

2. Data Link Layer

Types of Errors

Whenever bits flow from one point to another, they are subject to unpredictable changes because of interference. This interference can change the shape of the signal. In a single-bit error, a 0 is changed to a 1 or a 1 to a 0. In a burst error, multiple bits are changed. For example, a 11100 s burst of impulse noise on a transmission with a data rate of 1200 bps might change all or some of the 12 bits of information.

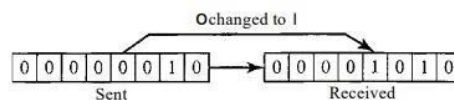
Single-Bit Error

The term single-bit error means that only 1 bit of a given data unit (such as a byte, character, or packet) is changed from 1 to 0 or from 0 to 1.

In a single-bit error, only 1 bit in the data unit has changed.

Figure 10.1 shows the effect of a single-bit error on a data unit. To understand the impact of the change, imagine that each group of 8 bits is an ASCII character with a 0 bit added to the left. In Figure 10.1, 00000010 (ASCII STX) was sent, meaning start of text, but 00001010 (ASCII LF) was received, meaning line feed. (For more information about ASCII code, see Appendix A.)

Figure 10.1 Single-bit error



Single-bit errors are the least likely type of error in serial data transmission. To understand why, imagine data sent at 1 Mbps. This means that each bit lasts only $1/1,000,000$ s, or 1 μ s. For a single-bit error to occur, the noise must have a duration of only 1 μ s, which is very rare; noise normally lasts much longer than this.

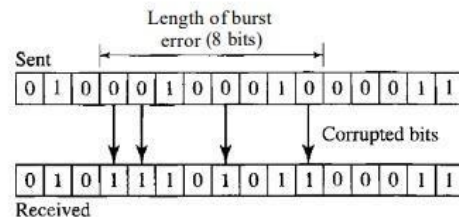
Burst Error

The term burst error means that 2 or more bits in the data unit have changed from 1 to 0 or from 0 to 1.

A burst error means that 2 or more bits in the data unit have changed.

Figure 10.2 shows the effect of a burst error on a data unit. In this case, 0100010001000011 was sent, but 0101110101100011 was received. Note that a burst error does not necessarily mean that the errors occur in consecutive bits. The length of the burst is measured from the first corrupted bit to the last corrupted bit. Some bits in between may not have been corrupted.

Figure 10.2 *Burst error of length 8*



A burst error is more likely to occur than a single-bit error. The duration of noise is normally longer than the duration of 1 bit, which means that when noise affects data, it affects a set of bits. The number of bits affected depends on the data rate and duration of noise. For example, if we are sending data at 1 kbps, a noise of 11100 s can affect 10 bits; if we are sending data at 1 Mbps, the same noise can affect 10,000 bits.

Redundancy

The central concept in detecting or correcting errors is redundancy. To be able to detect or correct errors, we need to send some extra bits with our data. These redundant bits are added by the sender and removed by the receiver. Their presence allows the receiver to detect or correct corrupted bits.

To detect or correct errors, we need to send **extra** (redundant) bits with data.

Detection Versus Correction

The correction of errors is more difficult than the detection. In error detection, we are looking only to see if any error has occurred. The answer is a simple yes or no. We are not even interested in the number of errors. A single-bit error is the same for us as a burst error.

In error correction, we need to know the exact number of bits that are corrupted and more importantly, their location in the message. The number of the errors and the size of the message are important factors. If we need to correct one single error in an 8-bit data unit, we need to consider eight possible error locations; if we need to correct two errors in a data unit of the same size, we need to consider 28 possibilities. You can imagine the receiver's difficulty in finding 10 errors in a data unit of 1000 bits.

Forward Error Correction Versus Retransmission

There are two main methods of error correction. Forward error correction is the process in which the receiver tries to guess the message by using redundant bits. This is possible, as we see later, if the number of errors is small. Correction by retransmission is a technique in which the receiver detects the occurrence of an error and asks the sender to resend the message. Resending is repeated until a message arrives that the receiver believes is error-free (usually, not all errors can be detected).

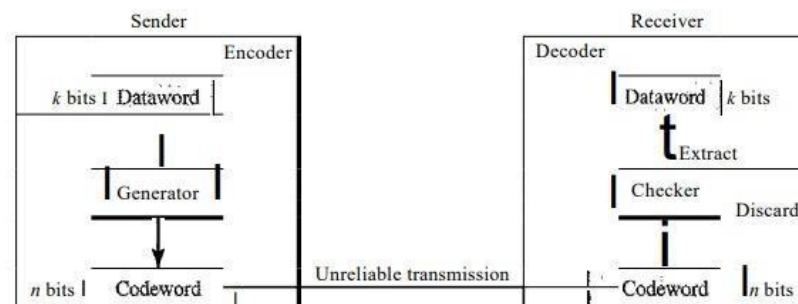
Error Detection

How can errors be detected by using block coding? If the following two conditions are met, the receiver can detect a change in the original codeword.

1. The receiver has (or can find) a list of valid codewords.
2. The original codeword has changed to an invalid one.

Figure 10.6 shows the role of block coding in error detection.

Figure 10.6 *Process of error detection in block coding*



The sender creates codewords out of datawords by using a generator that applies the rules and procedures of encoding (discussed later). Each codeword sent to the receiver may change during transmission. If the received codeword is the same as one of the valid codewords, the word is accepted; the corresponding dataword is extracted for use. If the received codeword is not valid, it is discarded. However, if the codeword is corrupted during transmission but the received word still matches a valid codeword, the error remains undetected. This type of coding can detect only single errors. Two or more errors may remain undetected.

Example 10.2

Let us assume that $k = 2$ and $n = 3$. Table 10.1 shows the list of datawords and codewords. Later, we will see how to derive a codeword from a dataword.

Table 10.1 A code for error detection (Example 10.2)

<i>Datawords</i>	<i>Codewords</i>
00	000
01	011
10	101
11	110

Assume the sender encodes the dataword 01 as 011 and sends it to the receiver. Consider the following cases:

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.

2. The codeword is corrupted during transmission, and 111 is received (the leftmost bit is corrupted). This is not a valid codeword and is discarded.

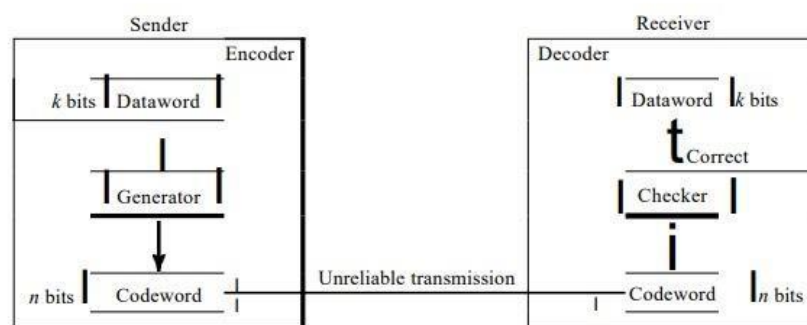
3. The codeword is corrupted during transmission, and 000 is received (the right two bits are corrupted). This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two corrupted bits have made the error undetectable.

An error-detecting code can detect only the types of errors for which it is designed;
other types of errors may remain undetected.

Error Correction

As we said before, error correction is much more difficult than error detection. In error detection, the receiver needs to know only that the received codeword is invalid; in error correction the receiver needs to find (or guess) the original codeword sent. We can say that we need more redundant bits for error correction than for error detection. Figure 10.7 shows the role of block coding in error correction. We can see that the idea is the same as error detection but the checker functions are much more complex.

Figure 10.7 Structure of encoder and decoder in error correction



Example 10.3

Let us add more redundant bits to Example 10.2 to see if the receiver can correct an error without knowing what was actually sent. We add 3 redundant bits to the 2-bit dataword to make 5-bit codewords. Again, later we will show how we chose the redundant bits. For the moment let us concentrate on the error correction concept. Table 10.2 shows the datawords and codewords.

Assume the dataword is 01. The sender consults the table (or uses an algorithm) to create the codeword 01011. The codeword is corrupted during transmission, and 01001 is received (error in the second bit from the right). First, the receiver finds that the received codeword is not in the table. This means an error has occurred. (Detection must come before correction.) The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword.

Table 10.2 *A code for error correction (Example 10.3)*

<i>Dataword</i>	<i>Codeword</i>
00	00000
01	01011
10	10101
11	11110

1. Comparing the received codeword with the first codeword in the table (01001 versus 00000), the receiver decides that the first codeword is not the one that was sent because there are two different bits.
2. By the same reasoning, the original codeword cannot be the third or fourth one in the table.
3. The original codeword must be the second one in the table because this is the only one that differs from the received codeword by 1 bit. The receiver replaces 01001 with 01011 and consults the table to find the dataword 01.

Cyclic Redundancy Check

We can create cyclic codes to correct errors. However, the theoretical background required is beyond the scope of this book. In this section, we simply discuss a category of cyclic codes called the cyclic redundancy check (CRC) that is used in networks such as LANs and WANs.

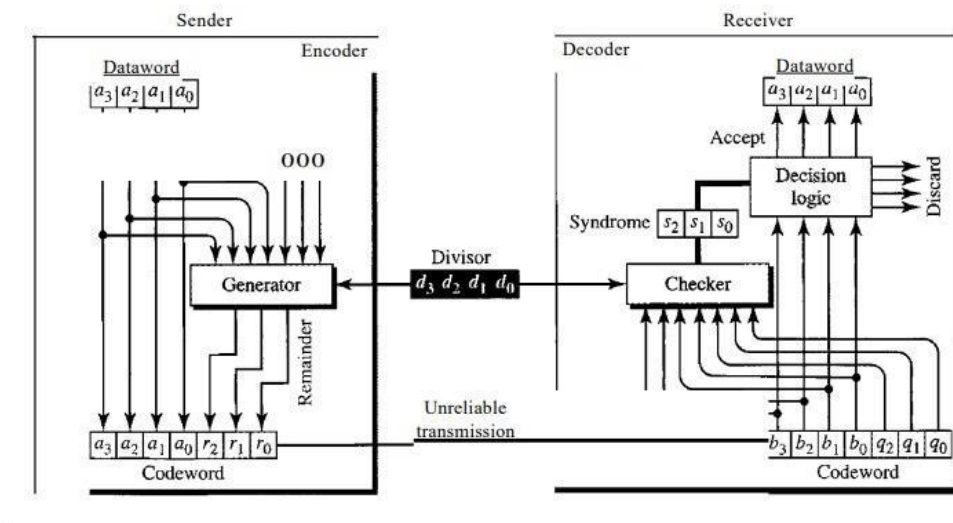
Table 10.6 shows an example of a CRC code. We can see both the linear and cyclic properties of this code.

Table 10.6 *A CRC code with $C(7, 4)$*

<i>Dataword</i>	<i>Codeword</i>	<i>Dataword</i>	<i>Codeword</i>
0000	0000000	1000	1000101
0001	0001011	1001	1001110
0010	0010110	1010	1010011
0011	0011101	1011	1011000
0100	0100111	1100	1100010
0101	0101100	1101	1101001
0110	0110001	1110	1110100
0111	0111010	1111	1111111

Figure 10.14 shows one possible design for the encoder and decoder.

Figure 10.14 CRC encoder and decoder



In the encoder, the dataword has k bits (4 here); the codeword has n bits (7 here). The size of the dataword is augmented by adding $n - k$ (3 here) 0s to the right-hand side of the word. The n -bit result is fed into the generator. The generator uses a divisor of size $n - k + 1$ (4 here), predefined and agreed upon. The generator divides the augmented dataword by the divisor (modulo-2 division). The quotient of the division is discarded; the remainder ($r_2r_1r_0$) is appended to the dataword to create the codeword.

The decoder receives the possibly corrupted codeword. A copy of all n bits is fed to the checker which is a replica of the generator. The remainder produced by the checker is a syndrome of $n - k$ (3 here) bits, which is fed to the decision logic analyzer. The analyzer has a simple function. If the syndrome bits are all as, the 4 leftmost bits of the codeword are accepted as the dataword (interpreted as no error); otherwise, the 4 bits are discarded (error).

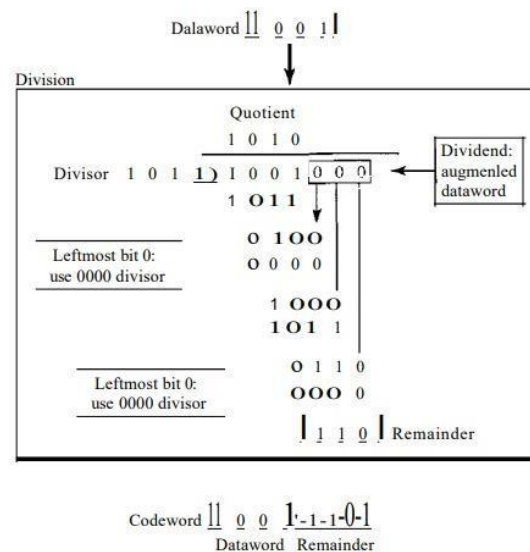
Encoder

Let us take a closer look at the encoder. The encoder takes the dataword and augments it with $n - k$ number of as. It then divides the augmented dataword by the divisor, as shown in Figure 10.15.

The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. However, as mentioned at the beginning of the chapter, in this case addition and subtraction are the same. We use the XOR operation to do both.

As in decimal division, the process is done step by step. In each step, a copy of the divisor is XORed with the 4 bits of the dividend. The result of the XOR operation (remainder) is 3 bits (in this case), which is used for the next step after 1 extra bit is pulled down to make it 4 bits long. There is one important point we need to remember in this type of division. If the leftmost bit of the dividend (or the part used in each step) is 0, the step cannot use the regular divisor; we need to use an all-0s divisor.

Figure 10.15 Division in CRC encoder

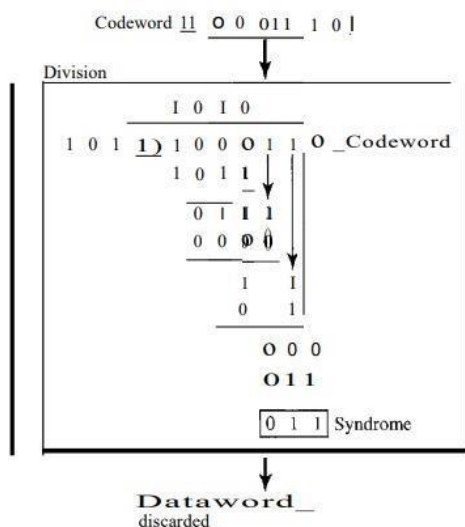


When there are no bits left to pull down, we have a result. The 3-bit remainder forms the check bits (r_2 , r_1 and r_0). They are appended to the dataword to create the codeword.

Decoder

The code word can change during transmission. The decoder does the same division process as the encoder. The remainder of the division is the syndrome. If the syndrome is all 0s, there is no error; the data word is separated from the received code word and accepted. Otherwise, everything is discarded. Figure 10.16 shows two cases: The left hand figure shows the value of syndrome when no error has occurred; the syndrome is 000. The right-hand part of the figure shows the case in which there is one single error. The syndrome is not all 0s (it is 011).

Figure 10.16 Division in the CRC decoder for two cases



Divisor

You may be wondering how the divisor+ 011 is chosen. Later in the chapter we present some criteria, but in general it involves abstract algebra.

Hardware Implementation

One of the advantages of a cyclic code is that the encoder and decoder can easily and cheaply be implemented in hardware by using a handful of electronic devices. Also, a hardware implementation increases the rate of check bit and syndrome bit calculation. In this section, we try to show, step by step, the process. The section, however, is optional and does not affect the understanding of the rest of the chapter.

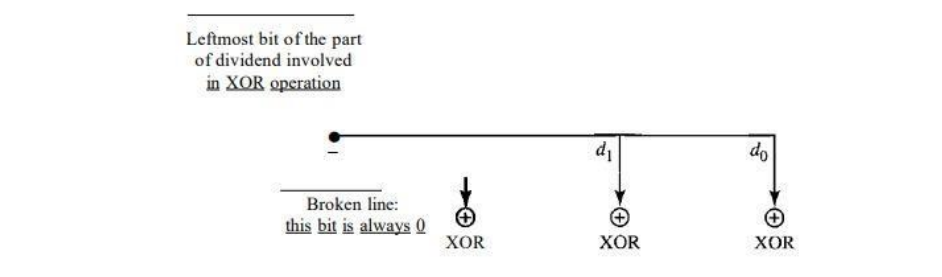
Divisor

Let us first consider the divisor. We need to note the following points:

1. The divisor is repeatedly XORed with part of the dividend.
2. The divisor has $n - k + 1$ bits which either are predefined or are all 0s. In other words, the bits do not change from one dataword to another. In our previous example, the divisor bits were either 1011 or 0000. The choice was based on the leftmost bit of the part of the augmented data bits that are active in the XOR operation.
3. A close look shows that only $n - k$ bits of the divisor is needed in the XOR operation. The leftmost bit is not needed because the result of the operation is always 0, no matter what the value of this bit. The reason is that the inputs to this XOR operation are either both 0s or both 1s. In our previous example, only 3 bits, not 4, is actually used in the XOR operation.

Using these points, we can make a fixed (hardwired) divisor that can be used for a cyclic code if we know the divisor pattern. Figure 10.17 shows such a design for our previous example. We have also shown the XOR devices used for the operation.

Figure 10.17 Hardwired design of the divisor in CRC



Note that if the leftmost bit of the part of dividend to be used in this step is 1, the divisor bits (d₂d₁d₀) are all; if the leftmost bit is 0, the divisor bits are 000. The design provides the right choice based on the leftmost bit.

Augmented Data word

In our paper-and-pencil division process in Figure 10.15, we show the augmented dataword as fixed in position with the divisor bits shifting to the right, 1 bit in each step. The divisor bits are aligned with the appropriate part of the augmented dataword. Now that our divisor is fixed, we need instead to shift the bits of the augmented dataword to the left (opposite direction) to align the divisor bits with the appropriate part. There is no need to store the augmented dataword bits.

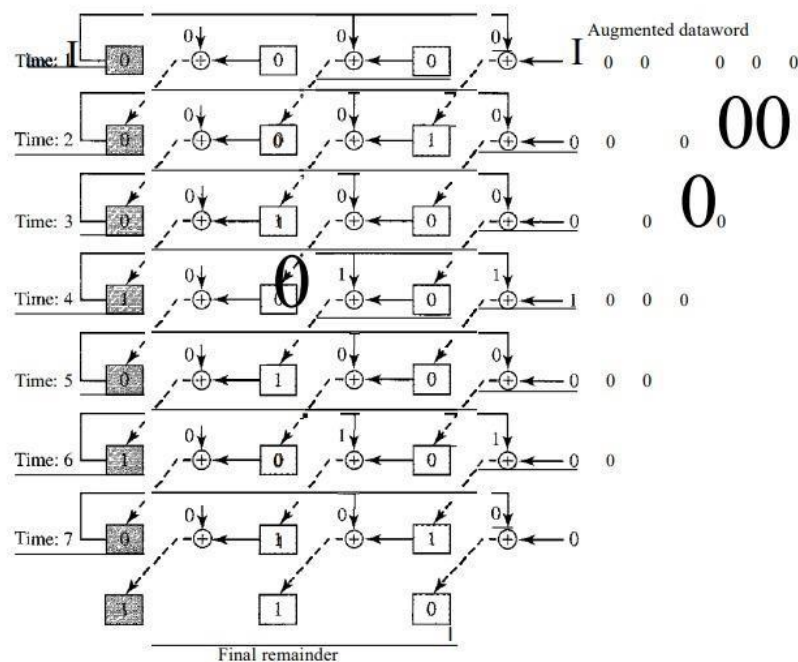
Remainder

In our previous example, the remainder is 3 bits ($n - k$ bits in general) in length. We can use three registers (single-bit storage devices) to hold these bits. To find the final remainder of the division, we need to modify our division process. The following is the step-by-step process that can be used to simulate the division process in hardware (or even in software).

1. We assume that the remainder is originally all 0s (000 in our example).
2. At each time click (arrival of 1 bit from an augmented dataword), we repeat the following two actions:
 - a. We use the leftmost bit to make a decision about the divisor (011 or 000).
 - b. The other 2 bits of the remainder and the next bit from the augmented dataword (total of 3 bits) are XORed with the 3-bit divisor to create the next remainder.

Figure 10.18 shows this simulator, but note that this is not the final design; there will be more improvements.

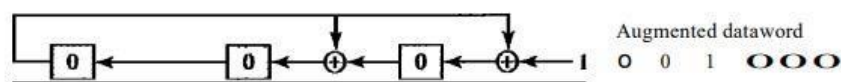
Figure 10.18 *Simulation of division in CRC encoder*



At each clock tick, shown as different times, one of the bits from the augmented dataword is used in the XOR process. If we look carefully at the design, we have seven steps here, while in the paper-and-pencil method we had only four steps. The first three steps have been added here to make each step equal and to make the design for each step the same. Steps 1, 2, and 3 push the first 3 bits to the remainder registers; steps 4, 5, 6, and 7 match the paper-and-pencil design. Note that the values in the remainder register in steps 4 to 7 exactly match the values in the paper-and-pencil design. The final remainder is also the same.

The above design is for demonstration purposes only. It needs simplification to be practical. First, we do not need to keep the intermediate values of the remainder bits; we need only the final bits. We therefore need only 3 registers instead of 24. After the XOR operations, we do not need the bit values of the previous remainder. Also, we do not need 21 XOR devices; two are enough because the output of an XOR operation in which one of the bits is 0 is simply the value of the other bit. This other bit can be used as the output. With these two modifications, the design becomes tremendously simpler and less expensive, as shown in Figure 10.19.

Figure 10.19 The CRC encoder design using shift registers

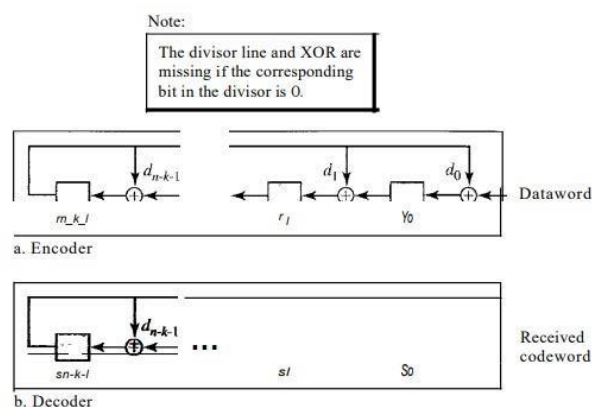


We need, however, to make the registers shift registers. A 1-bit shift register holds a bit for a duration of one clock time. At a time click, the shift register accepts the bit at its input port, stores the new bit, and displays it on the output port. The content and the output remain the same until the next input arrives. When we connect several 1-bit shift registers together, it looks as if the contents of the register are shifting.

General Design

A general design for the encoder and decoder is shown in Figure 10.20.

Figure 10.20 General design of encoder and decoder of a CRC code



Note that we have $n - k$ 1-bit shift registers in both the encoder and decoder. We have up to $n - k$ XOR devices, but the divisors normally have several 0s in their pattern, which reduces the number of devices. Also note that, instead of augmented datawords, we show the dataword itself as the input because after the bits in the dataword are all fed into the encoder, the extra bits, which all are 0s, do not have any effect on the right most XOR. Of course, the process needs to be continued for another $n - k$ steps before the check bits are ready. This fact is one of the criticisms of this design. Better schemes have been designed to eliminate this waiting time (the check bits are ready after k steps), but we leave this as a research topic for the reader. In the decoder, however, the entire codeword must be fed to the decoder before the syndrome is ready.

CHECKSUM

The last error detection method we discuss here is called the checksum. The checksum is used in the Internet by several protocols although not at the data link layer. However, we briefly discuss it here to complete our discussion on error checking.

Like linear and cyclic codes, the checksum is based on the concept of redundancy. Several protocols still use the checksum for error detection as we will see in future chapters, although the tendency is to replace it with a CRC. This means that the CRC is also used in layers other than the data link layer.

Idea

The concept of the checksum is not difficult. Let us illustrate it with a few examples.

Example 10.18

Suppose our data is a list of five 4-bit numbers that we want to send to a destination. In addition to sending these numbers, we send the sum of the numbers. For example, if the set of numbers is (7, 11, 12, 0, 6), we send (7, 11, 12, 0, 6, 36), where 36 is the sum of the original numbers. The receiver adds the five numbers and compares the result with the sum. If the two are the same, the receiver assumes no error, accepts the five numbers, and discards the sum. Otherwise, there is an error somewhere and the data are not accepted.

Example 10.19

We can make the job of the receiver easier if we send the negative (complement) of the sum, called the checksum. In this case, we send (7, 11, 12, 0, 6, -36). The receiver can add all the numbers received (including the checksum). If the result is 0, it assumes no error; otherwise, there is an error.

One's Complement

The previous example has one major drawback. All of our data can be written as a 4-bit word (they are less than 15) except for the checksum. One solution is to use one's complement arithmetic. In this arithmetic, we can represent unsigned numbers between 0 and $2^n - 1$ using only n bits. If the number has more than n bits, the extra leftmost bits need to be added to the n rightmost bits (wrapping). In one's complement arithmetic, a negative number can be represented by inverting all bits (changing a 0 to a 1 and a 1 to a 0). This is the same as subtracting the number from $2^n - 1$.

Example 10.20

How can we represent the number 21 in one's complement arithmetic using only four bits?

Although one's complement can represent both positive and negative numbers, we are concerned only with unsigned representation here.

Solution

The number 21 in binary is 10101 (it needs five bits). We can wrap the leftmost bit and add it to the four rightmost bits. We have $(0101 + 1) = 0110$ or 6.

Example 10.21

How can we represent the number -6 in one's complement arithmetic using only four bits?

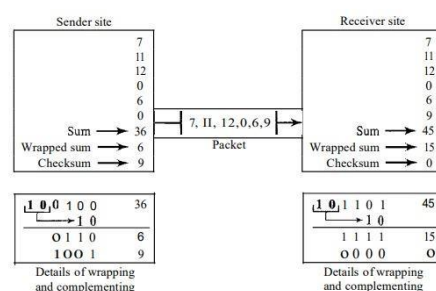
Solution

In one's complement arithmetic, the negative or complement of a number is found by inverting all bits. Positive 6 is 0110; negative 6 is 1001. If we consider only unsigned numbers, this is 9. In other words, the complement of 6 is 9. Another way to find the complement of a number in one's complement arithmetic is to subtract the number from $2n - 1$ ($16 - 1$ in this case).

Example 10.22

Let us redo Exercise 10.19 using one's complement arithmetic. Figure 10.24 shows the process at the sender and at the receiver. The sender initializes the checksum to 0 and adds all data items and the checksum (the checksum is considered as one data item and is shown in color). The result is 36. However, 36 cannot be expressed in 4 bits. The extra two bits are wrapped and added with the sum to create the wrapped sum value 6. In the figure, we have shown the details in binary. The sum is then complemented, resulting in the checksum value 9 ($15 - 6 = 9$). The sender now sends six data items to the receiver including the checksum 9. The receiver follows the same procedure as the sender. It adds all data items (including the checksum); the result is 45. The sum is wrapped and becomes 15. The wrapped sum is complemented and becomes 0. Since the value of the checksum is 0, this means that the data is not corrupted. The receiver drops the checksum and keeps the other data items. If the checksum is not zero, the entire packet is dropped.

Figure 10.24



Internet Checksum

Traditionally, the Internet has been using a 16-bit checksum. The sender calculates the checksum by following these steps.

Sender site:

1. The message is divided into 16-bit words.
 2. The value of the checksum word is set to 0.
 3. All words including the checksum are added using one's complement addition.
 4. The sum is complemented and becomes the checksum.
 5. The checksum is sent with the data.
-

The receiver uses the following steps for error detection.

Receiver site:

1. The message (including checksum) is divided into 16-bit words.
 2. All words are added using one's complement addition.
 3. The sum is complemented and becomes the new checksum.
 4. If the value of checksum is 0, the message is accepted; otherwise, it is rejected.
-

The nature of the checksum (treating words as numbers and adding and complementing them) is well-suited for software implementation. Short programs can be written to calculate the checksum at the receiver site or to check the validity of the message at the receiver site.

Example 10.23

Let us calculate the checksum for a text of 8 characters ("Forouzan"). The text needs to be divided into 2-byte (16-bit) words. We use ASCII (see Appendix A) to change each byte to a 2-digit hexadecimal number. For example, F is represented as 0x46 and 0 is represented as 0x0F. Figure 10.25 shows how the checksum is calculated at the sender and receiver sites. In part a of the figure, the value of partial sum for the first column is 0x36. We keep the rightmost digit (6) and insert the

Figure 10.25

1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	F	(ro)
7	5	7	A	luz)
6	1	6	E	(an)
0	0	0	0	Checksum (initial)
8	F	C	6	Sum (partial)
				1
8	F	C	7	Sum
7	0	3	8	Checksum (to send)

a. Checksum at the sender site

1	0	1	3	Carries
4	6	6	F	IFo)
7	2	6	F	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
7	0	3	8	Checksum (received)
F	F	F	E	Sum (partial)
				1
F	F	F	F	Sum
0	0	0	0	Checksum (new)

b. Checksum at the receiver site

leftmost digit (3) as the carry in the second column. The process is repeated for each column. Hexadecimal numbers are reviewed in Appendix B.

Note that if there is any corruption, the checksum recalculated by the receiver is not all as. We leave this an exercise.

Performance

The traditional checksum uses a small number of bits (16) to detect errors in a message of any size (sometimes thousands of bits). However, it is not as strong as the CRC in error-checking capability. For example, if the value of one word is incremented and the value of another word is decremented by the same amount, the two errors cannot be detected because the sum and checksum remain the same. Also if the values of several words are incremented but the total change is a multiple of 65535, the sum and the checksum does not change, which means the errors are not detected. Fletcher and Adler have proposed some weighted checksums, in which each word is multiplied by a number (its weight) that is related to its position in the text. This will eliminate the first problem we mentioned. However, the tendency in the Internet, particularly in designing new protocols, is to replace the checksum with a CRC.

FRAMING

Data transmission in the physical layer means moving bits in the form of a signal from the source to the destination. The physical layer provides bit synchronization to ensure that the sender and receiver use the same bit durations and timing.

The data link layer, on the other hand, needs to pack bits into frames, so that each frame is distinguishable from another. Our postal system practices a type of framing. The simple act of inserting a letter into an envelope separates one piece of information from another; the envelope serves as the delimiter. In addition, each envelope defines the sender and receiver addresses since the postal system is a many-to-many carrier facility.

Framing in the data link layer separates a message from one source to a destination, or from other messages to other destinations, by adding a sender address and a destination address. The destination address defines where the packet is to go; the sender address helps the recipient acknowledge the receipt.

Although the whole message could be packed in one frame, that is not normally done. One reason is that a frame can be very large, making flow and error control very inefficient. When a message is carried in one very large frame, even a single-bit error would require the retransmission of the whole message. When a message is divided into smaller frames, a single-bit error affects only that small frame.

Fixed-Size Framing

Frames can be of fixed or variable size. In fixed-size framing, there is no need for defining the

boundaries of the frames; the size itself can be used as a delimiter. An example of this type of framing is the ATM wide-area network, which uses frames of fixed size called cells. We discuss ATM in Chapter 18.

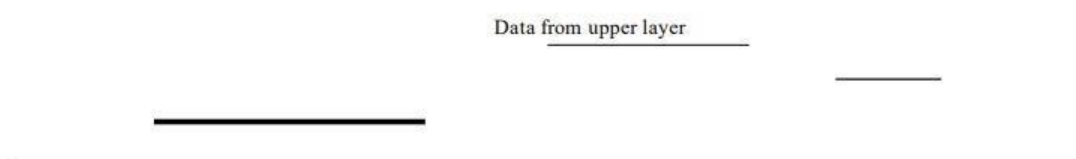
Variable-Size Framing

Our main discussion in this chapter concerns variable-size framing, prevalent in local area networks. In variable-size framing, we need a way to define the end of the frame and the beginning of the next. Historically, two approaches were used for this purpose: a character-oriented approach and a bit-oriented approach.

Character-Oriented Protocols

In a character-oriented protocol, data to be carried are 8-bit characters from a coding system such as ASCII (see Appendix A). The header, which normally carries the source and destination addresses and other control information, and the trailer, which carries error detection or error correction redundant bits, are also multiples of 8 bits. To separate one frame from the next, an 8-bit (1-byte) flag is added at the beginning and the end of a frame. The flag, composed of protocol-dependent special characters, signals the start or end of a frame. Figure 11.1 shows the format of a frame in a character-oriented protocol.

Figure 11.1 *A frame in a character-oriented protocol*

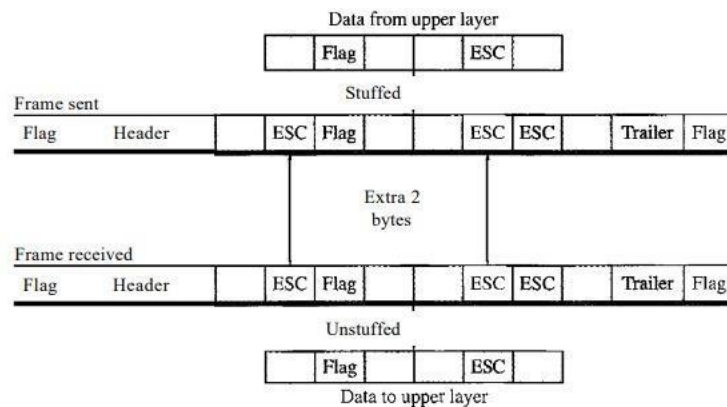


Character-oriented framing was popular when only text was exchanged by the data link layers. The flag could be selected to be any character not used for text communication. Now, however, we send other types of information such as graphs, audio, and video. Any pattern used for the flag could also be part of the information. If this happens, the receiver, when it encounters this pattern in the middle of the data, thinks it has reached the end of the frame. To fix this problem, a byte-stuffing strategy was added to character-oriented framing. In byte stuffing (or character stuffing), a special byte is added to the data section of the frame when there is a character with the same pattern as the flag.

The data section is stuffed with an extra byte. This byte is usually called the escape character (ESC), which has a predefined bit pattern. Whenever the receiver encounters the ESC character, it removes it from the data section and treats the next character as data, not a delimiting flag. Byte stuffing by the escape character allows the presence of the flag in the data section of the frame, but it creates another problem. What happens if the text contains one or more escape characters followed by a flag? The receiver removes the escape character, but keeps the flag, which is incorrectly interpreted as the end of the frame. To solve this problem, the escape characters that are

part of the text must also be marked by another escape character. In other words, if the escape character is part of the text, an extra one is added to show that the second one is part of the text. Figure 11.2 shows the situation.

Figure 11.2 *Byte stuffing and unstuffing*



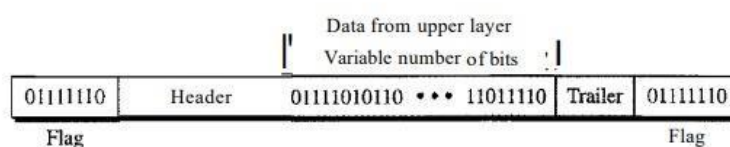
Byte stuffing is the process of adding 1 extra byte whenever there is a flag or escape character in the text.

Character-oriented protocols present another problem in data communications. The universal coding systems in use today, such as Unicode, have 16-bit and 32-bit characters that conflict with 8-bit characters. We can say that in general, the tendency is moving toward the bit-oriented protocols that we discuss next.

Bit-Oriented Protocols

In a bit-oriented protocol, the data section of a frame is a sequence of bits to be interpreted by the upper layer as text, graphic, audio, video, and so on. However, in addition to headers (and possible trailers), we still need a delimiter to separate one frame from the other. Most protocols use a special 8-bit pattern flag 01111110 as the delimiter to define the beginning and the end of the frame, as shown in Figure 11.3.

Figure 11.3 *A frame in a bit-oriented protocol*



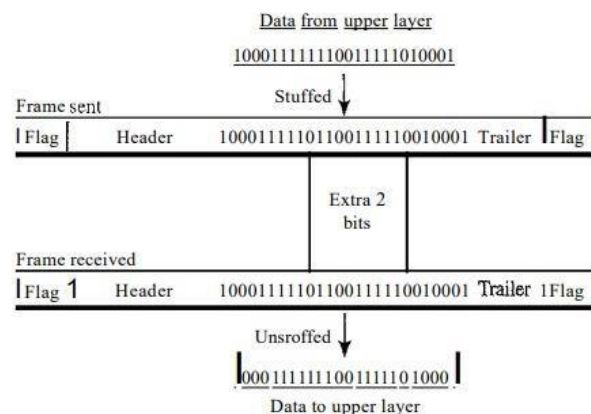
This flag can create the same type of problem we saw in the byte-oriented protocols. That is, if the flag pattern appears in the data, we need to somehow inform the receiver that this is not the end of the frame. We do this by stuffing 1 single bit (instead of 1 byte) to prevent the pattern

from looking like a flag. The strategy is called bit stuffing. In bit stuffing, if a 0 and five consecutive 1s are encountered, an extra 0 is added. This extra stuffed bit is eventually removed from the data by the receiver. Note that the extra bit is added after one 0 followed by five 1s regardless of the value of the next bit. This guarantees that the flag field sequence does not inadvertently appear in the frame.

Bit stuffing is the process of adding one extra 0 whenever five consecutive 1s follow a 0 in the data, so that the receiver does not mistake the pattern 0111110 for a flag.

Figure 11.4 shows bit stuffing at the sender and bit removal at the receiver. Note that even if we have a 0 after five 1s, we still stuff a 0. The 0 will be removed by the receiver.

Figure 11.4 *Bit stuffing and unstuffing*



This means that if the flaglike pattern 01111110 appears in the data, it will change to 011111010 (stuffed) and is not mistaken as a flag by the receiver. The real flag 01111110 is not stuffed by the sender and is recognized by the receiver.

FLOW AND ERROR CONTROL

Data communication requires at least two devices working together, one to send and the other to receive. Even such a basic arrangement requires a great deal of coordination for an intelligible exchange to occur. The most important responsibilities of the data link layer are flow control and error control. Collectively, these functions are known as data link control.

Flow Control

Flow control coordinates the amount of data that can be sent before receiving an acknowledgment and is one of the most important duties of the data link layer. In most protocols, flow control is a set of procedures that tells the sender how much data it can transmit before it must wait for an acknowledgment from the receiver. The flow of data must not be allowed to overwhelm the receiver. Any receiving device has a limited speed at which it can process incoming data and a limited amount of memory in which to store incoming data. The receiving device must be able to inform the

sending device before those limits are reached and to request that the transmitting device send fewer frames or stop temporarily. Incoming data must be checked and processed before they can be used. The rate of such processing is often slower than the rate of transmission. For this reason, each receiving device has a block of memory, called a buffer, reserved for storing incoming data until they are processed. If the buffer begins to fill up, the receiver must be able to tell the sender to halt transmission until it is once again able to receive.

Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.

Error Control

Error control is both error detection and error correction. It allows the receiver to inform the sender of any frames lost or damaged in transmission and coordinates the retransmission of those frames by the sender. In the data link layer, the term error control refers primarily to methods of error detection and retransmission. Error control in the data link layer is often implemented simply: Any time an error is detected in an exchange, specified frames are retransmitted. This process is called automatic repeat request (ARQ).

Error control in the data link layer is based on automatic repeat request, which is the retransmission of data.

NOISELESS CHANNELS

Let us first assume we have an ideal channel in which no frames are lost, duplicated, or corrupted. We introduce two protocols for this type of channel. The first is a protocol that does not use flow control; the second is the one that does. Of course, neither has error control because we have assumed that the channel is a perfect noiseless channel.

Simplest Protocol

Our first protocol, which we call the Simplest Protocol for lack of any other name, is one that has no flow or error control. Like other protocols we will discuss in this chapter, it is a unidirectional protocol in which data frames are traveling in only one direction—from the sender to receiver. We assume that the receiver can immediately handle any frame it receives with a processing time that is small enough to be negligible. The data link layer of the receiver immediately removes the header from the frame and hands the data packet to its network layer, which can also accept the packet immediately. In other words, the receiver can never be overwhelmed with incoming frames.

Design

There is no need for flow control in this scheme. The data link layer at the sender site gets data from its network layer, makes a frame out of the data, and sends it. The data link layer at the receiver site receives a frame from its physical layer, extracts data from the frame, and delivers the data to its

Figure 11.6 The design of the simplest protocol with no flow or error control

The diagram illustrates the design of the simplest protocol with no flow or error control. It shows the interaction between a Sender and a Receiver across three layers: Network, Data link, and Physical.

Sender Side:

- Network Layer:** Get data
- Data link Layer:** (Indicated by a downward arrow from the Network layer)
- Physical Layer:** Send frame

Receiver Side:

- Physical Layer:** Receive frame
- Data link Layer:** (Indicated by an upward arrow from the Physical layer)
- Network Layer:** Deliver data

Data Flow: Data frames are sent from the Sender's Physical layer to the Receiver's Physical layer, as indicated by the horizontal arrow labeled "Data frames".

Sender Algorithm:

- Event:** Request from network layer
- Repeat forever:** Algorithm for sender site

Receiver Algorithm:

- Repeat forever:** Algorithm for receiver site
- Event:** Notification from physical layer

Algorithms

Algorithm 11.1 Sender-site algorithm for the simplest protocol

```

1 while (true)                                // Repeat forever
2 {
3     WaitForEvent()i                          // Sleep until an event occurs
4     if(Event(RequestToSend»                 //There is a packet to send
5     {
6         GetData()i
7         MakeFrame()i
8         SendFrame()i                        //Send the frame
9     }
10 }

```

Analysis The algorithm has an infinite loop, which means lines 3 to 9 are repeated forever once the program starts. The algorithm is an event-driven one, which means that it sleeps (line 3) until an event wakes it up (line 4). This means that there may be an undefined span of time between the execution of line 3 and line 4; there is a gap between these actions. When the event, a request from the network layer, occurs, lines 6 through 8 are executed. The program then repeats the loop and again sleeps at line 3 until the next occurrence of the event. We have written pseudocode for the main process. We do not show any details for the modules GetData, MakeFrame, and SendFrame. GetDataO takes a data packet from the network layer, MakeFrameO adds a header and delimiter flags to the data packet to make a frame, and SendFrameO delivers the frame to the physical layer for transmission.

Algorithm 11.2 shows the procedure at the receiver site.

Algorithm 11.2 Receiver-site algorithm for the simplest protocol

```

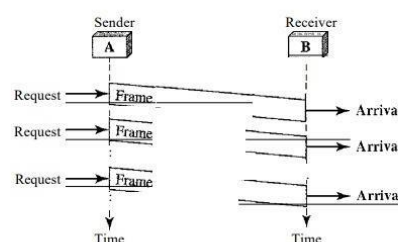
1  while(true)                                // Repeat forever
2  {
3      WaitForEvent()i                         // Sleep until an event occurs
4      if(Event(ArrivalNotification)»         //Data frame arrived
5      {
6          ReceiveFrame()i
7          ExtractData()i
8          DeliverData ()i                     //Deliver data to network layer
9      }
10 }
```

Analysis This algorithm has the same format as Algorithm 11.1, except that the direction of the frames and data is upward. The event here is the arrival of a data frame. After the event occurs, the data link layer receives the frame from the physical layer using the ReceiveFrameO process, extracts the data from the frame using the ExtractDataO process, and delivers the data to the network layer using the DeliverDataO process. Here, we also have an event-driven algorithm because the algorithm never knows when the data frame will arrive.

Example 11.1

Figure 11.7 shows an example of communication using this protocol. It is very simple. The sender sends a sequence of frames without even thinking about the receiver. To send three frames, three events occur at the sender site and three events at the receiver site. Note that the data frames are shown by tilted boxes; the height of the box defines the transmission time difference between the first bit and the last bit in the frame.

Figure 11.7 Flow diagram for Example 11.1



Stop-and-Wait Protocol

If data frames arrive at the receiver site faster than they can be processed, the frames must be stored until their use. Normally, the receiver does not have enough storage space, especially if it is receiving data from many sources. This may result in either the discarding of frames or denial of service. To prevent the receiver from becoming overwhelmed with frames, we somehow need to tell the sender to slow down. There must be feedback from the receiver to the sender.

The protocol we discuss now is called the Stop-and-Wait Protocol because the sender sends one frame, stops until it receives confirmation from the receiver (okay to go ahead), and then sends the next frame. We still have unidirectional communication for data frames, but auxiliary ACK frames (simple tokens of acknowledgment) travel from the other direction. We add flow control to our previous protocol.

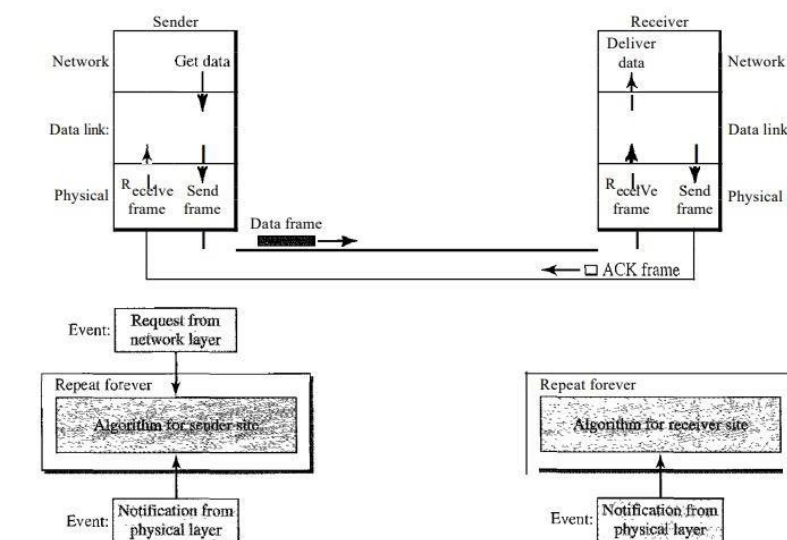
Design

Figure 11.8 illustrates the mechanism. Comparing this figure with Figure 11.6, we can see the traffic on the forward channel (from sender to receiver) and the reverse channel. At any time, there is either one data frame on the forward channel or one ACK frame on the reverse channel. We therefore need a half-duplex link.

Algorithms

Algorithm 11.3 is for the sender site.

Figure 11.8 Design of Stop-and-Wait Protocol



Algorithm 11.3 Sender-site algorithm for Stop-and-Wait Protocol

```

1 while (true)                                //Repeat forever
2   canSend = true                             //Allow the first frame to go
3   {
4     WaitForEvent();                          // Sleep until an event occurs
5     if(Event(RequestToSend) AND canSend)
6     {
7       GetData();
8       MakeFrame();
9       SendFrame();                          //Send the data frame
10      canSend = false;                      //cannot send until ACK arrives
11    }
12    WaitForEvent();                          // Sleep until an event occurs
13    if(Event(ArrivalNotification) // An ACK has arrived
14    {
15      ReceiveFrame();                       //Receive the ACK frame
16      canSend = true;
17    }
18  }

```

Analysis Here two events can occur: a request from the network layer or an arrival notification from the physical layer. The responses to these events must alternate. In other words, after a frame is sent, the algorithm must ignore another network layer request until that frame is acknowledged. We know that two arrival events cannot happen one after another because the channel is error-free and does not duplicate the frames. The requests from the network layer, however, may happen one after another without an arrival event in between. We need somehow to prevent the immediate sending of the data frame. Although there are several methods, we have used a simple canSend variable that can either be true or false. When a frame is sent, the variable is set to false to indicate that a new network request cannot be sent until canSend is true. When an ACK is received, canSend is set to true to allow the sending of the next frame.

Algorithm 11.4 shows the procedure at the receiver site.

Algorithm 11.4 Receiver-site algorithm for Stop-and-Wait Protocol

```

1 while (true)                                //Repeat forever
2   {
3     WaitForEvent();                          // Sleep until an event occurs
4     if(Event(ArrivalNotification)) //Data frame arrives
5     {
6       ReceiveFrame();
7       ExtractData();
8       Deliver(data);                       //Deliver data to network layer
9       SendFrame();                         //Send an ACK frame
10    }
11  }

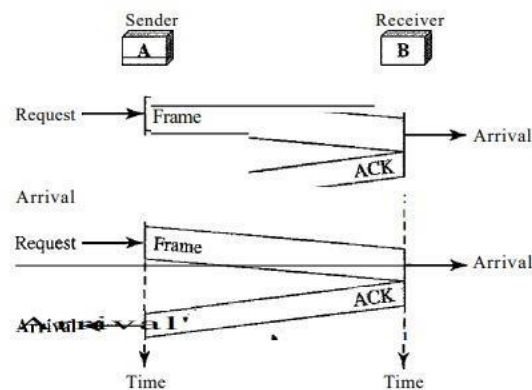
```

Analysis This is very similar to Algorithm 11.2 with one exception. After the data frame arrives, the receiver sends an ACK frame (line 9) to acknowledge the receipt and allow the sender to send the next frame.

Example 11.2

Figure 11.9 shows an example of communication using this protocol. It is still very simple. The sender sends one frame and waits for feedback from the receiver. When the ACK arrives, the sender sends the next frame. Note that sending two frames in the protocol involves the sender in four events and the receiver in two events.

Figure 11.9 Flow diagram for Example 11.2



NOISY CHANNELS

Although the Stop-and-Wait Protocol gives us an idea of how to add flow control to its predecessor, noiseless channels are nonexistent. We can ignore the error (as we sometimes do), or we need to add error control to our protocols. We discuss three protocols in this section that use error control.

Stop-and-Wait Automatic Repeat Request

Our first protocol, called the Stop-and-Wait Automatic Repeat Request (Stop-and-Wait ARQ), adds a simple error control mechanism to the Stop-and-Wait Protocol. Let us see how this protocol detects and corrects errors.

To detect and correct corrupted frames, we need to add redundancy bits to our data frame (see Chapter 10). When the frame arrives at the receiver site, it is checked and if it is corrupted, it is silently discarded. The detection of errors in this protocol is manifested by the silence of the receiver.

Lost frames are more difficult to handle than corrupted ones. In our previous protocols, there was no way to identify a frame. The received frame could be the correct one, or a duplicate, or a frame out of order. The solution is to number the frames. When the receiver receives a data frame that is out of order, this means that frames were either lost or duplicated.

The corrupted and lost frames need to be resent in this protocol. If the receiver does not respond when there is an error, how can the sender know which frame to resend? To remedy this problem, the sender keeps a copy of the sent frame. At the same time, it starts a timer. If the timer

expires and there is no ACK for the sent frame, the frame is resent, the copy is held, and the timer is restarted. Since the protocol uses the stop-and-wait mechanism, there is only one specific frame that needs an ACK even though several copies of the same frame can be in the network.

Error correction in Stop-and-Wait ARQ is done by keeping a copy of the sent frame and retransmitting of the frame when the timer expires.

Since an ACK frame can also be corrupted and lost, it too needs redundancy bits and a sequence number. The ACK frame for this protocol has a sequence number field. In this protocol, the sender simply discards a corrupted ACK frame or ignores an out-of-order one.

Sequence Numbers

As we discussed, the protocol specifies that frames need to be numbered. This is done by using sequence numbers. A field is added to the data frame to hold the sequence number of that frame.

One important consideration is the range of the sequence numbers. Since we want to minimize the frame size, we look for the smallest range that provides unambiguous communication. The sequence numbers of course can wrap around. For example, if we decide that the field is m bits long, the sequence numbers start from 0, go to $2^m - 1$, and then are repeated.

Let us reason out the range of sequence numbers we need. Assume we have used x as a sequence number; we only need to use $x + 1$ after that. There is no need for $x + 2$. To show this, assume that the sender has sent the frame numbered x . Three things can happen.

1. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next frame numbered $x + 1$.
2. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the frame (numbered x) after the time-out. Note that the frame here is a duplicate. The receiver can recognize this fact because it expects frame $x + 1$ but frame x was received.
3. The frame is corrupted or never arrives at the receiver site; the sender resends the frame (numbered x) after the time-out.

We can see that there is a need for sequence numbers x and $x + 1$ because the receiver needs to distinguish between case 1 and case 2. But there is no need for a frame to be numbered $x + 2$. In case 1, the frame can be numbered x again because frames x and $x + 1$ are acknowledged and there is no ambiguity at either site. In cases 2 and 3, the new frame is $x + 1$, not $x + 2$. If only x and $x + 1$ are needed, we can let $x = 0$ and $x + 1 = 1$. This means that the sequence is 0, 1, 0, 1, 0, and so on. Is this pattern familiar? This is modulo-2 arithmetic as we saw in Chapter 10.

In Stop-and-Wait ARQ, we use sequence numbers to number the frames.
The sequence numbers are based on modulo-2 arithmetic.

Acknowledgment Numbers

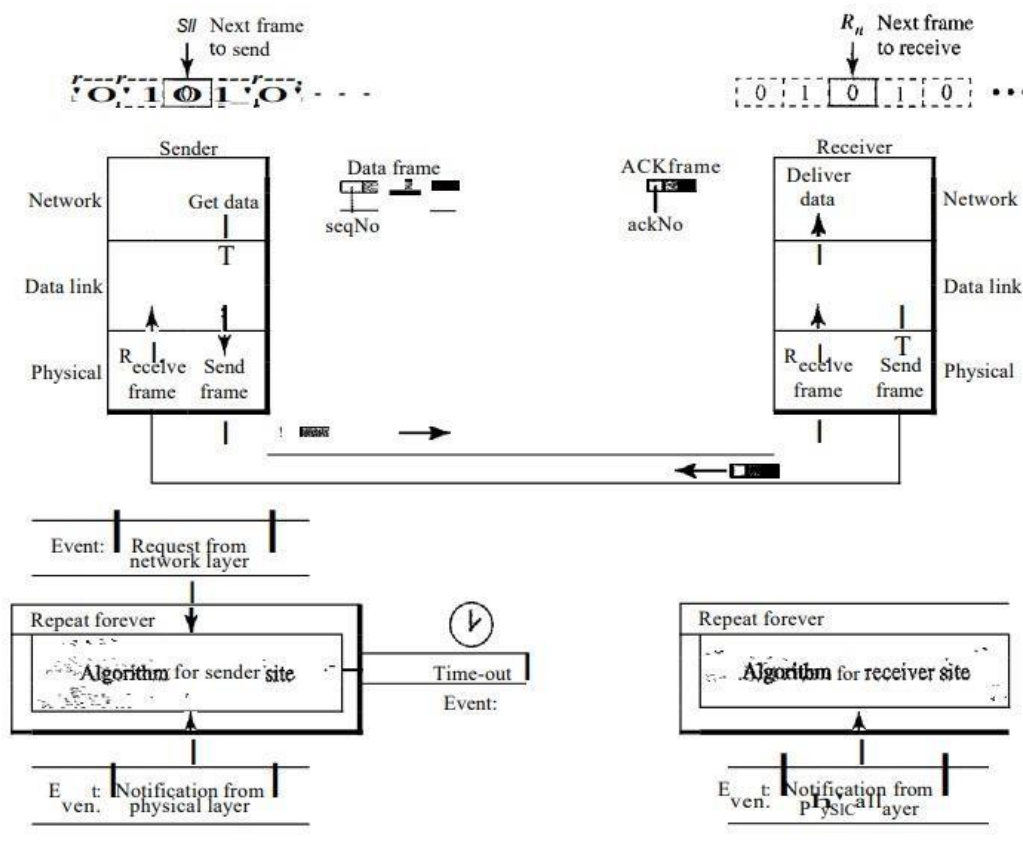
Since the sequence numbers must be suitable for both data frames and ACK frames, we use this convention: The acknowledgment numbers always announce the sequence number of the next frame expected by the receiver. For example, if frame 0 has arrived safe and sound, the receiver sends an ACK frame with acknowledgment 1 (meaning frame 1 is expected next). If frame 1 has arrived safe and sound, the receiver sends an ACK frame with acknowledgment 0 (meaning frame 0 is expected).

In **Stop-and-Wait ARQ**, the acknowledgment number always announces in
modul0-2 arithmetic the sequence number of the next frame expected.

Design

Figure 11.10 shows the design of the Stop-and-Wait ARQ Protocol. The sending device keeps a copy of the last frame transmitted until it receives an acknowledgment for that frame. A data frames uses a seqNo (sequence number); an ACK frame uses an ackNo (acknowledgment number). The sender has a control variable, which we call Sn (sender, next frame to send), that holds the sequence number for the next frame to be sent (0 or 1).

Figure 11.10 *Design of the Stop-and-Wait ARQ Protocol*



The receiver has a control variable, which we call Rn (receiver, next frame expected), that holds the number of the next frame expected. When a frame is sent, the value of Sn is incremented (modulo-2), which means if it is 0, it becomes 1 and vice versa. When a frame is received, the value of Rn is incremented (modulo-2), which means if it is 0, it becomes 1 and vice versa. Three events can happen at the sender site; one event can happen at the receiver site. Variable Sn points to the slot that matches the sequence number of the frame that has been sent, but not acknowledged; Rn points to the slot that matches the sequence number of the expected frame.

Algorithms

Algorithm 11.5 is for the sender site.

Algorithm 11.5 Sender-site algorithm for Stop-and-Wait ARQ

```

1  n = 0;                                // Frame 0 should be sent first
2  anSend = true;                         // Allow the first request to go
3  hile(true)                             // Repeat forever
4  {
5    WaitForEvent();                      // Sleep until an event occurs

6    if(Event(RequestToSend) AND canSend)
7    {
8      GetData();
9      MakeFrame(Sn);                     //The seqNo is Sn
10     StoreFrame(Sn);                    //Keep copy
11     SendFrame(Sn);
12     StartTimerO;
13     Sn = Sn + 1;
14     canSend = false;
15   }
16   WaitForEvent();                      // Sleep
17   if(Event(ArrivaINotification)        // An ACK has arrived
18   {
19     ReceiveFrame(ackNo);                //Receive the ACE fram
20     if(not corrupted AND ackNo == Sn) //Valid ACK
21     {
22       Stoptimer{};
23       PurgeFrame(Sn_1);                 //Copy is not needed
24       canSend = true;
25     }
26   }
27
28   if(Event(TimeOUT)                     // The timer expired
29   {
30     StartTimer();
31     ResendFrame(Sn_1);                 //Resend a copy check
32   }
33 }
```

Analysis We first notice the presence of S_n the sequence number of the next frame to be sent. This variable is initialized once (line 1), but it is incremented every time a frame is sent (line 13) in preparation for the next frame. However, since this is modulo-2 arithmetic, the sequence numbers are 0, 1, 0, 1, and so on. Note that the processes in the first event (SendFrame, StoreFrame, and Purge Frame) use an S_n defining the frame sent out. We need at least one buffer to hold this frame until we are sure that it is received safe and sound. Line 10 shows that before the frame is sent, it is stored. The copy is used for resending a corrupt or lost frame. We are still using the can Send variable to prevent the network layer from making a request before the previous frame is received safe and sound. If the frame is not corrupted and the ackNo of the ACK frame matches the sequence number of the next frame to send, we stop the timer and purge the copy of the data frame we saved. Otherwise, we just ignore this event and wait for the next event to happen. After each frame is sent, a timer is started. When the timer expires (line 28), the frame is resent and the timer is restarted.

Algorithm 11.6 shows the procedure at the receiver site.

Algorithm 11.6 Receiver-site algorithm for Stop-and-Wait ARQ Protocol

```

1  = 0;                                // Frame 0 expected to arrive first
2  hile(true)
3  {
4  WaitForEvent();                      // Sleep until an event occurs

5  if(Event(ArrivalNotification))      //Data frame arrives
6  {
7      ReceiveFrame(i)
8      if(corrupted(frame))i
9          sleep() i
10     if(seqNo ==  $R_n$ )                //Valid data frame
11     {
12         ExtractData();
13         DeliverData(i)              //Deliver data
14          $R_n = R_n + 1$ ;
15     }
16     SendFrame( $R_n$ );                //Send an ACK
17 }
18 }
```

Analysis This is noticeably different from Algorithm 11.4. First, all arrived data frames that are corrupted are ignored. If the seq No of the frame is the one that is expected (R_n), the frame is accepted, the data are delivered to the network layer, and the value of R_n is incremented. However, there is one subtle point here. Even if the sequence number of the data frame does not match the next frame expected, an ACK is sent to the sender. This ACK, however, just reconfirms the previous ACK instead of confirming the frame received. This is done because the receiver assumes that the previous ACK might have been lost; the receiver is sending a duplicate frame. The resent ACK may solve the problem before the time-out does it.

Example 11.3

Figure 11.11 shows an example of Stop-and-Wait ARQ. Frame 0 is sent and acknowledged. Frame 1 is lost and resent after the time-out. The resent frame 1 is acknowledged and the timer stops. Frame 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the frame or the acknowledgment is lost, so after the time-out, it resends frame 0, which is acknowledged.

Efficiency

The Stop-and-Wait ARQ discussed in the previous section is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth; by long, we mean the round-trip delay is long. The product of these two is called the bandwidth delay product, as we discussed in Chapter 3. We can think of the channel as a pipe. The bandwidth-delay product then is the volume of the pipe in bits. The pipe is always there. If we do not use it, we are inefficient. The bandwidth-delay product is a measure of the number of bits we can send out of our system while waiting for news from the receiver.

Example 11.4

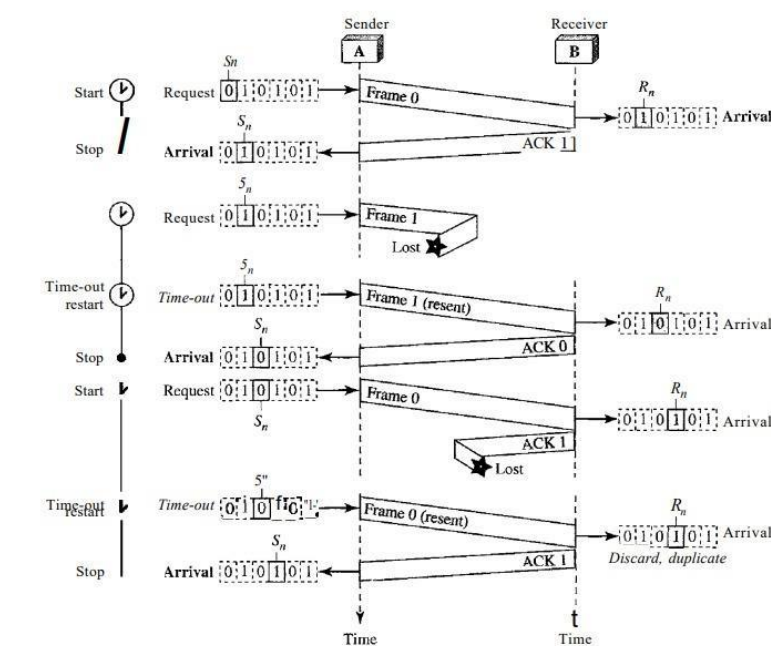
Assume that, in a Stop-and-Wait ARQ system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 ms to make a round trip. What is the bandwidth-delay product? If the system data frames are 1000 bits in length, what is the utilization percentage of the link?

Solution

The bandwidth-delay product is

$$(1 \times 10^6) \times (20 \times 10^{-3}) = 20,000 \text{ bits}$$

Figure 11.11 Flow diagram for Example 11.3



The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and then back again. However, the system sends only 1000 bits. We can say that the link utilization is only $1000/20,000$, or 5 percent. For this reason, for a link with a high bandwidth or long delay, the use of Stop-and-Wait ARQ wastes the capacity of the link.

Example 11.5

What is the utilization percentage of the link in Example 11.4 if we have a protocol that can send up to 15 frames before stopping and worrying about the acknowledgments?

Solution

The bandwidth-delay product is still 20,000 bits. The system can send up to 15 frames or 15,000 bits during a round trip. This means the utilization is $15,000/20,000$, or 75 percent. Of course, if there are damaged frames, the utilization percentage is much less because frames have to be resent.

Pipelining

In networking and in other areas, a task is often begun before the previous task has ended. This is known as pipelining. There is no pipelining in Stop-and-Wait ARQ because we need to wait for a frame to reach the destination and be acknowledged before the next frame can be sent. However, pipelining does apply to our next two protocols because several frames can be sent before we receive news about the previous frames. Pipelining improves the efficiency of the transmission if the number of bits in transition is large with respect to the bandwidth-delay product.

Go-Back-N Automatic Repeat Request

To improve the efficiency of transmission (filling the pipe), multiple frames must be in transition while waiting for acknowledgment. In other words, we need to let more than one frame be outstanding to keep the channel busy while the sender is waiting for acknowledgment. In this section, we discuss one protocol that can achieve this goal; in the next section, we discuss a second.

The first is called Go-Back-N Automatic Repeat Request (the rationale for the name will become clear later). In this protocol we can send several frames before receiving acknowledgments; we keep a copy of these frames until the acknowledgments arrive.

Sequence Numbers

Frames from a sending station are numbered sequentially. However, because we need to include the sequence number of each frame in the header, we need to set a limit. If the header of the frame allows m bits for the sequence number, the sequence numbers range from 0 to $2^m - 1$. For example, if m is 4, the only sequence numbers are 0 through 15 inclusive. However, we can repeat the sequence. So the sequence numbers are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

In other words, the sequence numbers are modulo- 2^m

In the Go-Back-N Protocol, the sequence numbers are modulo 2^m ,
where m is the size of the sequence number field in bits.

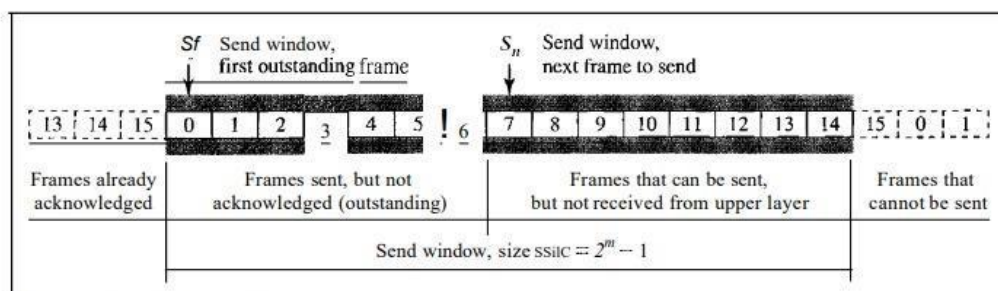
Sliding Window

In this protocol (and the next), the sliding window is an abstract concept that defines the range of sequence numbers that is the concern of the sender and receiver. In other words, the sender and receiver need to deal with only part of the possible sequence numbers. The range which is the concern of the sender is called the send sliding window; the range that is the concern of the receiver is called the receive sliding window. We discuss both here.

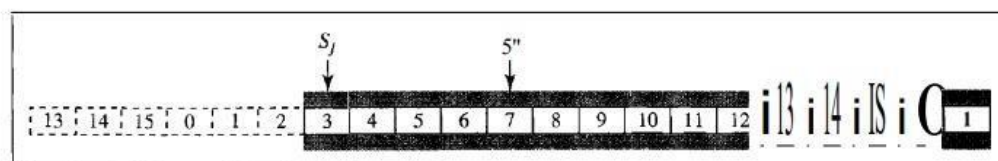
The send window is an imaginary box covering the sequence numbers of the data frames which can be in transit. In each window position, some of these sequence numbers define the frames that have been sent; others define those that can be sent. The maximum size of the window is $2^m - 1$ for reasons that we discuss later. In this chapter, we let the size be fixed and set to the maximum value, but we will see in future chapters that some protocols may have a variable window size. Figure 11.12 shows a sliding window of size 15 ($m = 4$).

The window at any time divides the possible sequence numbers into four regions. The first region, from the far left to the left wall of the window, defines the sequence numbers belonging to frames that are already acknowledged. The sender does not worry about these frames and keeps no copies of them. The second region, colored in Figure 11.12a, defines the range of sequence numbers belonging to the frames that are sent and have an unknown status. The sender needs to wait to find out if these frames have been received or were lost. We call these outstanding frames. The third range, white in the figure, defines the range of sequence numbers for frames that can be sent; however, the corresponding data packets have not yet been received from the network layer. Finally, the fourth region defines sequence numbers that cannot be used until the window slides, as we see next.

Figure 11.12 Send window for Go-Back-NARQ



a. Send window before sliding



b. Send window after sliding

The window itself is an abstraction; three variables define its size and location at any time. We call these variables S_f (send window, the first outstanding frame), S_n (send window, the next frame to be sent), and $Ssize$ (send window, size). The variable S_f defines the sequence number of the first (oldest) outstanding frame. The variable S_n holds the sequence number that will be assigned to the next frame to be sent. Finally, the variable $Ssize$ defines the size of the window, which is fixed in our protocol.

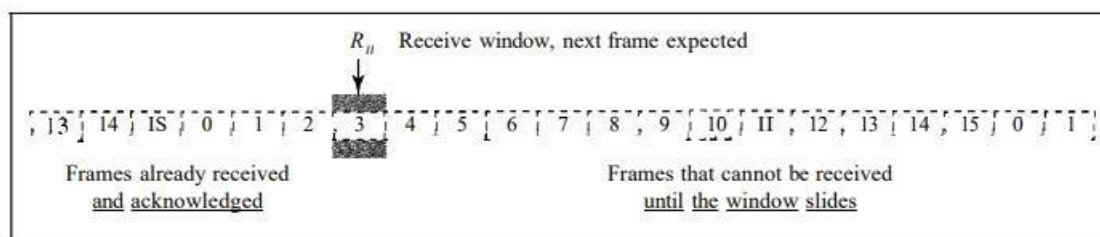
The send window is an abstract concept defining an imaginary box of size $2^m - 1$ with three variables: S_f, S_n , and $Ssize$

Figure 11.12b shows how a send window can slide one or more slots to the right when an acknowledgment arrives from the other end. As we will see shortly, the acknowledgments in this protocol are cumulative, meaning that more than one frame can be acknowledged by an ACK frame. In Figure 11.12b, frames 0, 1, and 2 are acknowledged, so the window has slid to the right three slots. Note that the value of S_f is 3 because frame 3 is now the first outstanding frame.

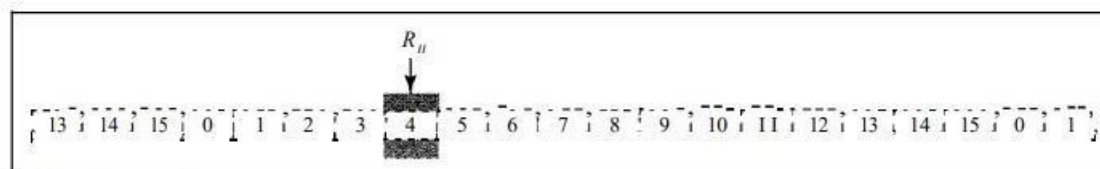
The send window can slide one or more slots when a valid acknowledgment arrives.

The receive window makes sure that the correct data frames are received and that the correct acknowledgments are sent. The size of the receive window is always 1. The receiver is always looking for the arrival of a specific frame. Any frame arriving out of order is discarded and needs to be resent. Figure 11.13 shows the receive window.

Figure 11.13 *Receive window for Go-Back-NARQ*



a. Receive window



b. Window after sliding

The receive window is an abstract concept defining an imaginary box of size 1 with one single variable R_n . The window slides when a correct frame has arrived; sliding occurs one slot at a time.

Note that we need only one variable R_n (receive window, next frame expected) to define this abstraction. The sequence numbers to the left of the window belong to the frames already received and acknowledged; the sequence numbers to the right of this window define the frames that cannot be received. Any received frame with a sequence number in these two regions is discarded. Only a frame with a sequence number matching the value of R_n is accepted and acknowledged.

The receive window also slides, but only one slot at a time. When a correct frame is received (and a frame is received only one at a time), the window slides.

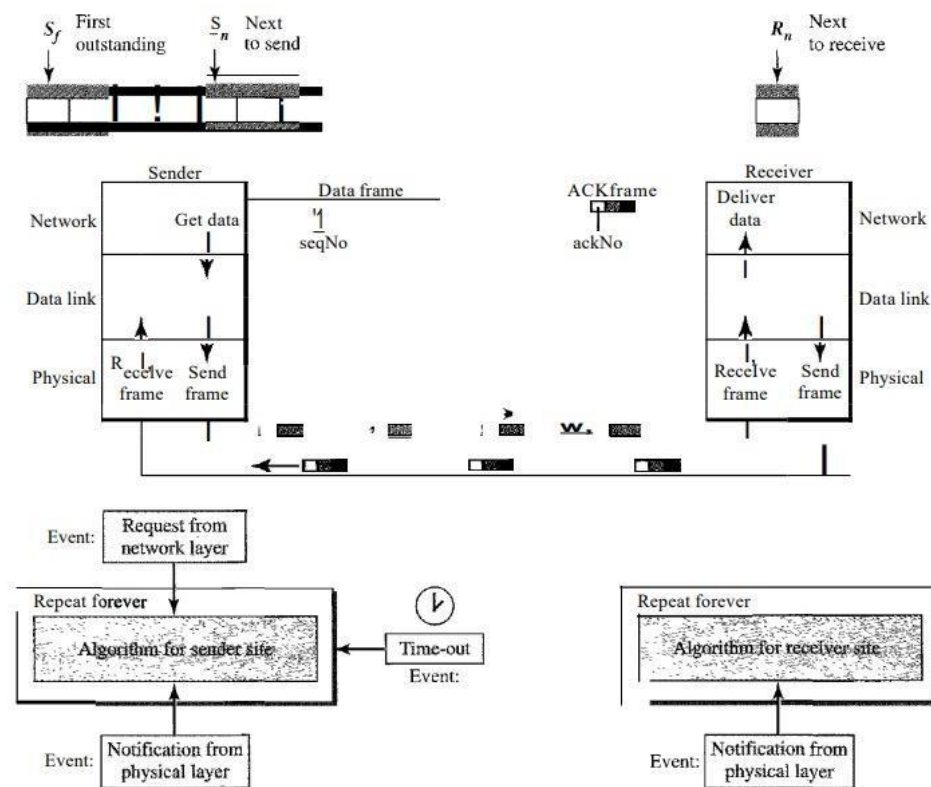
Timers

Although there can be a timer for each frame that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding frame always expires first; we send all outstanding frames when this timer expires.

Acknowledgment

The receiver sends a positive acknowledgment if a frame has arrived safe and sound and in order. If a frame is damaged or is received out of order, the receiver is silent and will discard all subsequent frames until it receives the one it is expecting. The silence of the receiver causes the timer of the unacknowledged frame at the sender site to expire. This, in turn, causes the sender to go back and resend all frames, beginning with the one with the expired timer. The receiver does not have to acknowledge each frame received. It can send one cumulative acknowledgment for several frames.

Figure 11.14 Design of Go-Back-NARQ



Resending a Frame

When the timer expires, the sender resends all outstanding frames. For example, suppose the sender has already sent frame 6, but the timer for frame 3 expires. This means that frame 3 has not been acknowledged; the sender goes back and sends frames 3, 4, 5, and 6 again. That is why the protocol is called Go-Back-N ARQ.

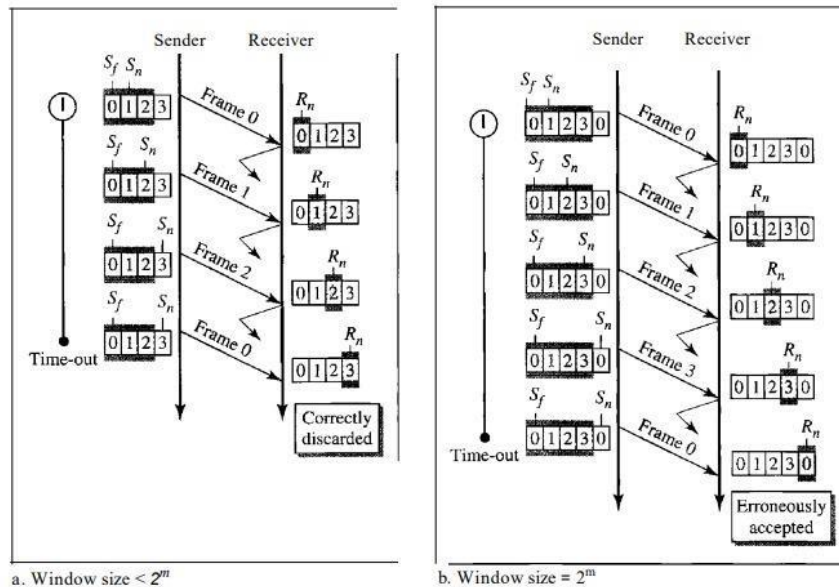
Design

Figure 11.14 shows the design for this protocol. As we can see, multiple frames can be in transit in the forward direction, and multiple acknowledgments in the reverse direction. The idea is similar to Stop-and-Wait ARQ; the difference is that the send window allows us to have as many frames in transition as there are slots in the send window.

Send Window Size

We can now show why the size of the send window must be less than 2^m . As an example, we choose $m = 2$, which means the size of the window can be $2^m - 1$, or 3. Figure 11.15 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2^2) and all three acknowledgments are lost, the frame's timer expires and all three frames are resent. The receiver is now expecting frame 3, not frame 0, so the duplicate frame is correctly discarded. On the other hand, if the size of the window is 4 (equal to 2^2) and all acknowledgments are lost, the sender will send a duplicate of frame 0. However, this time the window of the receiver expects to receive frame 0, so it accepts frame 0, not as a duplicate, but as the first frame in the next cycle. This is an error.

Figure 11.15 Window size for Go-Back-N ARQ



In Go-Back-N ARQ, the size of the send window must be less than 2^m ;
the size of the receiver window is always 1.

Algorithms

Algorithm 11.7 shows the procedure for the sender in this protocol.

Algorithm 11.7 *Go-Back-N sender algorithm*

```

1 Sw =  $2^m - 1$ ;
2 Sf = 0;
3 Sn = 0;
4
5 while (true)                                //Repeat forever
6 {
7     WaitForEvent();
8     if(Event(RequestToSend))                //A packet to send
9     {
10         if(Sn-Sf >= Sw)                     //If window is full
11             Sleep();
12         GetData();
13         MakeFrame(Sn);
14         StoreFrame(Sn);
15         SendFrame(Sn);
16         Sn = Sn + 1;
17         if(timer not running)
18             StartTimer();
19     }
20
21     if(Event(ArrivalNotification))           //ACK arrives
22     {
23         Receive(ACK);
24         if(corrupted(ACK))
25             Sleep();
26         if({ackNo>Sf}&&{ackNo<=Sn})           //If a valid ACK
27             While(Sf <= ackNo)
28             {
29                 PurgeFrame(Sf);
30                 Sf = Sf + 1;
31             }
32             StopTimer();
33     }
34
35     if(Event(TimeOut))                       //If the timer expires
36     {
37         StartTimer();
38         Temp = Sf;
39         while(Temp < Sn);
40         {
41             SendFrame(Sf);
42             Sf = Sf + 1;
43         }
44     }
45 }

```

Analysis This algorithm first initializes three variables. Unlike Stop-and-Wait ARQ, this protocol allows several requests from the network layer without the need for other events to occur; we just need to be sure that the window is not full (line 12). In our approach, if the window is full, the request is just ignored and the network layer needs to try again. Some implementations use other methods such as enabling or disabling the network layer. The handling of the arrival event is more complex than in the

previous protocol. If we receive a corrupted ACK, we ignore it. If the ackNa belongs to one of the outstanding frames, we use a loop to purge the buffers and move the left wall to the right. The time-out event is also more complex. We first start a new timer. We then resend all outstanding frames.

Algorithm 11.8 is the procedure at the receiver site.

Algorithm 11.8 Go-Back-N receiver algorithm

```

1   $R_n = 0$ ;
2
3  while (true)                                II Repeat forever
4  {
5      WaitForEvent();
6
7      if(Event{ArrivalNotification}» /Data frame arrives
8      {
9          Receive(Frame);
10         if(corrupted(Frame)»
11             Sleep();
12         if(seqNo =  $R_n$ )                    III If expected frame
13         {
14             DeliverData()i                IID Deliver data
15              $R_n = R_n + 1$ ;                IISlide window
16             SendACK( $R_n$ );
17         }
18     }
19 }
```

Analysis This algorithm is simple. We ignore a corrupt or out-of-order frame. If a frame arrives with an expected sequence number, we deliver the data, update the value of R_n , and send an ACK with the ackNa showing the next frame expected.

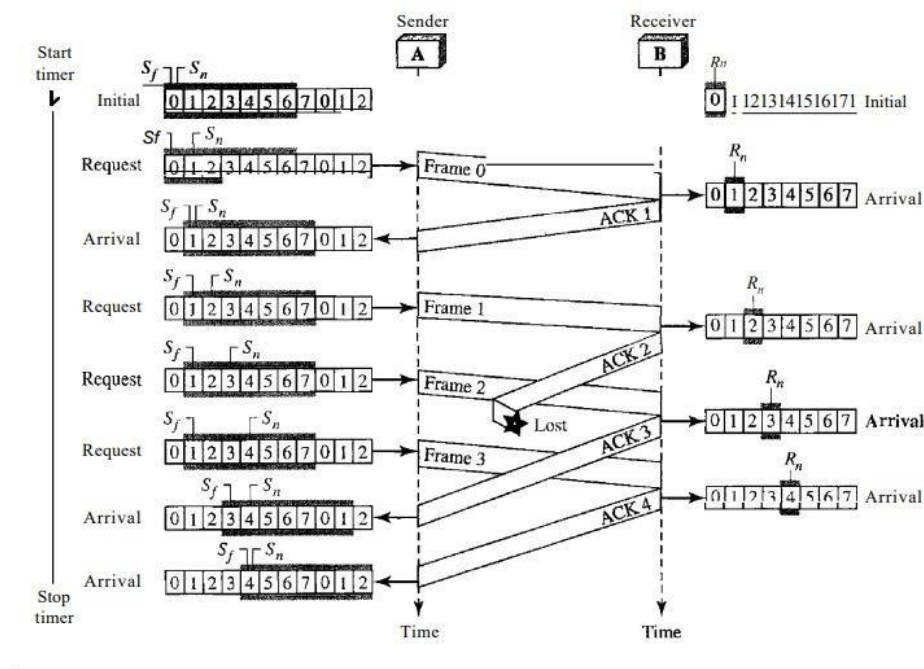
Example 11.6

Figure 11.16 shows an example of Go-Back-N. This is an example of a case where the forward channel is reliable, but the reverse is not. No data frames are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost.

After initialization, there are seven sender events. Request events are triggered by data from the network layer; arrival events are triggered by acknowledgments from the physical layer. There is no time-out event here because all outstanding frames are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 serves as both ACK 2 and ACK 3.

There are four receiver events, all triggered by the arrival of frames from the physical layer.

Figure 11.16 Flow diagram for Example 11.6



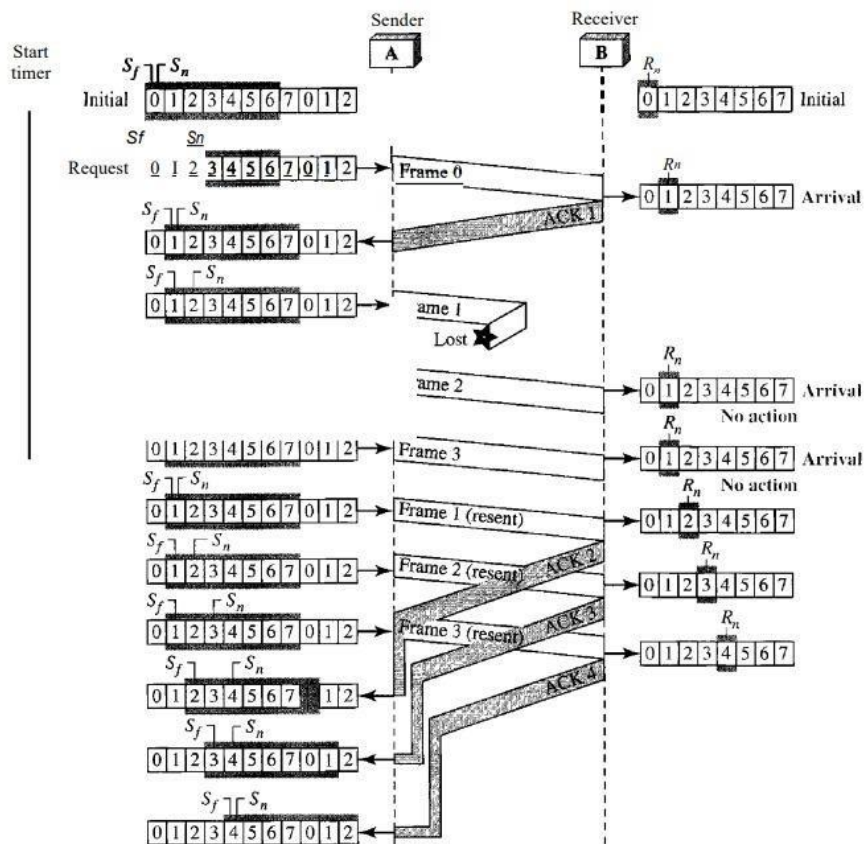
Example 11.7

Figure 11.17 shows what happens when a frame is lost. Frames 0, 1, 2, and 3 are sent. However, frame 1 is lost. The receiver receives frames 2 and 3, but they are discarded because they are received out of order (frame 1 is expected). The sender receives no acknowledgment about frames 1, 2, or 3. Its timer finally expires. The sender sends all outstanding frames (1, 2, and 3) because it does not know what is wrong. Note that the resending of frames 1, 2, and 3 is the response to one single event. When the sender is responding to this event, it cannot accept the triggering of other events. This means that when ACK 2 arrives, the sender is still busy with sending frame 3. The physical layer must wait until this event is completed and the data link layer goes back to its sleeping state. We have shown a vertical line to indicate the delay. It is the same story with ACK 3; but when ACK 3 arrives, the sender is busy responding to ACK 2. It happens again when ACK 4 arrives. Note that before the second timer expires, all outstanding frames have been sent and the timer is stopped.

Go-Back-N ARQ Versus Stop-and-Wait ARQ

The reader may find that there is a similarity between Go-Back-N ARQ and Stop-and-Wait ARQ. We can say that the Stop-and-Wait ARQ Protocol is actually a Go-Back-N ARQ in which there are only two sequence numbers and the send window size is 1. In other words, $m = 1$, $2^m - 1 = 1$. In Go-Back-N ARQ, we said that the addition is modulo- 2^m ; in Stop-and-Wait ARQ it is 2, which is the same as 2^m when $m = 1$.

Figure 11.17 Flow diagram for Example 11.7



Stop-and-WaitARQ is a special case of Go-Back-NARQ in which the size of the send window is 1.

HDLC

High-level Data Link Control (HDLC) is a bit-oriented protocol for communication over point-to-point and multipoint links. It implements the ARQ mechanisms we discussed in this chapter.

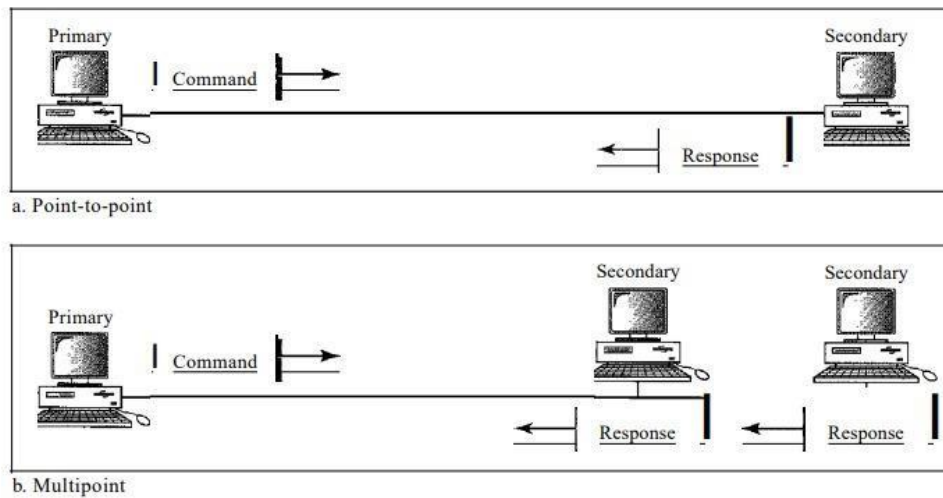
Configurations and Transfer Modes

HDLC provides two common transfer modes that can be used in different configurations: normal response mode (NRM) and asynchronous balanced mode (ABM).

Normal Response Mode

In normal response mode (NRM), the station configuration is unbalanced. We have one primary station and multiple secondary stations. A primary station can send commands; a secondary station can only respond. The NRM is used for both point-to-point and multiple-point links, as shown in Figure 11.25.

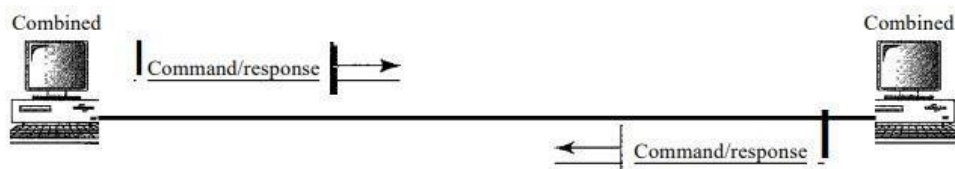
Figure 11.25 *Normal response mode*



Asynchronous Balanced Mode

In asynchronous balanced mode (ABM), the configuration is balanced. The link is point-to-point, and each station can function as a primary and a secondary (acting as peers), as shown in Figure 11.26. This is the common mode today.

Figure 11.26 *Asynchronous balanced mode*



Frames

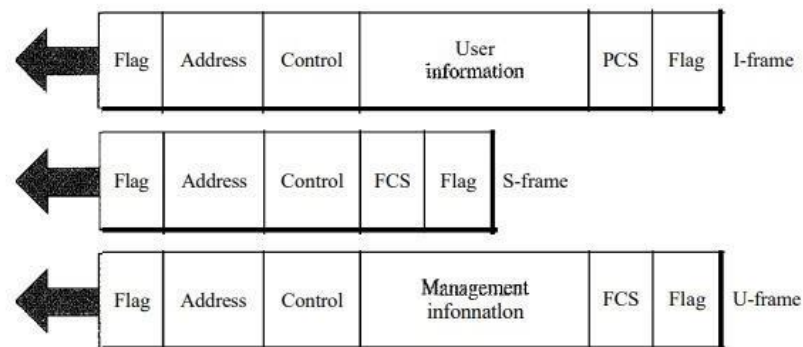
To provide the flexibility necessary to support all the options possible in the modes and configurations just described, HDLC defines three types of frames: information frames (I-frames), supervisory frames (S-frames), and unnumbered frames (V-frames). Each type of frame serves as an envelope for the transmission of a different type of message. I-frames are used to transport user data and control information relating to user data (piggybacking). S-frames are used only to transport control information. V-frames are reserved for system management. Information carried by V-frames is intended for managing the link itself.

Frame Format

Each frame in HDLC may contain up to six fields, as shown in Figure 11.27: a beginning flag field, an

address field, a control field, an information field, a frame check sequence (FCS) field, and an ending flag field. In multiple-frame transmissions, the ending flag of one frame can serve as the beginning flag of the next frame.

Figure 11.27 HDLC frames



Fields

Let us now discuss the fields and their use in different frame types.

Flag field. The flag field of an HDLC frame is an 8-bit sequence with the bit pattern 01111110 that identifies both the beginning and the end of a frame and serves as a synchronization pattern for the receiver.

Address field. The second field of an HDLC frame contains the address of the secondary station. If a primary station created the frame, it contains a to address. If a secondary creates the frame, it contains afrom address. An address field can be 1 byte or several bytes long, depending on the needs of the network. One byte can identify up to 128 stations (1 bit is used for another purpose). Larger networks require multiple-byte address fields. If the address field is only 1 byte, the last bit is always a 1. If the address is more than 1 byte, all bytes but the last one will end with 0; only the last will end with 1. Ending each intermediate byte with 0 indicates to the receiver that there are more address bytes to come.

Control field. The control field is a 1- or 2-byte segment of the frame used for flow and error control. The interpretation of bits in this field depends on the frame type. We discuss this field later and describe its format for each frame type.

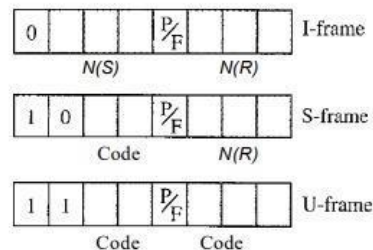
Information field. The information field contains the user's data from the network layer or management information. Its length can vary from one network to another.

FCS field. The frame check sequence (FCS) is the HDLC error detection field. It can contain either a 2- or 4-byte ITU-T CRC.

Control Field

The control field determines the type of frame and defines its functionality. So let us discuss the format of this field in greater detail. The format is specific for the type of frame, as shown in Figure 11.28.

Figure 11.28 Control field format for the different frame types



Control Field for I-Frames

I-frames are designed to carry user data from the network layer. In addition, they can include flow and error control information (piggybacking). The subfields in the control field are used to define these functions. The first bit defines the type. If the first bit of the control field is 0, this means the frame is an I-frame. The next 3 bits, called N(S), define the sequence number of the frame. Note that with 3 bits, we can define a sequence number between 0 and 7; but in the extension format, in which the control field is 2 bytes, this field is larger. The last 3 bits, called N(R), correspond to the acknowledgment number when piggybacking is used. The single bit between N(S) and N(R) is called the P/F bit. The P/F field is a single bit with a dual purpose. It has meaning only when it is set (bit = 1) and can mean poll or final. It means poll when the frame is sent by a primary station to a secondary (when the address field contains the address of the receiver). It means final when the frame is sent by a secondary to a primary (when the address field contains the address of the sender).

Control Field for S-Frames

Supervisory frames are used for flow and error control whenever piggybacking is either impossible or inappropriate (e.g., when the station either has no data of its own to send or needs to send a command or response other than an acknowledgment). S-frames do not have information fields. If the first 2 bits of the control field is 10, this means the frame is an S-frame. The last 3 bits, called N(R), corresponds to the acknowledgment number (ACK) or negative acknowledgment number (NAK) depending on the type of S-frame. The 2 bits called code is used to define the type of S-frame itself. With 2 bits, we can have four types of S-frames, as described below:

0 Receive ready (RR). If the value of the code subfield is 00, it is an RR S-frame. This kind of frame acknowledges the receipt of a safe and sound frame or group of frames. In this case, the value N(R) field defines the acknowledgment number.

0 Receive not ready (RNR). If the value of the code subfield is 10, it is an RNR S-frame. This kind of frame is an RR frame with additional functions. It acknowledges the receipt of a frame or group of frames, and it announces that the receiver is busy and cannot receive more frames. It acts as a kind

of congestion control mechanism by asking the sender to slow down. The value of NCR) is the acknowledgment number.

O Reject (REJ). If the value of the code subfield is 01, it is a REJ S-frame. This is a NAK frame, but not like the one used for Selective Repeat ARQ. It is a NAK that can be used in Go-Back-N ARQ to improve the efficiency of the process by informing the sender, before the sender time expires, that the last frame is lost or damaged. The value of NCR) is the negative acknowledgment number.

O Selective reject (SREJ). If the value of the code subfield is 11, it is an SREJ S-frame. This is a NAK frame used in Selective Repeat ARQ. Note that the HDLC Protocol uses the term selective reject instead of selective repeat. The value of N(R) is the negative acknowledgment number.

Control Field for V-Frames

Unnumbered frames are used to exchange session management and control information between connected devices. Unlike S-frames, U-frames contain an information field, but one used for system management information, not user data. As with S-frames, however, much of the information carried by U-frames is contained in codes included in the control field. U-frame codes are divided into two sections: a 2-bit prefix before the PtF bit and a 3-bit suffix after the PtF bit. Together, these two segments (5 bits) can be used to create up to 32 different types of U-frames. Some of the more common types are shown in Table 11.1.

Table 11.1 U-frame control command and response

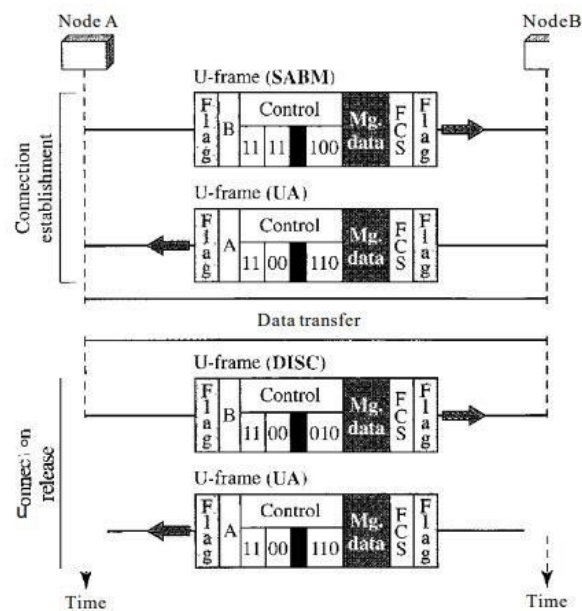
Code	Command	Response	Meaning
00 001	SNRM		Set normal response mode
11 011	SNRME		Set normal response mode, extended
11 100	SABM	DM	Set asynchronous balanced mode or disconnect mode
11 110	SABME		Set asynchronous balanced mode, extended
00 000	UI	UI	Unnumbered information
00 110		UA	Unnumbered acknowledgment
00 010	DISC	RD	Disconnect or request disconnect
10 000	SIM	RIM	Set initialization mode or request information mode
00 100	UP		Unnumbered poll
11 001	RSET		Reset
11 101	XID	XID	Exchange ID
10 001	FRMR	FRMR	Frame reject

Example 11.9: Connection/Disconnection

Figure 11.29 shows how V-frames can be used for connection establishment and connection release. Node A asks for a connection with a set asynchronous balanced mode (SABM) frame; node B gives a positive response with an unnumbered acknowledgment (UA) frame. After these two exchanges,

data can be transferred between the two nodes (not shown in the figure). After data transfer, node A sends a DISC (disconnect) frame to release the connection; it is confirmed by node B responding with a UA (unnumbered acknowledgment).

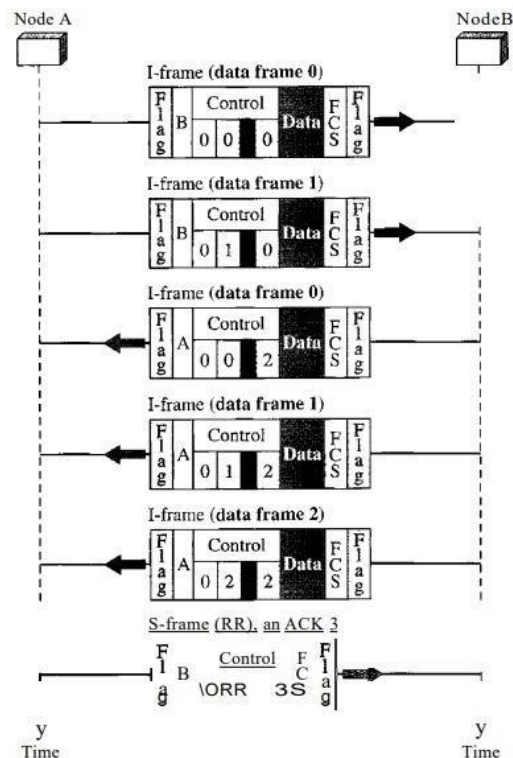
Figure 11.29 Example of connection and disconnection



Example 11.10: Piggybacking without Error

Figure 11.30 shows an exchange using piggybacking. Node A begins the exchange of information with an I-frame numbered 0 followed by another I-frame numbered 1. Node B piggybacks its acknowledgment of both frames onto an I-frame of its own. Node B's first I-frame is also numbered 0 *N(S) field+ and contains a 2 in its N(R) field, acknowledging the receipt of Ns frames 1 and 0 and indicating that it expects frame 2 to arrive next. Node B transmits its second and third I-frames (numbered 1 and 2) before accepting further frames from node A. Its N(R) information, therefore, has not changed: B frames 1 and 2 indicate that node B is still expecting Ns frame 2 to arrive next. Node A has sent all its data. Therefore, it cannot piggyback an acknowledgment onto an I-frame and sends an S-frame instead. The RR code indicates that A is still ready to receive. The number 3 in the N(R) field tells B that frames 0, 1, and 2 have all been accepted and that A is now expecting frame number 3.

Figure 11.30 Example of piggybacking without error



Example 11.11: Piggybacking with Error

Figure 11.31 shows an exchange in which a frame is lost. Node B sends three data frames (0, 1, and 2), but frame 1 is lost. When node A receives frame 2, it discards it and sends a REI frame for frame 1. Note that the protocol being used is Go-Back-N with the special use of an REI frame as a NAK frame. The NAK frame does two things here: It confirms the receipt of frame 0 and declares that frame 1 and any following frames must be resent. Node B, after receiving the REI frame, resends frames 1 and 2. Node A acknowledges the receipt by sending an RR frame (ACK) with acknowledgment number 3.

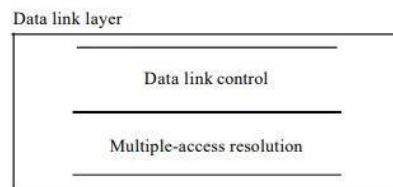
Multiple Access

In Chapter 11 we discussed data link control, a mechanism which provides a link with reliable communication. In the protocols we described, we assumed that there is an available dedicated link (or channel) between the sender and the receiver. This assumption may or may not be true. If, indeed, we have a dedicated link, as when we connect to the Internet using PPP as the data link control protocol, then the assumption is true and we do not need anything else.

On the other hand, if we use our cellular phone to connect to another cellular phone, the channel (the band allocated to the vendor company) is not dedicated. A person a few feet away from us may be using the same channel to talk to her friend.

We can consider the data link layer as two sublayers. The upper sublayer is responsible for data link control, and the lower sublayer is responsible for resolving access to the shared media. If the channel is dedicated, we do not need the lower sublayer. Figure 12.1 shows these two sublayers in the data link layer.

Figure 12.1 *Data link layer divided into two functionality-oriented sublayers*

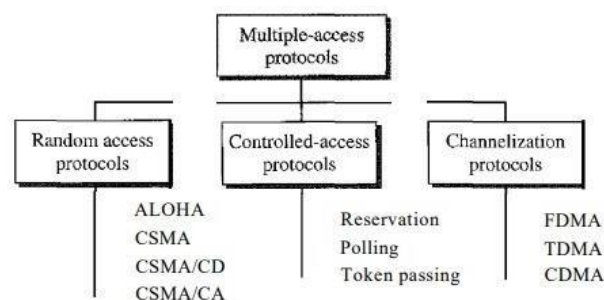


We will see in Chapter 13 that the IEEE has actually made this division for LANs. The upper sub layer that is responsible for flow and error control is called the logical link control (LLC) layer; the lower sub layer that is mostly responsible for multipleaccess resolution is called the media access control (MAC) layer.

When nodes or stations are connected and use a common link, called a multipoint or broadcast link, we need a multiple-access protocol to coordinate access to the link. The problem of controlling the access to the medium is similar to the rules of speaking in an assembly. The procedures guarantee that the right to speak is upheld and ensure that two people do not speak at the same time, do not interrupt each other, do not monopolize the discussion, and so on.

The situation is similar for multipoint networks. Many formal protocols have been devised to handle access to a shared lillie We categorize them into three groups. Protocols belonging to each group are shown in Figure 12.2.

Figure 12.2 *Taxonomy of multiple-access protocols discussed in this chapter*



RANDOM ACCESS

In random access or contention methods, no station is superior to another station and none is assigned the control over another. No station permits, or does not permit, another station to send. At each instance, a station that has data to send uses a procedure defined by the protocol to make a

decision on whether or not to send. This decision depends on the state of the medium (idle or busy). In other words, each station can transmit when it desires on the condition that it follows the predefined procedure, including the testing of the state of the medium.

Two features give this method its name. First, there is no scheduled time for a station to transmit. Transmission is random among the stations. That is why these methods are called random access. Second, no rules specify which station should send next. Stations compete with one another to access the medium. That is why these methods are also called contention methods.

In a random access method, each station has the right to the medium without being controlled by any other station. However, if more than one station tries to send, there is an access conflict-collision-and the frames will be either destroyed or modified. To avoid access conflict or to resolve it when it happens, each station follows a procedure that answers the following questions:

- o When can the station access the medium?
- o What can the station do if the medium is busy?
- o How can the station determine the success or failure of the transmission?
- o What can the station do if there is an access conflict?

The random access methods we study in this chapter have evolved from a very interesting protocol known as ALOHA, which used a very simple procedure called multiple access (MA). The method was improved with the addition of a procedure that forces the station to sense the medium before transmitting. This was called carrier sense multiple access. This method later evolved into two parallel methods: carrier sense multiple access with collision detection (CSMA/CD) and carrier sense multiple access with collision avoidance (CSMA/CA). CSMA/CD tells the station what to do when a collision is detected. CSMA/CA tries to avoid the collision.

ALOHA

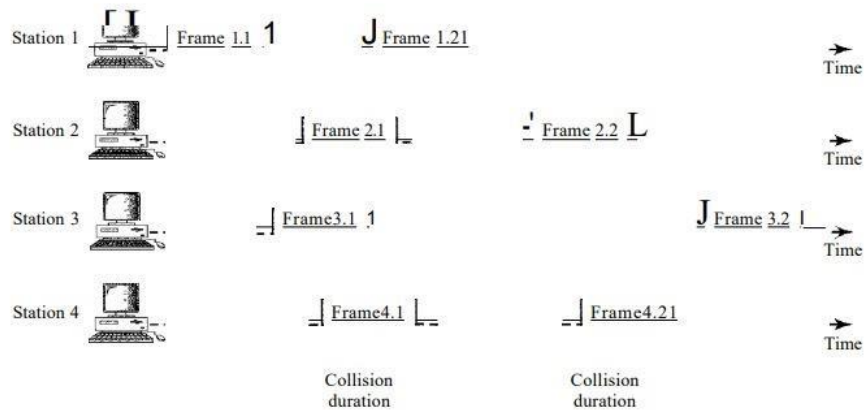
ALOHA, the earliest random access method, was developed at the University of Hawaii in early 1970. It was designed for a radio (wireless) LAN, but it can be used on any shared medium.

It is obvious that there are potential collisions in this arrangement. The medium is shared between the stations. When a station sends data, another station may attempt to do so at the same time. The data from the two stations collide and become garbled.

Pure ALOHA

The original ALOHA protocol is called pure ALOHA. This is a simple, but elegant protocol. The idea is that each station sends a frame whenever it has a frame to send. However, since there is only one channel to share, there is the possibility of collision between frames from different stations. Figure 12.3 shows an example of frame collisions in pure ALOHA.

Figure 12.3 Frames in a pure ALOHA network



There are four stations (unrealistic assumption) that contend with one another for access to the shared channel. The figure shows that each station sends two frames; there are a total of eight frames on the shared medium. Some of these frames collide because multiple frames are in contention for the shared channel. Figure 12.3 shows that only two frames survive: frame 1.1 from station 1 and frame 3.2 from station 3. We need to mention that even if one bit of a frame coexists on the channel with one bit from another frame, there is a collision and both will be destroyed.

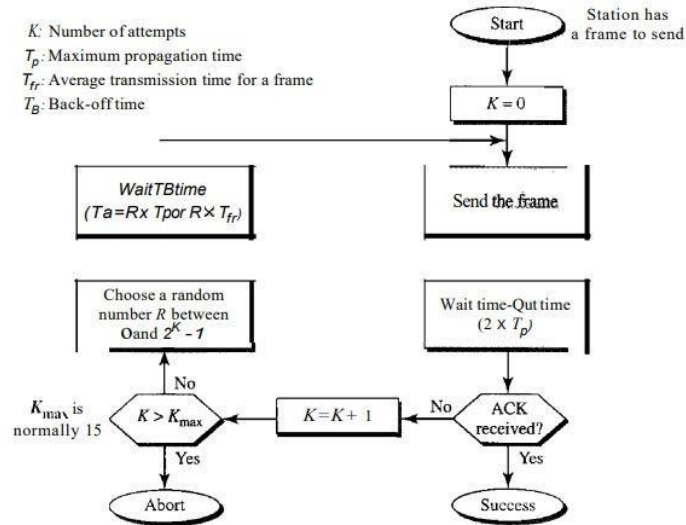
It is obvious that we need to resend the frames that have been destroyed during transmission. The pure ALOHA protocol relies on acknowledgments from the receiver. When a station sends a frame, it expects the receiver to send an acknowledgment. If the acknowledgment does not arrive after a time-out period, the station assumes that the frame (or the acknowledgment) has been destroyed and resends the frame.

A collision involves two or more stations. If all these stations try to resend their frames after the time-out, the frames will collide again. Pure ALOHA dictates that when the time-out period passes, each station waits a random amount of time before resending its frame. The randomness will help avoid more collisions. We call this time the back-off time T_B .

Pure ALOHA has a second method to prevent congesting the channel with retransmitted frames. After a maximum number of retransmission attempts K_{\max} , a station must give up and try later. Figure 12.4 shows the procedure for pure ALOHA based on the above strategy.

The time-out period is equal to the maximum possible round-trip propagation delay, which is twice the amount of time required to send a frame between the two most widely separated stations ($2 \times T_p$). The back-off time T_B is a random value that normally depends on K (the number of attempted unsuccessful transmissions). The formula for T_B depends on the implementation. One common formula is the **binary exponential back-off**. In this method, for each retransmission, a multiplier in the range 0 to $2^K - 1$ is randomly chosen and multiplied by T_p (maximum propagation time) or T_{rr} (the average time required to send out a frame) to find T_B . Note that in this procedure, the range of the random numbers increases after each collision. The value of K_{\max} is usually chosen as 15.

Figure 12.4 Procedure for pure ALOHA protocol

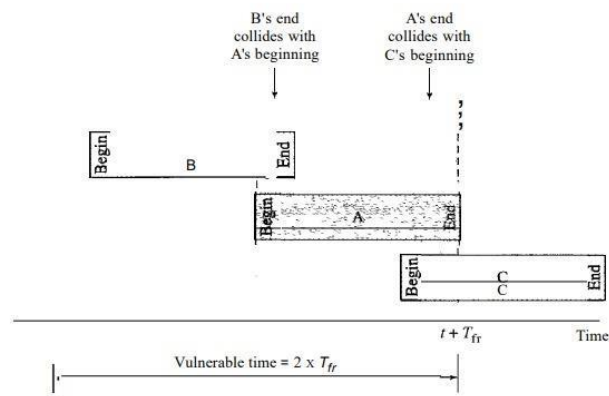


Example 12.1

The stations on a wireless ALOHA network are a maximum of 600 km apart. If we assume that signals propagate at 3×10^8 m/s, we find $T_p = (600 \times 10^3) / (3 \times 10^8) = 2$ ms. Now we can find the value of T_B for different values of K .

- For $K = 1$, the range is 0, 1. The station needs to generate a random number with a value of 0 or 1. This means that T_B is either 0 ms (0×2) or 2 ms (1×2), based on the outcome of the random variable.
- For $K = 2$, the range is 0, 1, 2, 3. This means that T_B can be 0, 2, 4, or 6 ms, based on the outcome of the random variable.
- For $K = 3$, the range is 0, 1, 2, 3, 4, 5, 6, 7. This means that T_B can be 0, 2, 4, ..., 14 ms, based on the outcome of the random variable.
- We need to mention that if $K > 10$, it is normally set to 10.

Figure 12.5 Vulnerable time for pure ALOHA protocol



Vulnerable time Let us find the length of time, the vulnerable time, in which there is a possibility of collision. We assume that the stations send fixed-length frames with each frame taking T_{fr} to send. Figure 12.5 shows the vulnerable time for station A.

Station A sends a frame at time t . Now imagine station B has already sent a frame between $t - T_{fr}$ and t . This leads to a collision between the frames from station A and station B. The end of B's frame collides with the beginning of A's frame. On the other hand, suppose that station C sends a frame between t and $t + T_{fr}$. Here, there is a collision between frames from station A and station C. The beginning of C's frame collides with the end of A's frame.

Looking at Figure 12.5, we see that the vulnerable time, during which a collision may occur in pure ALOHA, is 2 times the frame transmission time.

$$\text{Pure ALOHA vulnerable time} = 2 \times T_{fr}$$

Example 12.2

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the requirement to make this frame collision-free?

Solution

Average frame transmission time T_{fr} is 200 bits/200 kbps or 1 ms. The vulnerable time is $2 \times 1 \text{ ms} = 2 \text{ ms}$. This means no station should send later than 1 ms before this station starts transmission and no station should start sending during the one 1-ms period that this station is sending.

Throughput Let us call G the average number of frames generated by the system during one frame transmission time. Then it can be proved that the average number of successful transmissions for pure ALOHA is $S = G \times e^{-2G}$. The maximum throughput S_{max} is 0.184, for $G = 1$. In other words, if one-half a frame is generated during one 2 frame transmission time (in other words, one frame during two frame transmission times), then 18.4 percent of these frames reach their destination successfully. This is an expected result because the vulnerable time is 2 times the frame transmission time. Therefore, if a station generates only one frame in this vulnerable time (and no other stations generate a frame during this time), the frame will reach its destination successfully.

$$\begin{aligned} \text{The throughput for pure ALOHA is } S &= G \times e^{-2G}. \\ \text{The maximum throughput } S_{max} &= 0.184 \text{ when } G = (1/2). \end{aligned}$$

Example 12.3

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces

- a. 1000 frames per second
- b. 500 frames per second
- c. 250 frames per second

Solution

The frame transmission time is 2001200 kbps or 1 ms

a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G \times e^{-2G}$ or $S = 0.135$ (13.5 percent). This means that the throughput is $1000 \times 0.135 = 135$ frames. Only 135 frames out of 1000 will probably survive.

b. If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case $S = G \times e^{-2G}$ or $S = 0.184$ (18.4 percent). This means that the throughput is $500 \times 0.184 = 92$ and that only 92 frames out of 500 will probably survive. Note that this is the maximum throughput case, percentagewise.

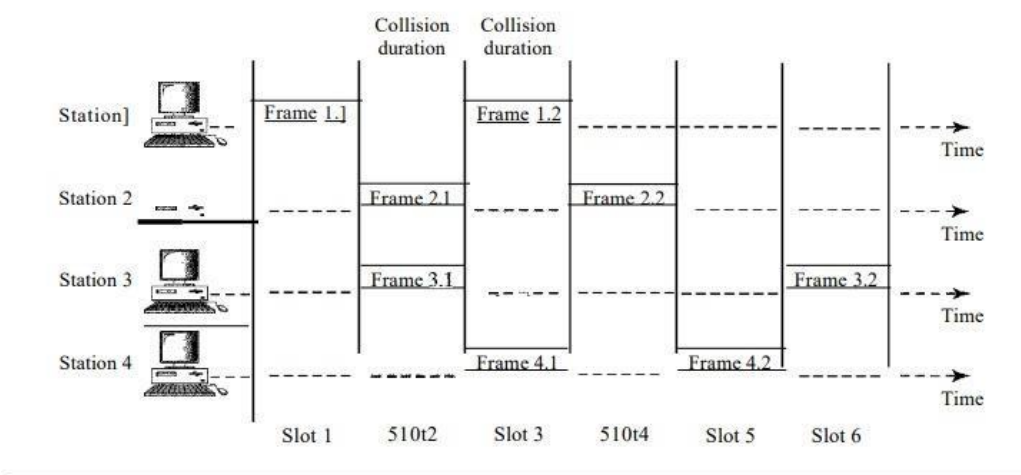
c. If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case $S = G \times e^{-2G}$ or $S = 0.152$ (15.2 percent). This means that the throughput is $250 \times 0.152 = 38$. Only 38 frames out of 250 will probably survive.

Slotted ALOHA

Pure ALOHA has a vulnerable time of $2 \times T_{fr}$. This is so because there is no rule that defines when the station can send. A station may send soon after another station has started or soon before another station has finished. Slotted ALOHA was invented to improve the efficiency of pure ALOHA.

In slotted ALOHA we divide the time into slots of T_{fr} s and force the station to send only at the beginning of the time slot. Figure 12.6 shows an example of frame collisions in slotted ALOHA.

Figure 12.6 Frames in a slotted ALOHA network



Because a station is allowed to send only at the beginning of the synchronized time slot, if a station misses this moment, it must wait until the beginning of the next time slot. This means that the station which started at the beginning of this slot has already finished sending its frame. Of course, there is still the possibility of collision if two stations try to send at the beginning of the same time slot. However, the vulnerable time is now reduced to one-half, equal to T_{fr} . Figure 12.7 shows the situation.

Figure 12.7 shows that the vulnerable time for slotted ALOHA is one-half that of pure ALOHA.

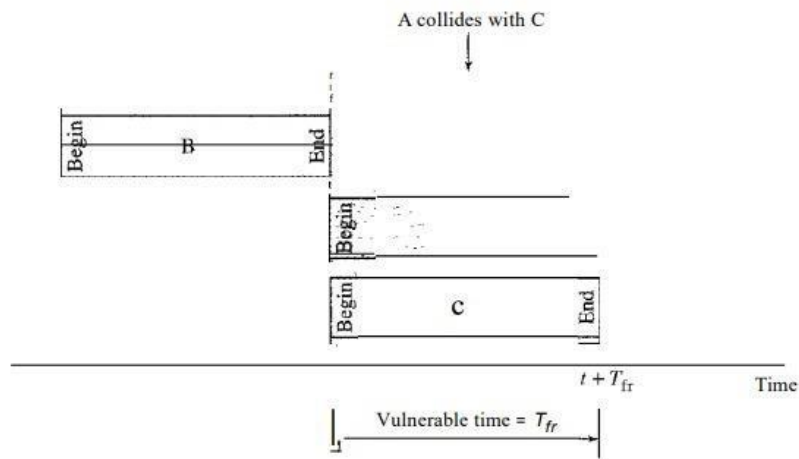
Slotted ALOHA vulnerable time = T_{fr}

Throughput It can be proved that the average number of successful transmissions for slotted ALOHA is $S = G \times e^{-G}$. The maximum throughput S_{max} is 0.368, when $G = 1$. In other words, if a frame is generated during one frame transmission time, then 36.8 percent of these frames reach their destination successfully. This result can be expected because the vulnerable time is equal to the frame transmission time. Therefore, if a station generates only one frame in this vulnerable time (and no other station generates a frame during this time), the frame will reach its destination successfully.

The throughput for slotted ALOHA is $S = G \times e^{-G}$.

The maximum throughput $S_{max} = 0.368$ when $G = 1$.

Figure 12.7 Vulnerable time for slotted ALOHA protocol



Example 12.4

A slotted ALOHA network transmits 200-bit frames using a shared channel with a 200-kbps bandwidth. Find the throughput if the system (all stations together) produces

- 1000 frames per second
- 500 frames per second
- 250 frames per second

Solution

This situation is similar to the previous exercise except that the network is using slotted ALOHA instead of pure ALOHA. The frame transmission time is $200/200$ kbps or 1 ms.

a. In this case G is 1. So $S = G \times e^{-G}$ or $S = 0.368$ (36.8 percent). This means that the throughput is $1000 \times 0.0368 = 368$ frames. Only 368 out of 1000 frames will probably survive. Note that this is the maximum throughput case, percentagewise.

b. Here G is ∞ . In this case $S = G \times e^{-G}$ or $S = 0.303$ (30.3 percent). This means that the throughput is $500 \times 0.0303 = 151$. Only 151 frames out of 500 will probably survive.

c. Now G is 1. In this case $S = G \times e^{-G}$ or $S = 0.195$ (19.5 percent). This means that the throughput is $250 \times 0.195 = 49$. Only 49 frames out of 250 will probably survive.

CONTROLLED ACCESS

In controlled access, the stations consult one another to find which station has the right to send. A station cannot send unless it has been authorized by other stations. We discuss three popular controlled-access methods.

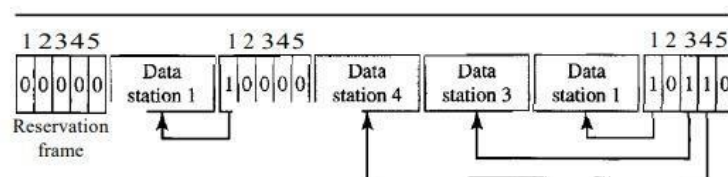
Reservation

In the reservation method, a station needs to make a reservation before sending data. Time is divided into intervals. In each interval, a reservation frame precedes the data frames sent in that interval.

If there are N stations in the system, there are exactly N reservation minislots in the reservation frame. Each minislot belongs to a station. When a station needs to send a data frame, it makes a reservation in its own minislot. The stations that have made reservations can send their data frames after the reservation frame.

Figure 12.18 shows a situation with five stations and a five-minislot reservation frame. In the first interval, only stations 1, 3, and 4 have made reservations. In the second interval, only station 1 has made a reservation.

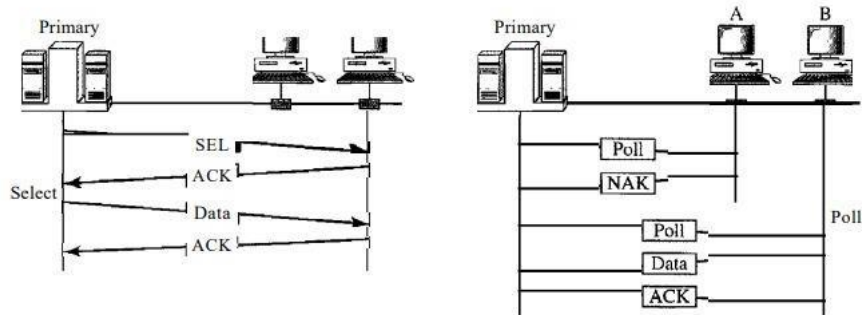
Figure 12.18 Reservation access method



Polling

Polling works with topologies in which one device is designated as a primary station and the other devices are secondary stations. All data exchanges must be made through the primary device even when the ultimate destination is a secondary device. The primary device controls the link; the secondary devices follow its instructions. It is up to the primary device to determine which device is allowed to use the channel at a given time. The primary device, therefore, is always the initiator of a session (see Figure 12.19).

Figure 12.19 *Select and poll functions in polling access method*



If the primary wants to receive data, it asks the secondaries if they have anything to send; this is called poll function. If the primary wants to send data, it tells the secondary to get ready to receive; this is called select function.

Select

The select function is used whenever the primary device has something to send. Remember that the primary controls the link. If the primary is neither sending nor receiving data, it knows the link is available.

If it has something to send, the primary device sends it. What it does not know, however, is whether the target device is prepared to receive. So the primary must alert the secondary to the upcoming transmission and wait for an acknowledgment of the secondary's ready status. Before sending data, the primary creates and transmits a select (SEL) frame, one field of which includes the address of the intended secondary.

Poll

The poll function is used by the primary device to solicit transmissions from the secondary devices. When the primary is ready to receive data, it must ask (poll) each device in turn if it has anything to send. When the first secondary is approached, it responds either with a NAK frame if it has nothing to send or with data (in the form of a data frame) if it does. If the response is negative (a NAK frame), then the primary polls the next secondary in the same manner until it finds one with data to send. When the response is positive (a data frame), the primary reads the frame and returns an acknowledgment (ACK frame), verifying its receipt.

Token Passing

In the token-passing method, the stations in a network are organized in a logical ring. In other words, for each station, there is a predecessor and a successor. The predecessor is the station which is logically before the station in the ring; the successor is the station which is after the station in the ring. The current station is the one that is accessing the channel now. The right to this access has been passed from the predecessor to the current station. The right will be passed to the successor when the current station has no more data to send.

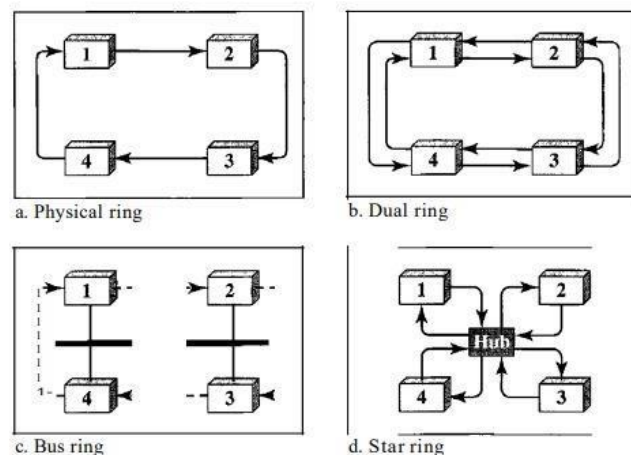
But how is the right to access the channel passed from one station to another? In this method, a special packet called a token circulates through the ring. The possession of the token gives the station the right to access the channel and send its data. When a station has some data to send, it waits until it receives the token from its predecessor. It then holds the token and sends its data. When the station has no more data to send, it releases the token, passing it to the next logical station in the ring. The station cannot send data until it receives the token again in the next round. In this process, when a station receives the token and has no data to send, it just passes the data to the next station.

Token management is needed for this access method. Stations must be limited in the time they can have possession of the token. The token must be monitored to ensure it has not been lost or destroyed. For example, if a station that is holding the token fails, the token will disappear from the network. Another function of token management is to assign priorities to the stations and to the types of data being transmitted. And finally, token management is needed to make low-priority stations release the token to highpriority stations.

Logical Ring

In a token-passing network, stations do not have to be physically connected in a ring; the ring can be a logical one. Figure 12.20 show four different physical topologies that can create a logical ring.

Figure 12.20 *Logical ring and physical topology in token-passing access method*



In the physical ring topology, when a station sends the token to its successor, the token cannot be seen by other stations; the successor is the next one in line. This means that the token does not have to have the address of the next successor. The problem with this topology is that if one of the links-the medium between two adjacent stations-fails, the whole system fails.

The dual ring topology uses a second (auxiliary) ring which operates in the reverse direction compared with the main ring. The second ring is for emergencies only (such as a spare tire for a car). If one of the links in the main ring fails, the system automatically combines the two rings to form a temporary ring. After the failed link is restored, the auxiliary ring becomes idle again. Note that for

this topology to work, each station needs to have two transmitter ports and two receiver ports. The high-speed Token Ring networks called FDDI (Fiber Distributed Data Interface) and CDDI (Copper Distributed Data Interface) use this topology.

In the bus ring topology, also called a token bus, the stations are connected to a single cable called a bus. They, however, make a logical ring, because each station knows the address of its successor (and also predecessor for token management purposes). When a station has finished sending its data, it releases the token and inserts the address of its successor in the token. Only the station with the address matching the destination address of the token gets the token to access the shared media. The Token Bus LAN, standardized by IEEE, uses this topology.

In a star ring topology, the physical topology is a star. There is a hub, however, that acts as the connector. The wiring inside the hub makes the ring; the stations are connected to this ring through the two wire connections. This topology makes the network less prone to failure because if a link goes down, it will be bypassed by the hub and the rest of the stations can operate. Also adding and removing stations from the ring is easier. This topology is still used in the Token Ring LAN designed by IBM.

CHANNELIZATION

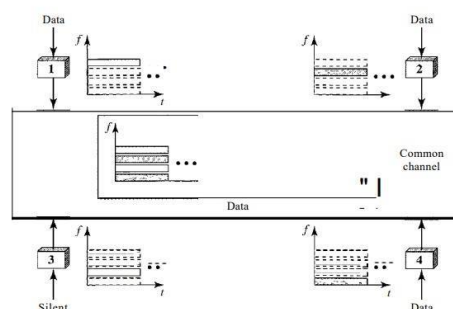
Channelization is a multiple-access method in which the available bandwidth of a link is shared in time, frequency, or through code, between different stations. In this section, we discuss three channelization protocols: FDMA, TDMA, and CDMA.

We see the application of all these methods in Chapter 16
when we discuss cellular phone systems.

Frequency-Division Multiple Access (FDMA)

In frequency-division multiple access (FDMA), the available bandwidth is divided into frequency bands. Each station is allocated a band to send its data. In other words, each band is reserved for a specific station, and it belongs to the station all the time. Each station also uses a bandpass filter to confine the transmitter frequencies. To prevent station interferences, the allocated bands are separated from one another by small guard bands. Figure 12.21 shows the idea of FDMA.

Figure 12.21 Frequency-division multiple access (FDMA)



In FDMA, the available bandwidth of the common channel
is divided into bands that are separated by guard bands.

FDMA specifies a predetermined frequency band for the entire period of communication. This means that stream data (a continuous flow of data that may not be packetized) can easily be used with FDMA. We will see in Chapter 16 how this feature can be used in cellular telephone systems.

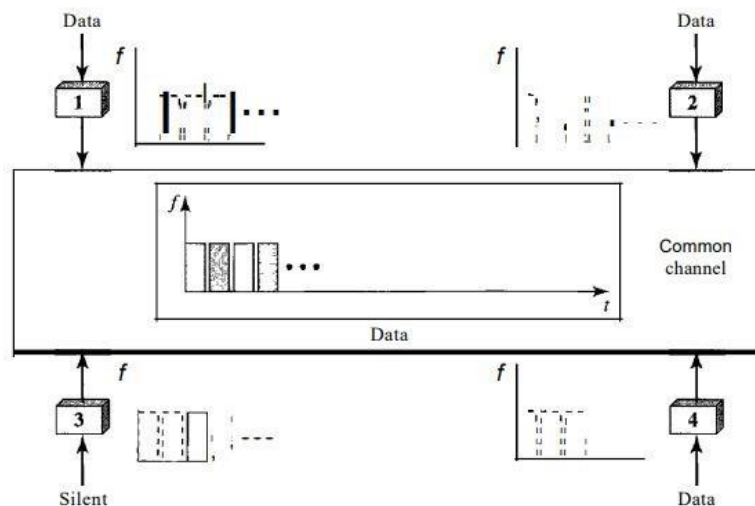
We need to emphasize that although FDMA and FDM conceptually seem similar, there are differences between them. FDM, as we saw in Chapter 6, is a physical layer technique that combines the loads from low-bandwidth channels and transmits them by using a high-bandwidth channel. The channels that are combined are low-pass. The multiplexer modulates the signals, combines them, and creates a bandpass signal. The bandwidth of each channel is shifted by the multiplexer.

FDMA, on the other hand, is an access method in the data link layer. The data link layer in each station tells its physical layer to make a bandpass signal from the data passed to it. The signal must be created in the allocated band. There is no physical multiplexer at the physical layer. The signals created at each station are automatically bandpass-filtered. They are mixed when they are sent to the common channel.

Time-Division Multiple Access (TDMA)

In time-division multiple access (TDMA), the stations share the bandwidth of the channel in time. Each station is allocated a time slot during which it can send data. Each station transmits its data in its assigned time slot. Figure 12.22 shows the idea behind TDMA.

Figure 12.22 Time-division multiple access (TDMA)



The main problem with TDMA lies in achieving synchronization between the different stations. Each station needs to know the beginning of its slot and the location of its slot. This may be difficult because of propagation delays introduced in the system if the stations are spread over a large area. To compensate for the delays, we can insert guard times. Synchronization is normally accomplished by having some synchronization bits (normally referred to as preamble bits) at the beginning of each slot.

In TDMA, the bandwidth is just one channel that is timeshared between different stations.

We also need to emphasize that although TDMA and TDM conceptually seem the same, there are differences between them. TDM, as we saw in Chapter 6, is a physical layer technique that combines the data from slower channels and transmits them by using a faster channel. The process uses a physical multiplexer that interleaves data units from each channel.

TDMA, on the other hand, is an access method in the data link layer. The data link layer in each station tells its physical layer to use the allocated time slot. There is no physical multiplexer at the physical layer.

Code-Division Multiple Access (CDMA)

Code-division multiple access (CDMA) was conceived several decades ago. Recent advances in electronic technology have finally made its implementation possible. CDMA differs from FDMA because only one channel occupies the entire bandwidth of the link. It differs from TDMA because all stations can send data simultaneously; there is no timesharing.

In CDMA, one channel carries all transmissions simultaneously.

Analogy

Let us first give an analogy. CDMA simply means communication with different codes. For example, in a large room with many people, two people can talk in English if nobody else understands English. Another two people can talk in Chinese if they are the only ones who understand Chinese, and so on. In other words, the common channel, the space of the room in this case, can easily allow communication between several couples, but in different languages (codes).

Idea

Let us assume we have four stations 1, 2, 3, and 4 connected to the same channel. The data from station 1 are d_1 , from station 2 are d_2 , and so on. The code assigned to the first station is c_1 , to the second is c_2 , and so on. We assume that the assigned codes have two properties.

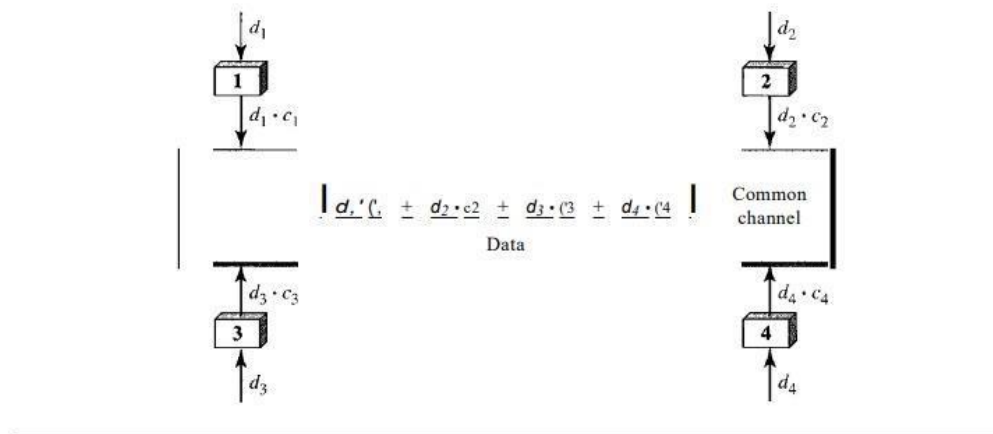
1. If we multiply each code by another, we get 0.
2. If we multiply each code by itself, we get 4 (the number of stations).

With these two properties in mind, let us see how the above four stations can send data using the same common channel, as shown in Figure 12.23.

Station 1 multiplies (a special kind of multiplication, as we will see) its data by its code to get $d_1 \cdot c_1$. Station 2 multiplies its data by its code to get $d_2 \cdot c_2$. And so on. The data that go on the channel are the sum of all these terms, as shown in the box. Any station that wants to receive data

from one of the other three multiplies the data on the channel by the code of the sender. For example, suppose stations 1 and 2 are talking to each other. Station 2 wants to hear what station 1 is saying. It multiplies the data on the channel by c_1 the code of station 1.

Figure 12.23 Simple idea of communication with code



Because $(c_1 \cdot c_1)$ is 4, but $(c_2 \cdot c_1)$, $(c_3 \cdot c_1)$, and $(c_4 \cdot c_1)$ are all 0s, station 2 divides the result by 4 to get the data from station 1.

$$\begin{aligned} \text{data} &= (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4) \cdot c_1 \\ &= d_1 \cdot c_1 \cdot c_1 + d_2 \cdot c_2 \cdot c_1 + d_3 \cdot c_3 \cdot c_1 + d_4 \cdot c_4 \cdot c_1 = 4 \times d_1 \end{aligned}$$

Chips

CDMA is based on coding theory. Each station is assigned a code, which is a sequence of numbers called chips, as shown in Figure 12.24. The codes are for the previous example.

Figure 12.24 Chip sequences



Later in this chapter we show how we chose these sequences. For now, we need to know that we did not choose the sequences randomly; they were carefully selected. They are called orthogonal sequences and have the following properties:

1. Each sequence is made of N elements, where N is the number of stations.
2. If we multiply a sequence by a number, every element in the sequence is multiplied by that element. This is called multiplication of a sequence by a scalar. For example,

$$2 \cdot [+1 \ +1 \ -1 \ -1] = [+2 \ +2 \ -2 \ -2]$$

3. If we multiply two equal sequences, element by element, and add the results, we get N, where N is the number of elements in the each sequence. This is called the inner product of two equal sequences. For example,

$$*+1+1-1-1 \cdot *+1+1-1-1+ = 1+1+1+1 = 4$$

4. If we multiply two different sequences, element by element, and add the results, we get 0. This is called inner product of two different sequences. For example,

$$*+1+1-1-1+ \cdot *+1+1+1+1+ = 1+1-1-1 = 0$$

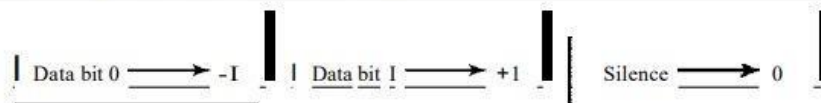
5. Adding two sequences means adding the corresponding elements. The result is another sequence. For example,

$$*+1+1-1-1++*+1+1+1+1+=*+2+2\ 00+$$

Data Representation

We follow these rules for encoding: If a station needs to send a 0 bit, it encodes it as -1; if it needs to send a 1 bit, it encodes it as +1. When a station is idle, it sends no signal, which is interpreted as a 0. These are shown in Figure 12.25.

Figure 12.25 Data representation in CDMA



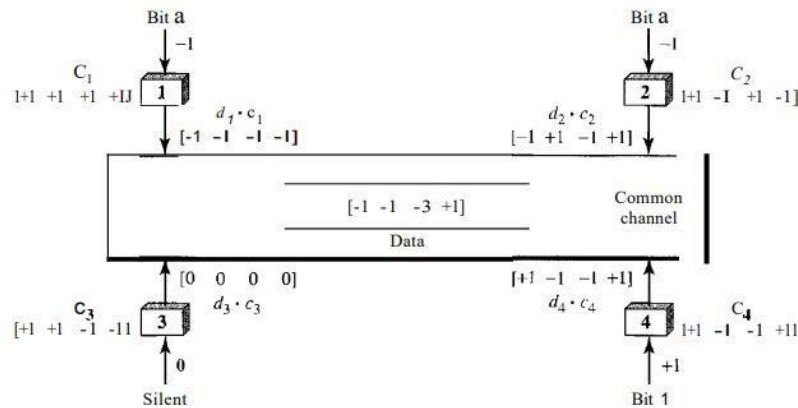
Encoding and Decoding

As a simple example, we show how four stations share the link during a 1-bit interval. The procedure can easily be repeated for additional intervals. We assume that stations 1 and 2 are sending a 0 bit and channel 4 is sending a 1 bit. Station 3 is silent. The data at the sender site are translated to -1, -1, 0, and +1. Each station multiplies the corresponding number by its chip (its orthogonal sequence), which is unique for each station. The result is a new sequence which is sent to the channel. For simplicity, we assume that all stations send the resulting sequences at the same time. The sequence on the channel is the sum of all four sequences as defined before. Figure 12.26 shows the situation.

Now imagine station 3, which we said is silent, is listening to station 2. Station 3 multiplies the total data on the channel by the code for station 2, which is $*+1-1+1-1+$, to get

$$*-1-1-3+1+ \cdot *+1-1+1-1+ = -4/4 = -1 \dots \text{bit 1}$$

Figure 12.26 *Sharing channel in CDMA*



Signal Level

The process can be better understood if we show the digital signal produced by each station and the data recovered at the destination (see Figure 12.27). The figure shows the corresponding signals for each station (using NRZ-L for simplicity) and the signal that is on the common channel.

Figure 12.27 *Digital signal created by four stations in CDMA*

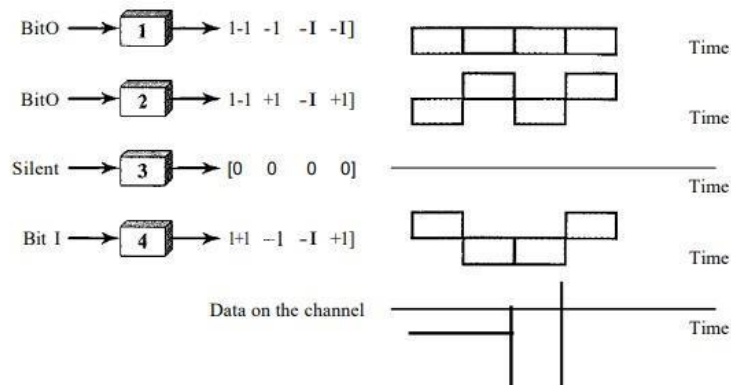
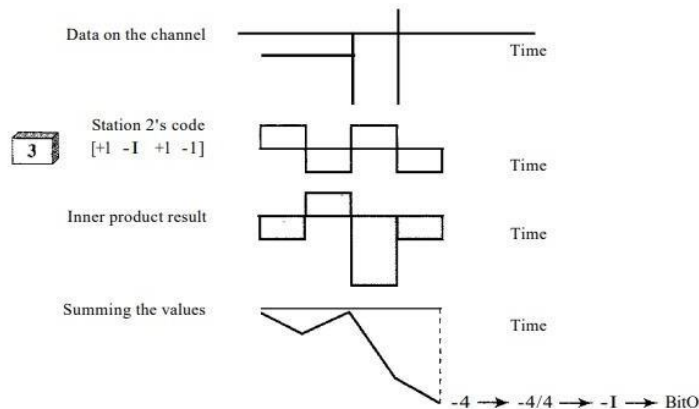


Figure 12.28 shows how station 3 can detect the data sent by station 2 by using the code for station 2. The total data on the channel are multiplied (inner product operation) by the signal representing station 2 chip code to get a new signal. The station then integrates and adds the area under the signal, to get the value -4, which is divided by 4 and interpreted as bit 0.

Figure 12.28 Decoding of the composite signal for one in CDMA



Sequence Generation

To generate chip sequences, we use a Walsh table, which is a two-dimensional table with an equal number of rows and columns, as shown in Figure 12.29.

Figure 12.29 General rule and examples of creating Walsh tables

$$W_1 = \begin{bmatrix} +1 \end{bmatrix} \quad W_{2N} = \begin{bmatrix} W_N & \overline{W_N} \\ W_N & \overline{W_N} \end{bmatrix}$$

a. Two basic rules

$$W_1 = \begin{bmatrix} +1 \end{bmatrix} \quad W_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \quad W_4 = \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix}$$

b. Generation of W_1 , W_2 , and W_4

In the Walsh table, each row is a sequence of chips. W_1 for a one-chip sequence has one row and one column. We can choose -1 or +1 for the chip for this trivial table (we chose +1). According to Walsh, if we know the table for N sequences W_N we can create the table for $2N$ sequences W_{2N} , as shown in Figure 12.29. The W_N with the over bar $\overline{W_N}$ stands for the complement of W_N where each +1 is changed to -1 and vice versa. Figure 12.29 also shows how we can create W_2 and W_4 from W_1 . After we select W_1 , W_2 can be made from four W_1 's, with the last one the complement of W_1 . After W_2 is generated, W_4 can be made of four W_2 's, with the last one the complement of W_2 . Of course, W_8 is composed of four W_4 's, and so on. Note that after W_N is made, each station is assigned a chip corresponding to a row.

Something we need to emphasize is that the number of sequences N needs to be a power of 2. In other words, we need to have $N = 2^m$

802.11 MAC Protocol

Once a wireless station is associated with an AP, it can start sending and receiving data frames to and from the access point. But because multiple stations may want to transmit data frames at the same time over the same channel, a multiple access protocol is needed to coordinate the transmissions. Here, a **station** is either a wireless station or an AP. As discussed in Chapter 5 and Section 6.2.1, broadly speaking there are three classes of multiple access protocols: channel partitioning (including CDMA), random access, and taking turns. Inspired by the huge success of Ethernet and its random access protocol, the designers of 802.11 chose a random access protocol for 802.11 wireless LANs. This random access protocol is referred to as **CSMA with collision avoidance**, or more succinctly as **CSMA/CA**. As with Ethernet's CSMA/CD, the "CSMA" in CSMA/CA stands for "carrier sense multiple access," meaning that each station senses the channel before transmitting, and refrains from transmitting when the channel is sensed busy. Although both Ethernet and 802.11 use carrier-sensing random access, the two MAC protocols have important differences.

First, instead of using collision detection, 802.11 uses collision-avoidance techniques. Second, because of the relatively high bit error rates of wireless channels, 802.11 (unlike Ethernet) uses a link-layer acknowledgment/retransmission (ARQ) scheme. We'll describe 802.11's collision-avoidance and link-layer acknowledgment schemes below.

Recall from Sections 5.3.2 and 5.4.2 that with Ethernet's collision-detection algorithm, an Ethernet station listens to the channel as it transmits. If, while transmitting, it detects that another station is also transmitting, it aborts its transmission and tries to transmit again after waiting a small, random amount of time. Unlike the 802.3 Ethernet protocol, the 802.11 MAC protocol does not implement collision detection. There are two important reasons for this:

- The ability to detect collisions requires the ability to send (the station's own signal) and receive (to determine whether another station is also transmitting) at the same time. Because the strength of the received signal is typically very small compared to the strength of the transmitted signal at the 802.11 adapter, it is costly to build hardware that can detect a collision.
- More importantly, even if the adapter could transmit and listen at the same time (and presumably abort transmission when it senses a busy channel), the adapter would still not be able to detect all collisions, due to the hidden terminal problem and fading, as discussed in Section 6.2.

Because 802.11 wireless LANs do not use collision detection, once a station begins to transmit a frame, it transmits the frame in its entirety; that is, once a station gets started, there is no turning back. As one might expect, transmitting entire frames (particularly long frames) when collisions are prevalent can significantly degrade a multiple access protocol's performance. In order to reduce the likelihood of collisions, 802.11 employs several collision-avoidance techniques, which we'll shortly discuss.

Before considering collision avoidance, however, we'll first need to examine 802.11's **link-layer acknowledgment scheme**. Recall from Section 6.2 that when a station in a wireless LAN sends a frame, the frame may not reach the destination station intact for a variety of reasons. To deal with this non-negligible chance of failure, the 802.11 MAC protocol uses link-layer acknowledgments. As shown in Figure 6.10, when the destination station receives a frame that passes the CRC, it waits a short period of time known as the **Short Inter-frame Spacing (SIFS)** and then sends back an acknowledgment frame. If the transmitting station does not receive an acknowledgment within a given amount of time, it assumes that an error has occurred and retransmits the frame, using the CSMA/CA protocol to access the channel. If an acknowledgment is not received after some fixed number of retransmissions, the transmitting station gives up and discards the frame.

Having discussed how 802.11 uses link-layer acknowledgments, we're now in a position to describe the 802.11 CSMA/CA protocol. Suppose that a station (wireless station or an AP) has a frame to transmit.

1. If initially the station senses the channel idle, it transmits its frame after a short period of time known as the **Distributed Inter-frame Space (DIFS)**; see Figure 6.10.
2. Otherwise, the station chooses a random back off value using binary exponential back off (as we encountered in Section 5.3.2) and counts down this value when the channel is sensed idle. While the channel is sensed busy, the counter value remains frozen.

3. When the counter reaches zero (note that this can only occur while the channel is sensed idle), the station transmits the entire frame and then waits for an acknowledgment.

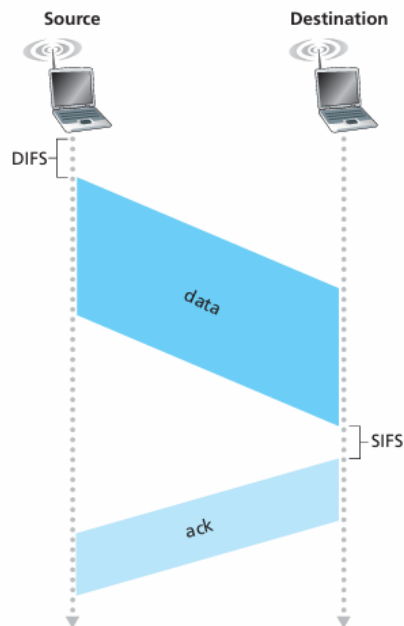


Figure 6.10 ♦ 802.11 uses link-layer acknowledgments

4. If an acknowledgment is received, the transmitting station knows that its frame has been correctly received at the destination station. If the station has another frame to send, it begins the CSMA/CA protocol at step 2. If the acknowledgment isn't received, the transmitting station reenters the back off phase in step 2, with the random value chosen from a larger interval.

Recall that under Ethernet's CSMA/CD, multiple access protocol (Section 5.3.2), a station begins transmitting as soon as the channel is sensed idle. With CSMA/CA, however, the station refrains from transmitting while counting down, even when it senses the channel to be idle. Why do CSMA/CD and CDMA/CA take such different approaches here?

Dealing with Hidden Terminals: RTS and CTS

The 802.11 MAC protocol also includes a nifty (but optional) reservation scheme that helps avoid collisions even in the presence of hidden terminals. Let's investigate this scheme in the context of Figure 6.11, which shows two wireless stations and one access point. Both of the wireless stations are within range of the AP (whose coverage is shown as a shaded circle) and both have associated with the AP. However, due to fading, the signal ranges of wireless stations are limited to the interiors of the shaded circles shown in Figure 6.11. Thus, each of the wireless stations is hidden from the other, although neither is hidden from the AP.

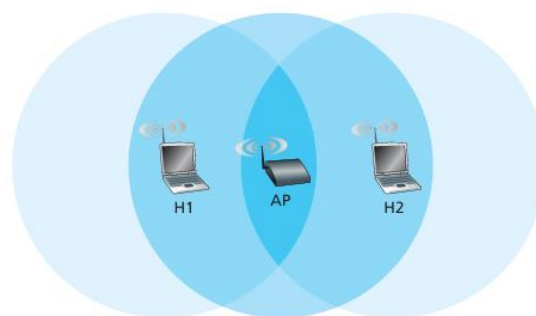


Figure 6.11 ♦ Hidden terminal example: H1 is hidden from H2, and vice versa

Let's now consider why hidden terminals can be problematic. Suppose Station H1 is transmitting a

frame and halfway through H1's transmission, Station H2 wants to send a frame to the AP. H2, not hearing the transmission from H1, will first wait a DIFS interval and then transmit the frame, resulting in a collision. The channel will therefore be wasted during the entire period of H1's transmission as well as during H2's transmission.

In order to avoid this problem, the IEEE 802.11 protocol allows a station to use a short **Request to Send (RTS)** control frame and a **short Clear to Send (CTS)** control frame to reserve access to the channel. When a sender wants to send a DATA frame, it can first send an RTS frame to the AP, indicating the total time required to transmit the DATA frame and the acknowledgment (ACK) frame. When the AP receives the RTS frame, it responds by broadcasting a CTS frame. This CTS frame serves two purposes: It gives the sender explicit permission to send and also instructs the other stations not to send for the reserved duration.

Thus, in Figure 6.12, before transmitting a DATA frame, H1 first broadcasts an RTS frame, which is heard by all stations in its circle, including the AP. The AP then responds with a CTS frame, which is heard by all stations within its range, including H1 and H2. Station H2, having heard the CTS, refrains from transmitting for the time specified in the CTS frame. The RTS, CTS, DATA, and ACK frames are shown in Figure 6.12.

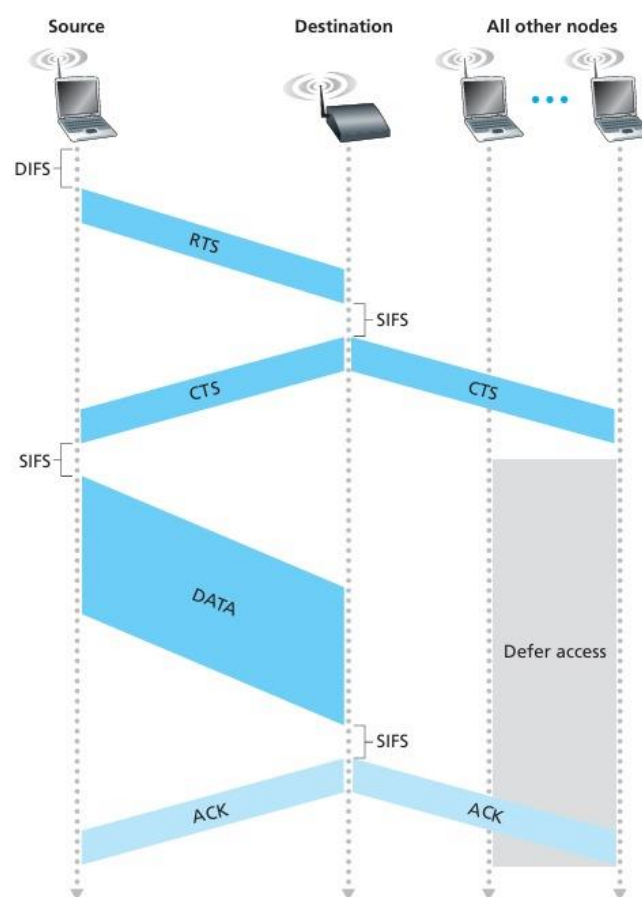


Figure 6.12 ♦ Collision avoidance using the RTS and CTS frames

The use of the RTS and CTS frames can improve performance in two important ways:

- The hidden station problem is mitigated, since a long DATA frame is transmitted only after the channel has been reserved.
- Because the RTS and CTS frames are short, a collision involving an RTS or CTS frame will last only for the duration of the short RTS or CTS frame. Once the RTS and CTS frames are correctly transmitted, the following DATA and ACK frames should be transmitted without collisions.

Our discussion so far has focused on the use of 802.11 in a multiple access setting. We should mention that if two nodes each have a directional antenna, they can point their directional antennas at each other and run the 802.11 protocol over what is essentially a point-to-point link. Given the low cost of commodity 802.11 hardware, the use of directional antennas and an increased transmission power allow 802.11 to be used as an inexpensive means of providing wireless point-to-point connections over tens of kilometers distance. [Raman 2007] describes such a multi-hop wireless network operating in the rural Ganges plains in India that contains point-to-point 802.11 links.

IEEE 802.11 Frame

Although the 802.11 frame shares many similarities with an Ethernet frame, it also contains a number of fields that are specific to its use for wireless links. The 802.11 frame is shown in Figure 6.13. The numbers above each of the fields in the frame represent the lengths of the fields in bytes; the numbers above each of the subfields in the frame control field represent the lengths of the subfields in bits. Let's now examine the fields in the frame as well as some of the more important subfields in the frame's control field.

Payload and CRC Fields

At the heart of the frame is the payload, which typically consists of an IP datagram or an ARP packet. Although the field is permitted to be as long as 2,312 bytes, it is typically fewer than 1,500 bytes, holding an IP datagram or an ARP packet. As with an Ethernet frame, an 802.11 frame includes a 32-bit cyclic redundancy check (CRC) so that the receiver can detect bit errors in the received frame. As we've seen, bit errors are much more common in wireless LANs than in wired LANs, so the CRC is even more useful here.

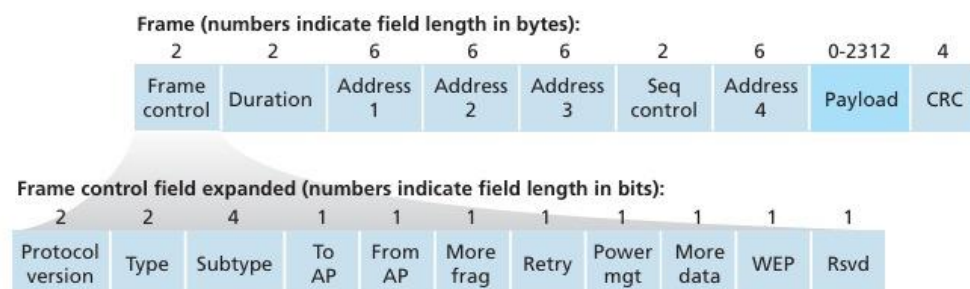


Figure 6.13 ♦ The 802.11 frame

Address Fields

Perhaps the most striking difference in the 802.11 frame is that it has four address fields, each of which can hold a 6-byte MAC address. But why four address fields? Doesn't a source MAC field and destination MAC field suffice, as they do for Ethernet? It turns out that three address fields are needed for internetworking purposes—specifically, for moving the network-layer datagram from a wireless station through an AP to a router interface. The fourth address field is used when APs forward frames to each other in ad hoc mode. Since we are only considering infrastructure networks here, let's focus our attention on the first three address fields. The 802.11 standard defines these fields as follows:

- Address 2 is the MAC address of the station that transmits the frame. Thus, if a wireless station transmits the frame, that station's MAC address is inserted in the address 2 field. Similarly, if an AP transmits the frame, the AP's MAC address is inserted in the address 2 field.
- Address 1 is the MAC address of the wireless station that is to receive the frame. Thus if a mobile wireless station transmits the frame, address 1 contains the MAC address of the destination AP.

Similarly, if an AP transmits the frame, address 1 contains the MAC address of the destination wireless station.

- To understand address 3, recall that the BSS (consisting of the AP and wireless stations) is part of a subnet, and that this subnet connects to other subnets via some router interface. Address 3 contains the MAC address of this router interface.

To gain further insight into the purpose of address 3, let's walk through an internetworking example in the context of Figure 6.14. In this figure, there are two APs, each of which is responsible for a number of wireless stations. Each of the APs has a direct connection to a router, which in turn connects to the global Internet. We should keep in mind that an AP is a link-layer device, and thus neither "speaks" IP nor understands IP addresses. Consider now moving a datagram from the router interface R1 to the wireless Station H1. The router is not aware that there is an AP between it and H1; from the router's perspective, H1 is just a host in one of the sub nets to which it (the router) is connected.

- The router, which knows the IP address of H1 (from the destination address of the datagram), uses ARP to determine the MAC address of H1, just as in an ordinary Ethernet LAN. After obtaining H1's MAC address, router interface R1 encapsulates the datagram within an Ethernet frame. The source address field of this frame contains R1's MAC address, and the destination address field contains H1's MAC address.

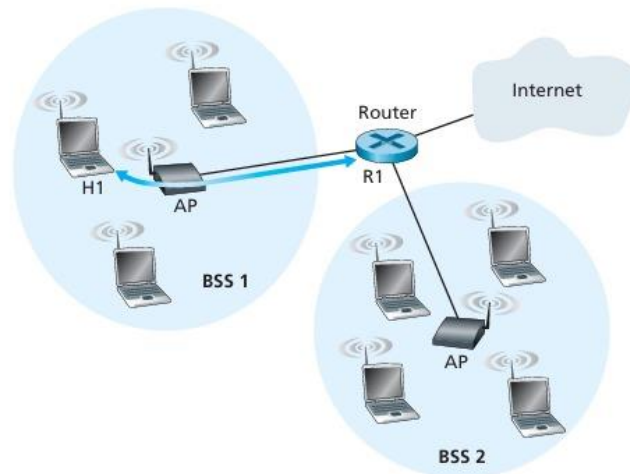


Figure 6.14 ♦ The use of address fields in 802.11 frames: Sending frames between H1 and R1

- When the Ethernet frame arrives at the AP, the AP converts the 802.3 Ethernet frame to an 802.11 frame before transmitting the frame into the wireless channel. The AP fills in address 1 and address 2 with H1's MAC address and its own MAC address, respectively, as described above. For address 3, the AP inserts the MAC address of R1. In this manner, H1 can determine (from address 3) the MAC address of the router interface that sent the datagram into the subnet.

Now consider what happens when the wireless station H1 responds by moving a datagram from H1 to R1.

- H1 creates an 802.11 frame, filling the fields for address 1 and address 2 with the AP's MAC address and H1's MAC address, respectively, as described above. For address 3, H1 inserts R1's MAC address.
- When the AP receives the 802.11 frame, it converts the frame to an Ethernet frame. The source address field for this frame is H1's MAC address, and the destination address field is R1's MAC address. Thus, address 3 allows the AP to determine the appropriate destination MAC address when constructing the Ethernet frame.

In summary, address 3 plays a crucial role for internetworking the BSS with a wired LAN.

Sequence Number, Duration, and Frame Control Fields

Recall that in 802.11, whenever a station correctly receives a frame from another station, it sends back an acknowledgment. Because acknowledgments can get lost, the sending station may send multiple copies of a given frame. As we saw in our discussion of the rdt2.1 protocol (Section 3.4.1), the use of sequence numbers allows the receiver to distinguish between a newly transmitted frame and the retransmission of a previous frame. The sequence number field in the 802.11 frame thus serves exactly the same purpose here at the link layer as it did in the transport layer in Chapter 3.

Recall that the 802.11 protocol allows a transmitting station to reserve the channel for a period of time that includes the time to transmit its data frame and the time to transmit an acknowledgment. This duration value is included in the frame's duration field (both for data frames and for the RTS and CTS frames).

As shown in Figure 6.13, the frame control field includes many subfields. We'll say just a few words about some of the more important subfields; for a more complete discussion, you are encouraged to consult the 802.11 specification [Held 2001; Crow 1997; IEEE 802.11 1999]. The type and subtype fields are used to distinguish the association, RTS, CTS, ACK, and data frames. The to and from fields are used to define the meanings of the different address fields. (These meanings change depending on whether ad hoc or infrastructure modes are used and, in the case of infrastructure mode, whether a wireless station or an AP is sending the frame.) Finally the WEP field indicates whether encryption is being used or not. (WEP is discussed in Chapter 8.)