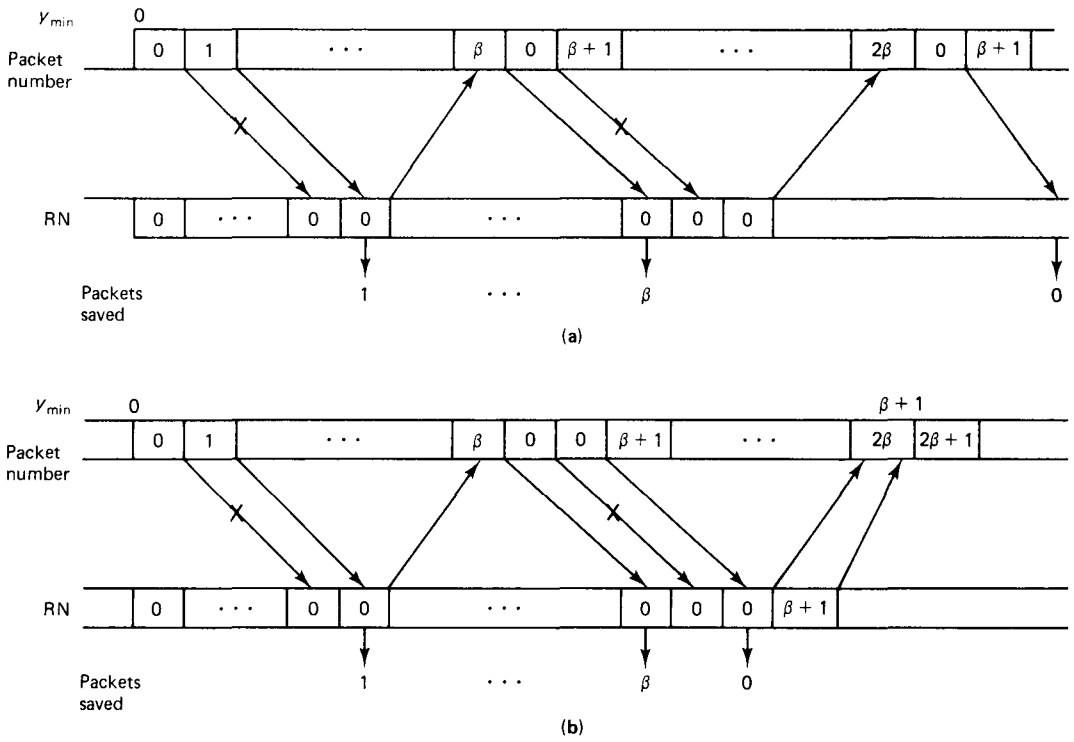

Figure 2.32 Selective repeat ARQ with $n = 2\beta + 2$ and receiver storage for $\beta + 1$ packets. (a) Note the wasted transmissions if a given packet (0) is transmitted twice with errors. (b) Note that this problem is cured, at the cost of one extra frame, if the second transmission of packet 0 is doubled. Feedback contains not only RN but additional information on accepted packets.

missions; double retransmissions rarely fail, but always require one extra retransmission. Thus, double retransmissions can increase throughput if $pn > 1$, but might be desirable in other cases to reduce the variability of delay. (See [Wel82] for further analysis.)

2.4.4 ARPANET ARQ

The ARPANET achieves efficiency by using eight stop-and-wait strategies in parallel, multiplexing the bit pipe between the eight. That is, each incoming packet is assigned to one of eight virtual channels, assuming that one of the eight is idle; if all the virtual channels are busy, the incoming packet waits outside the DLC (see Fig. 2.33). The busy virtual channels are multiplexed on the bit pipe in the sense that frames for the different virtual channels are sent one after the other on the link. The particular order in which frames are sent is not very important, but a simple approach is to send them in round-robin order. If a virtual channel's turn for transmission comes up before an ack has been received for that virtual channel, the packet is sent again, so that the multiplexing removes the need for any time-outs. (The actual ARPANET protocol, however, does use

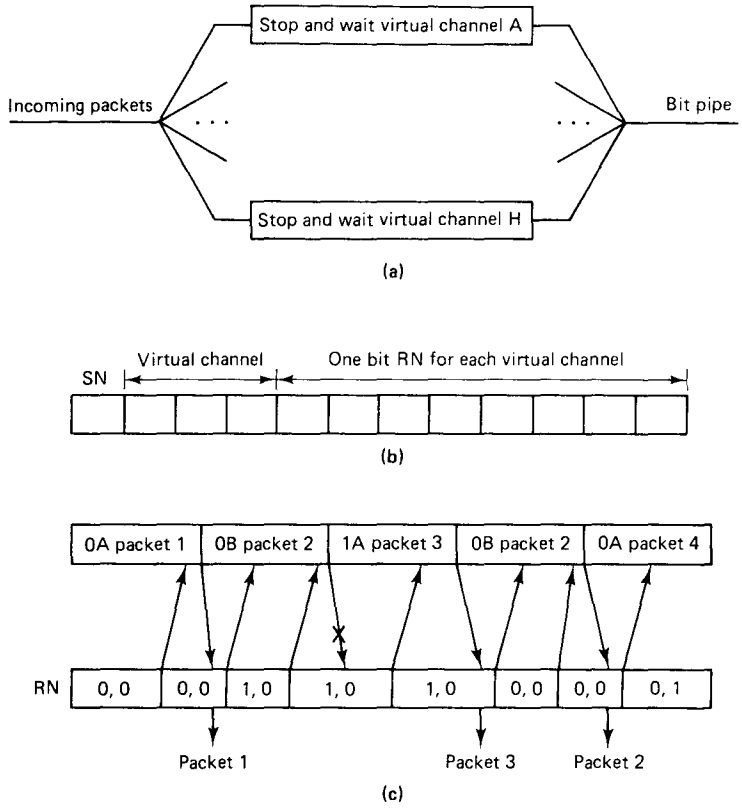

Figure 2.33 ARPANET ARQ. (a) Eight multiplexed, stop-and-wait virtual channels. (b) Bits in the header for ARQ control. (c) Operation of multiplexed stop and wait for two virtual channels. Top-to-bottom frames show SN and the channel number, and bottom-to-top frames show RN for both channels. The third frame from bottom to top acks packet 1 on the A channel.

time-outs.) When an ack is received for a frame on a given virtual channel, that virtual channel becomes idle and can accept a new packet from the higher layer.

Somewhat more overhead is required here than in the basic stop-and-wait protocol. In particular, each frame carries both the virtual channel number (requiring three bits) and the sequence number modulo 2 (*i.e.*, one bit) of the packet on that virtual channel. The acknowledgment information is piggybacked onto the frames going in the opposite direction. Each such frame, in fact, carries information for all eight virtual channels. In particular, an eight-bit field in the header of each return frame gives the number modulo 2 of the awaited packet for each virtual channel.

One of the desirable features of this strategy is that the ack information is repeated so often (*i.e.*, for all virtual channels in each return frame) that relatively few retransmissions are required because of transmission errors in the reverse direction. Typically, only one retransmission is required for each frame in error in the forward direction. The

undesirable feature of the ARPANET protocol is that packets are released to the higher layer at the receiving DLC in a different order from that of arrival at the sending DLC. The DLC layer could, in principle, reorder the packets, but since a packet on one virtual channel could be arbitrarily delayed, an arbitrarily large number of later packets might have to be stored. The ARPANET makes no effort to reorder packets on individual links, so this protocol is not a problem for ARPANET. We shall see later that the lack of ordering on links generates a number of problems at higher layers. Most modern networks maintain packet ordering for this reason, and consequently do not use this protocol despite its high efficiency and low overhead. For very poor communication links, where efficiency and overhead are very important, it is a reasonable choice.

2.5 FRAMING

The problem of framing is that of deciding, at the receiving DLC, where successive frames start and stop. In the case of a synchronous bit pipe, there is sometimes a period of idle fill between successive frames, so that it is also necessary to separate the idle fill from the frames. For an intermittent synchronous bit pipe, the idle fill is replaced by dead periods when no bits at all arrive. This does not simplify the problem since, first, successive frames are often transmitted with no dead periods in between, and second, after a dead period, the modems at the physical layer usually require some idle fill to reacquire synchronization.

There are three types of framing used in practice. The first, *character-based framing*, uses special communication control characters for idle fill and to indicate the beginning and ending of frames. The second, *bit-oriented framing with flags*, uses a special string of bits called a flag both for idle fill and to indicate the beginning and ending of frames. The third, *length counts*, gives the frame length in a field of the header. The following three subsections explain these three techniques, and the third also gives a more fundamental view of the problem. These subsections, except for a few comments, ignore the possibility of errors on the bit pipe. Section 2.5.4 then treats the joint problems of ARQ and framing in the presence of errors. Finally, Section 2.5.5 explains the trade-offs involved in the choice of frame length.

2.5.1 Character-Based Framing

Character codes such as ASCII generally provide binary representations not only for keyboard characters and terminal control characters, but also for various communication control characters. In ASCII, all these binary representations are seven bits long, usually with an extra parity bit which might or might not be stripped off in communication [since a cyclic redundancy check (CRC) can be used more effectively to detect errors in frames].

SYN (synchronous idle) is one of these communication control characters; a string of SYN characters provides idle fill between frames when a sending DLC has no data to send but a synchronous modem requires bits. SYN can also be used within frames,

sometimes for synchronization of older modems, and sometimes to bridge delays in supplying data characters. STX (start of text) and ETX (end of text) are two other communication control characters used to indicate the beginning and end of a frame, as shown in Fig. 2.34.

The character-oriented communication protocols used in practice, such as the IBM binary synchronous communication system (known as Bisynch or BSC), are far more complex than this, but our purpose here is simply to illustrate that framing presents no insurmountable problems. There is a slight problem in the example above in that either the header or the CRC might, through chance, contain a communication control character. Since these always appear in known positions after STX or ETX, this causes no problem for the receiver. If the packet to be transmitted is an arbitrary binary string, however, rather than a string of ASCII keyboard characters, serious problems arise; the packet might contain the ETX character, for example, which could be interpreted as ending the frame. Character-oriented protocols use a special mode of transmission, called *transparent mode*, to send such data.

The transparent mode uses a special control character called DLE (data link escape). A DLE character is inserted before the STX character to indicate the start of a frame in transparent mode. It is also inserted before intentional uses of communication control characters within such a frame. The DLE is not inserted before the possible appearances of these characters as part of the binary data. There is still a problem if the DLE character itself appears in the data, and this is solved by inserting an extra DLE before each appearance of DLE in the data proper. The receiving DLC then strips off one DLE from each arriving pair of DLEs, and interprets each STX or ETX preceded by an unpaired DLE as an actual start or stop of a frame. Thus, for example, DLE ETX (preceded by something other than DLE) would be interpreted as an end of frame, whereas DLE DLE ETX (preceded by something other than DLE) would be interpreted as an appearance of the bits corresponding to DLE ETX within the binary data.

With this type of protocol, the frame structure would appear as shown in Fig. 2.35. This frame structure is used in the ARPANET. It has two disadvantages, the first of which is the somewhat excessive use of framing overhead (*i.e.*, DLE STX precedes each frame, DLE ETX follows each frame, and two SYN characters separate each frame for a total of six framing characters per frame). The second disadvantage is that each frame must consist of an integral number of characters.

Let us briefly consider what happens in this protocol in the presence of errors. The CRC checks the header and packet of a frame, and thus will normally detect errors

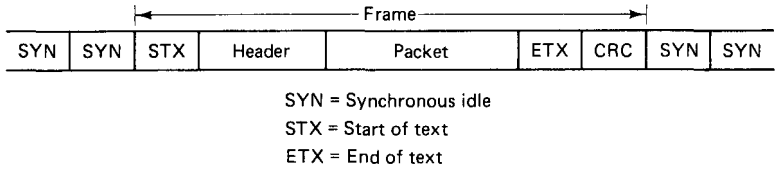

Figure 2.34 Simplified frame structure with character-based framing.

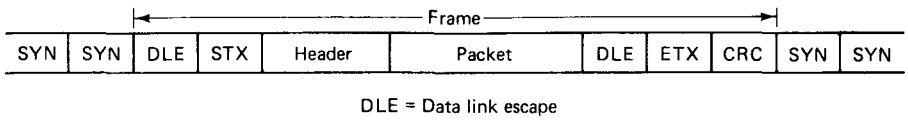

Figure 2.35 Character-based framing in a transparent mode as used in the ARPANET.

there. If an error occurs in the DLE ETX ending a frame, however, the receiver will not detect the end of the frame and thus will not check the CRC; this is why in the preceding section we assumed that frames could get lost. A similar problem is that errors can cause the appearance of DLE ETX in the data itself; the receiver would interpret this as the end of the frame and interpret the following bits as a CRC. Thus, we have an essentially random set of bits interpreted as a CRC, and the preceding data will be accepted as a packet with probability 2^{-L}, where L is the length of the CRC. The same problems occur in the bit-oriented framing protocols to be studied next and are discussed in greater detail in Section 2.5.4.

2.5.2 Bit-Oriented Framing: Flags

In the transparent mode for character-based framing, the special character pair DLE ETX indicated the end of a frame and was avoided within the frame by doubling each DLE character. Here we look at another approach, using a *flag* at the end of the frame. A flag is simply a known bit string, such as DLE ETX, that indicates the end of a frame. Similar to the technique of doubling DLEs, a technique called *bit stuffing* is used to avoid confusion between possible appearances of the flag as a bit string within the frame and the actual flag indicating the end of the frame. One important difference between bit-oriented and character based framing is that a bit-oriented frame can have any length (subject to some minimum and maximum) rather than being restricted to an integral number of characters. Thus, we must guard against appearances of the flag bit pattern starting in any bit position rather than just on character boundaries.

In practice, the flag is the bit string 01^60, where the notation 1^j means a string of j 1's. The rule used for bit stuffing is to insert (stuff) a 0 into the data string of the frame proper after each successive appearance of five 1's (see Fig. 2.36). Thus, the frame, after stuffing, never contains more than five consecutive 1's, and the flag at the end of the frame is uniquely recognizable. At the receiving DLC, the first 0 after each string of five consecutive 1's is deleted; if, instead, a string of five 1's is followed by a 1, the frame is declared to be finished.

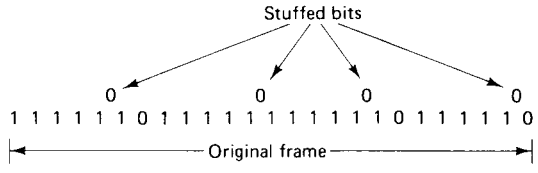

Figure 2.36 Bit stuffing. A 0 is stuffed after each consecutive five 1's in the original frame. A flag, 01111110, without stuffing, is sent at the end of the frame.

Bit stuffing has a number of purposes beyond eliminating flags within the frame. Standard DLCs have an abort capability in which a frame can be aborted by sending seven or more 1's in a row; in addition, a link is regarded as idle if 15 or more 1's in a row are received. What this means is that 01^6 is really the string denoting the end of a frame. If 01^6 is followed by a 0, it is the flag, indicating normal frame termination; if followed by a 1, it indicates abnormal termination. Bit stuffing is best viewed as preventing the appearance of 01^6 within the frame. One other minor purpose of bit stuffing is to break up long strings of 1's, which cause some older modems to lose synchronization.

It is easy to see that the bit stuffing rule above avoids the appearance of 01^6 within the frame, but it is less clear that so much bit stuffing is necessary. For example, consider the first stuffed bit in Fig. 2.36. Since the frame starts with six 1's (following a distinguishable flag), this could not be logically mistaken for 01^6. Thus, stuffing is not logically necessary after five 1's at the beginning of the frame (provided that the receiver's rule for deleting stuffed bits is changed accordingly).

The second stuffed bit in the figure is clearly necessary to avoid the appearance of 01^6. From a strictly logical standpoint, the third stuffed bit could be eliminated (except for the synchronization problem in older modems). Problem 2.31 shows how the receiver rule could be appropriately modified. Note that the reduction in overhead, by modifying the stuffing rules as above, is almost negligible; the purpose here is to understand the rules rather than to suggest changes.

The fourth stuffed bit in the figure is definitely required, although the reason is somewhat subtle. The original string 01^50 surrounding this stuffed bit could not be misinterpreted as 01^6, but the receiving DLC needs a rule to eliminate stuffed bits; it cannot distinguish a stuffed 0 following 01^5 from a data 0 following 01^5. Problem 2.32 develops this argument in detail.

There is nothing particularly magical about the string 01^6 as the bit string used to signal the termination of a frame (except for its use in preventing long strings of 1's on the link), and in fact, any bit string could be used with bit stuffing after the next-to-last bit of the string (see Problem 2.33). Such strings, with bit stuffing, are often useful in data communication to signal the occurrence of some rare event.

Consider the overhead incurred by using a flag to indicate the end of a frame. Assume that a frame (before bit stuffing and flag addition) consists of independent, identically distributed, random binary variables, with equal probability of 0 or 1. Assume for increased generality that the terminating signal for a frame is 01^j for some j (with 01^j0 being the flag and 01^{j+1} indicating abnormal termination); thus, $j = 6$ for the standard flag. An insertion will occur at (*i.e.*, immediately following) the i^{th} bit of the original frame (for $i \geq j$) if the string from $i - j + 1$ to i is 01^{j-1}; the probability of this is 2^{-j}. An insertion will also occur (for $i \geq 2j - 1$) if the string from $i - 2j + 2$ to i is 01^{2j-2}; the probability of this is 2^{-2j+1}. We ignore this term and the probability of insertions due to yet longer strings of 1's: first, because these probabilities are practically negligible, and second, because these insertions are used primarily to avoid long strings of 1's rather than to provide framing. Bit $j - 1$ in the frame is somewhat different than the other bits, since an insertion here occurs with probability 2^{-j+1} (*i.e.*, if the first $j - 1$ bits of the frame are all 1's).

Recall that the expected value of a sum of random variables is equal to the sum of the expected values (whether or not the variables are independent). Thus, the expected number of insertions in a frame of original length K is the sum, over i, of the expected number of insertions at each bit i of the frame. The expected number of insertions at a given bit, however, is just the probability of insertion there. Thus, the expected number of insertions in a string of length $K \geq j - 1$ is

$$(K - j + 3)2^{-j}$$

Taking the expected value of this over frame lengths K (with the assumption that all frames are longer than $j - 1$) and adding the $j + 1$ bits in the termination string, the expected overhead for framing becomes

$$E\{OV\} = (E\{K\} - j + 3)2^{-j} + j + 1 \tag{2.33}$$

Since $E\{K\}$ is typically very much larger than j, we have the approximation and upper bound (for $j \geq 3$)

$$E\{OV\} \leq E\{K\}2^{-j} + j + 1 \tag{2.34}$$

One additional bit is needed to distinguish a normal from an abnormal end of frame.

It will be interesting to find the integer value of j that minimizes this expression for a given value of expected frame length. As j increases from 1, the quantity on the right-hand side of Eq. (2.34) first decreases and then increases. Thus, the minimizing j is the smallest integer j for which the right-hand side is less than the same quantity with j increased by 1, that is, the smallest j for which

$$E\{K\}2^{-j} + j + 1 < E\{K\}2^{-j-1} + j + 2 \tag{2.35}$$

This inequality simplifies to $E\{K\}2^{-j-1} < 1$, and the smallest j that satisfies this is

$$j = \lfloor \log_2 E\{K\} \rfloor \tag{2.36}$$

where $\lfloor x \rfloor$ means the integer part of x. It is shown in Problem 2.34 that for this optimal value of j,

$$E\{OV\} \leq \log_2 E\{K\} + 2 \tag{2.37}$$

For example, with an expected frame length of 1000 bits, the optimal j is 9 and the expected framing overhead is less than 12 bits. For the standard flag, with $j = 6$, the expected overhead is about 23 bits (hardly cause for wanting to change the standard).

2.5.3 Length Fields

The basic problem in framing is to inform the receiving DLC where each idle fill string ends and where each frame ends. In principle, the problem of determining the end of an idle fill string is trivial; idle fill is represented by some fixed string (*e.g.*, repeated SYN characters or repeated flags) and idle fill stops whenever this fixed pattern is broken. In principle, one bit inverted from the pattern is sufficient, although in practice, idle fill is usually stopped at a boundary between flags or SYN characters.

Since a frame consists of an arbitrary and unknown bit string, it is somewhat harder to indicate where it ends. A simple alternative to flags or special characters is to include a length field in the frame header. DECNET, for example, uses this framing technique. Assuming no transmission errors, the receiving DLC simply reads this length in the header and then knows where the frame ends. If the length is represented by ordinary binary numbers, the number of bits in the length field has to be at least $\lfloor \log_2 K_{\max} \rfloor + 1$, where $K_{\max}$ is the maximum frame size. This is the overhead required for framing in this technique; comparing it with Eq. (2.37) for flags, we see that the two techniques require similar overhead.

Could any other method of encoding frame lengths require a smaller expected number of bits? This question is answered by information theory. Given any probability assignment $P(K)$ on frame lengths, the source coding theorem of information theory states that the minimum expected number of bits that can encode such a length is at least the entropy of that distribution, given by

$$H = \sum_K P(K) \log_2 \frac{1}{P(K)} \tag{2.38}$$

According to the theorem, at least this many bits of framing overhead, on the average, must be sent over the link per frame for the receiver to know where each frame ends. If $P(K) = 1/K_{\max}$, for $1 \leq K \leq K_{\max}$, then H is easily calculated to be $\log_2 K_{\max}$. Similarly, for a geometric distribution on lengths, with given $E\{K\}$, the entropy of the length distribution is approximately $\log_2 E\{K\} + \log_2 e$, for large $E\{K\}$. This is about 1/2 bit below the expression in Eq. (2.37). Thus, for the geometric distribution on lengths, the overhead using flags for framing is essentially minimum. The geometric distribution has an interesting extremal property; it can be shown to have the largest entropy of any probability distribution over the positive integers with given $E\{K\}$ (*i.e.*, it requires more bits than any other distribution).

The general idea of source coding is to map the more likely values of K into short bit strings and less likely values into long bit strings; more precisely, one would like to map a given K into about $\log_2[1/P(K)]$ bits. If one does this for a geometric distribution, one gets an interesting encoding known as the unary–binary encoding. In particular, for a given j, the frame length K is represented as

$$K = i2^j + r; \quad 0 \leq r < 2^j \tag{2.39}$$

The encoding for K is then i 0's followed by a 1 (this is known as a unary encoding of i) followed by the ordinary binary encoding of r (using j bits). For example, if $j = 2$ and $K = 7$, K is represented by $i = 1$, $r = 3$, which encodes into 0111 (where 01 is the unary encoding of $i = 1$ and 11 is the binary encoding of $r = 3$). Note that different values of K are encoded into different numbers of bits, but the end of the encoding can always be recognized as occurring j bits after the first 1.

In general, with this encoding, a given K maps into a bit string of length $\lfloor K/2^j \rfloor + 1 + j$. If the integer value above is neglected and the expected value over K is taken, then

$$E\{OV\} = E\{K\}2^{-j} + 1 + j \tag{2.40}$$

Note that this is the same as the flag overhead in Eq. (2.34). This is again minimized by choosing $j = \lfloor \log_2 E\{K\} \rfloor$. Thus, this unary–binary length encoding and flag framing both require essentially the minimum possible framing overhead for the geometric distribution, and no more overhead for any other distribution of given $E\{K\}$.

2.5.4 Framing with Errors

Several peculiar problems arise when errors corrupt the framing information on the communication link. First, consider the flag technique. If an error occurs in the flag at the end of a frame, the receiver will not detect the end of frame and will not check the cyclic redundancy check (CRC). In this case, when the next flag is detected, the receiver assumes the CRC to be in the position preceding the flag. This perceived CRC might be the actual CRC for the following frame, but the receiver interprets it as checking what was transmitted as two frames. Alternatively, if some idle fill follows the frame in which the flag was lost, the perceived CRC could include the error-corrupted flag. In any case, the perceived CRC is essentially a random bit string in relation to the perceived preceding frame, and the receiver fails to detect the errors with a probability essentially 2^{-L}, where L is the length of the CRC.

An alternative scenario is for an error within the frame to change a bit string into the flag, as shown for the flag 01^60:

$$0\,1\,0\,0\,1\,1\,0\,1\,1\,1\,0\,0\,1\ \ldots \qquad \text{(sent)}$$

$$0\,1\,0\,0\,1\,1\,1\,1\,1\,1\,0\,0\,1\ \ldots \qquad \text{(received)}$$

It is shown in Problem 2.35 that the probability of this happening somewhere in a frame of K independent equiprobable binary digits is approximately $(1/32)Kp$, where p is the probability of a bit error. In this scenario, as before, the bits before the perceived flag are interpreted by the receiver as a CRC, and the probability of accepting a false frame, given this occurrence, is 2^{-L}. This problem is often called the data sensitivity problem of DLC, since even though the CRC is capable of detecting any combination of three or fewer errors, a single error that creates or destroys a flag, plus a special combination of data bits to satisfy the perceived preceding CRC, causes an undetectable error.

If a length field in the header provides framing, an error in this length field again causes the receiver to look for the CRC in the wrong place, and again an incorrect frame is accepted with probability about 2^{-L}. The probability of such an error is smaller using a length count than using a flag (since errors can create false flags within the frame); however, after an error occurs in a length field, the receiver does not know where to look for any subsequent frames. Thus, if a length field is used for framing, some synchronizing string must be used at the beginning of a frame whenever the sending DLC goes back to retransmit. (Alternatively, synchronization could be used at the start of every frame, but this would make the length field almost redundant.)

There are several partial solutions to these problems, but none are without disadvantages. DECNET uses a fixed-length header for each frame and places the length of the frame in that header; in addition, the header has its own CRC. Thus, if an error

occurs in the length field of the header, the receiver can detect it by the header CRC, which is in a known position. One difficulty with this strategy is that the transmitter must still resynchronize after such an error, since even though the error is detected, the receiver will not know when the next frame starts. The other difficulty is that two CRCs must be used in place of one, which is somewhat inefficient.

A similar approach is to put the length field of one frame into the trailer of the preceding frame. This avoids the inefficiency of the DECNET approach, but still requires a special synchronizing sequence after each detected error. This also requires a special header frame to be sent whenever the length of the next frame is unknown when a given frame is transmitted.

Another approach, for any framing technique, is to use a longer CRC. This at least reduces the probability of falsely accepting a frame if framing errors occur. It appears that this is the most likely alternative to be adopted in practice; a standard 32 bit CRC exists as an option in standard DLCs.

A final approach is to regard framing as being at a higher layer than ARQ. In such a system, packets would be separated by flags, and the resulting sequence of packets and flags would be divided into fixed-length frames. Thus, frame boundaries and packet boundaries would bear no relation. If a packet ended in the middle of a frame and no further packets were available, the frame would be completed with idle fill. These frames would then enter the ARQ system, and because of the fixed-length frames, the CRC would always be in a known place. One disadvantage of this strategy is delay; a packet could not be accepted until the entire frame containing the end of the packet was accepted. This extra delay would occur on each link of the packet's path.

2.5.5 Maximum Frame Size

The choice of a maximum frame length, or maximum packet length, in a data network depends on many factors. We first discuss these factors for networks with variable packet lengths and then discuss some additional factors for networks using fixed packet lengths. Most existing packet networks use variable packet lengths, but the planners of broadband ISDN are attempting to standardize on a form of packet switching called asynchronous transfer mode (ATM) which uses very short frames (called cells in ATM) with a fixed length of 53 bytes. The essential reason for the fixed-length is to simplify the hardware for high-speed switching. There are also some advantages of fixed-length frames for multiaccess systems, as discussed in Chapter 4.

Variable frame length Assume that each frame contains a fixed number V of overhead bits, including frame header and trailer, and let K_{max} denote the maximum length of a packet. Assume, for the time being, that each message is broken up into as many maximum-length packets as possible, with the last packet containing what is left over. That is, a message of length M would be broken into $\lceil M/K_{max} \rceil$ packets, where $\lceil x \rceil$ is the smallest integer greater than or equal to x. The first $\lceil M/K_{max} \rceil - 1$ of these packets each contain K_{max} bits and the final packet contains between 1 and K_{max} bits.

The total number of bits in the resulting frames is then

$$\text{total bits} = M + \left\lceil \frac{M}{K_{max}} \right\rceil V \tag{2.41}$$

We see from this that as K_{max} decreases, the number of frames increases and thus the total overhead in the message, $\lceil M/K_{max} \rceil V$, increases. In the limit of very long messages, a fraction $V/(V + K_{max})$ of the transmitted bits are overhead bits. For shorter messages, the fraction of overhead bits is typically somewhat larger because of the reduced length of the final packet.

A closely related factor is that the nodes and external sites must do a certain amount of processing on a frame basis; as the maximum frame length decreases, the number of frames, and thus this processing load, increase. With the enormous increase in data rates available from optical fiber, it will become increasingly difficult to carry out this processing for small frame lengths. In summary, transmission and processing overhead both argue for a large maximum frame size.

We next discuss the many factors that argue for small frame size. The first of these other factors is the pipelining effect illustrated in Fig. 2.37. Assume that a packet must be completely received over one link before starting transmission over the next. If an entire message is sent as one packet, the delay in the network is the sum of the message transmission times over each link. If the message is broken into several packets, however, the earlier packets may proceed along the path while the later packets are still being transmitted on the first link, thus reducing overall message delay.

Since delay could be reduced considerably by starting the transmission of a packet on one link before receiving it completely on the preceding link, we should understand why this is not customarily done. First, if the DLC is using some form of ARQ, the CRC must be checked before releasing the packet to the next link. This same argument holds even if a CRC is used to discard packets in error rather than to retransmit them. Finally, if the links on a path have different data rates, the timing required to supply incoming bits to the outgoing link becomes very awkward; the interface between the DLC layer and the network layer also becomes timing dependent.

Let us investigate the combined effect of overhead and pipelining on message delay, assuming that each packet is completely received over one link before starting transmission on the next. Suppose that a message of length M is broken into maximum-length packets, with a final packet of typically shorter length. Suppose that the message must be transmitted over j equal-capacity links and that the network is lightly loaded, so that waiting at the nodes for other traffic can be ignored. Also, ignore errors on the links (which will be discussed later), and ignore propagation delays (which are independent of maximum packet length). The total time T required to transmit the message to the destination is the time it takes the first packet to travel over the first $j - 1$ links, plus the time it takes the entire message to travel over the final link (*i.e.*, when a nonfinal frame finishes traversing a link, the next frame is always ready to start traversing the link). Let C be the capacity of each link in bits per second, so that TC is the number of bit transmission times required for message delivery. Then, assuming that $M \geq K_{max}$,

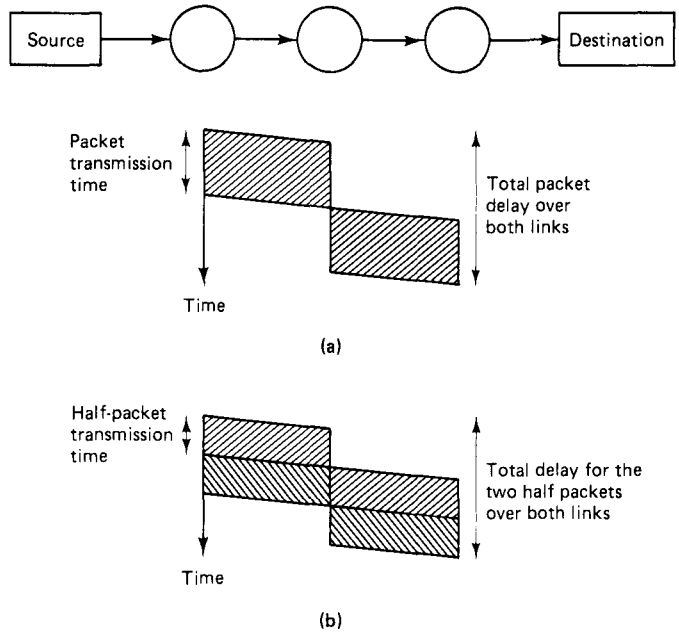

Figure 2.37 Decreasing delay by shortening packets to take advantage of pipelining. (a) The total packet delay over two empty links equals twice the packet transmission time on a link plus the overall propagation delay. (b) When each packet is split in two, a pipelining effect occurs. The total delay for the two half packets equals 1.5 times the original packet transmission time on a link plus the overall propagation delay.

$$TC = (K_{max} + V)(j - 1) + M + \left\lceil \frac{M}{K_{max}} \right\rceil V \tag{2.42}$$

In order to find the expected value of this over message lengths M, we make the approximation that $E\{\lceil M/K_{max} \rceil\} = E\{M/K_{max}\} + \frac{1}{2}$ (this is reasonable if the distribution of M is reasonably uniform over spans of K_{max} bits). Then

$$E\{TC\} \approx (K_{max} + V)(j - 1) + E\{M\} + \frac{E\{M\}V}{K_{max}} + \frac{V}{2} \tag{2.43}$$

We can differentiate this with respect to K_{max} (ignoring the integer constraint) to find the value of K_{max} that minimizes $E\{TC\}$. The result is

$$K_{max} \approx \sqrt{\frac{E\{M\}V}{j - 1}} \tag{2.44}$$

This shows the trade-off between overhead and pipelining. As the overhead V increases, K_{max} should be increased, and as the path length j increases, K_{max} should be reduced. As a practical detail, recall that delay is often less important for file transfers than for other messages, so file transfers should probably be weighted less than other

messages in the estimation of $E\{M\}$, thus arguing for a somewhat smaller K_{max} than otherwise.

As the loading in a network increases, the pipelining effect remains, although packets typically will have to queue up at the nodes before being forwarded. The effect of overhead becomes more important at heavier loads, however, because of the increased number of bits that must be transmitted with small packet sizes. On the other hand, there are several other effects at heavier loads that argue for small packet sizes. One is the "slow truck" effect. If many packets of widely varying lengths are traveling over the same path, the short packets will pile up behind the long packets because of the high transmission delay of the long packets on each link; this is analogous to the pileup of cars behind a slow truck on a single-lane road. This effect is analyzed for a single link in Section 3.5. The effect is more pronounced over a path, but is mathematically intractable.

Delay for stream-type traffic (such as voice) is quite different from delay for data messages. For stream-type traffic, one is interested in the delay from when a given bit enters the network until that bit leaves, whereas for message traffic, one is interested in the delay from arrival of the message to delivery of the complete message. Consider the case of light loading again and assume an arrival rate of R and a packet length K. The first bit in a packet is then held up for a time K/R waiting for the packet to be assembled. Assuming that the links along the path have capacities $C_1, C_2, \ldots$, each exceeding R, and assuming V bits of framing overhead, a given packet is delayed by $(K+V)/C_i$ on the ith link. When a given packet is completely received at the last node of the network, the first bit of the packet can be released immediately, yielding a total delay

$$T = \frac{K}{R} + (K+V)\sum_i \frac{1}{C_i} \tag{2.45}$$

Assuming that the received data stream is played out at rate R, all received bits have the same delay, which is thus given by Eq. (2.45). We have tacitly assumed in deriving this equation that $(K+V)/C_i \leq K/R_i$ for each link i (*i.e.*, that each link can transmit frames as fast as they are generated). If this is violated, the queueing delay becomes infinite, even with no other traffic in the network. We see then from Eq. (2.45) that T decreases as K decreases until $(K+V)/C_i = K/R$ for some link, and this yields the minimum possible delay. Packet lengths for stream traffic are usually chosen much larger than this minimum because of the other traffic that is expected on the links. As link speeds increase, however, the dominant delay term in Eq. (2.45) is K/R, which is unaffected by other traffic. For 64 kbps voice traffic, for example, packets usually contain on the order of 500 bits or less, since the delay from the K/R term starts to become objectionable for longer lengths.

Note that under light loading, the delay of a session (either message or stream) is controlled by the packet length of that session. Under heavy-loading conditions, however, the use of long packets by some users generally increases delay for all users. Thus a maximum packet length should be set by the network rather than left to the users.

Several effects of high variability in frame lengths on go back n ARQ systems were discussed in Section 2.4.2. High variability either increases the number of packets that must be retransmitted or increases waiting time. This again argues for small maximum

packet size. Finally, there is the effect of transmission errors. Large frames have a somewhat higher probability of error than small frames (although since errors are usually correlated, this effect is not as pronounced as one would think). For most links in use (with the notable exception of radio links), the probability of error on reasonable-sized frames is on the order of 10^{-4} or less, so that this effect is typically less important than the other effects discussed. Unfortunately, there are many analyses of optimal maximum frame length in the literature that focus only on this effect. Thus, these analyses are relevant only in those special cases where error probabilities are very high.

In practice, typical maximum frame lengths for wide area networks are on the order of 1 to a few thousand bits. Local area networks usually have much longer maximum frame lengths, since the path usually consists of a single multiaccess link. Also, delay and congestion are typically less important there and long frames allow most messages to be sent as a single packet.

Fixed frame length When all frames (and thus all packets) are required to be the same length, message lengths are not necessarily an integer multiple of the packet length, and the last packet of a message will have to contain some extra bits, called fill, to bring it up to the required length. Distinguishing fill from data at the end of a packet is conceptually the same problem as determining the end of a frame for variable-length frames. One effect of the fill in the last packet is an additional loss of efficiency, especially if the fixed packet length is much longer than many messages. Problem 2.39 uses this effect to repeat the optimization of Eq. (2.44), and as expected, the resulting optimal packet length is somewhat smaller with fixed frame lengths than with variable frame lengths. A considerably more important practical effect comes from the need to achieve a small delay for stream-type traffic. As was pointed out earlier, 64 kbps voice traffic must use packets on the order or 500 bits or less, and this requirement, for fixed frame length, forces *all* packets to have such short lengths. This is the primary reason why ATM (see Section 2.10) uses 53 byte frames even though very much longer frames would be desirable for most of the other traffic.

2.6 STANDARD DLCs

There are a number of similar standards for data link control, namely HDLC, ADCCP, LAPB, and SDLC. These standards (like most standards in the network field) are universally known by their acronyms, and the words for which the acronyms stand are virtually as cryptic as the acronyms themselves. HDLC was developed by the International Standards Organization (ISO), ADCCP by the American National Standards Institute (ANSI), LAPB by the International Consultative Committee on Telegraphy and Telephony (CCITT), and SDLC by IBM. HDLC and ADCCP are virtually identical and are described here. They have a wide variety of different options and modes of operation. LAPB is the DLC layer for X.25, which is the primary standard for connecting an external site to a subnet; LAPB is, in essence, a restricted subset of the HDLC options and modes, as will be described later. Similarly, SDLC, which was the precursor of HDLC and ADCCP, is essentially another subset of options and modes.