MODULE 2

DATA LINK LAYER

3.1 DATA LINK LAYER DESIGN ISSUES

The data link layer uses the services of the physical layer to send and receive bits over communication channels. It has a number of functions, including:

1. Providing a well-defined service interface to the network layer.

2. Dealing with transmission errors

3. Regulating the flow of data so that slow receivers are not swamped by fast senders.

To accomplish these goals, the data link layer takes the packets it gets from the network layer and encapsulates them into frames for transmission. Each frame contains a frame header, a payload field for holding the packet, and a frame trailer, as illustrated in Fig. 3-1.

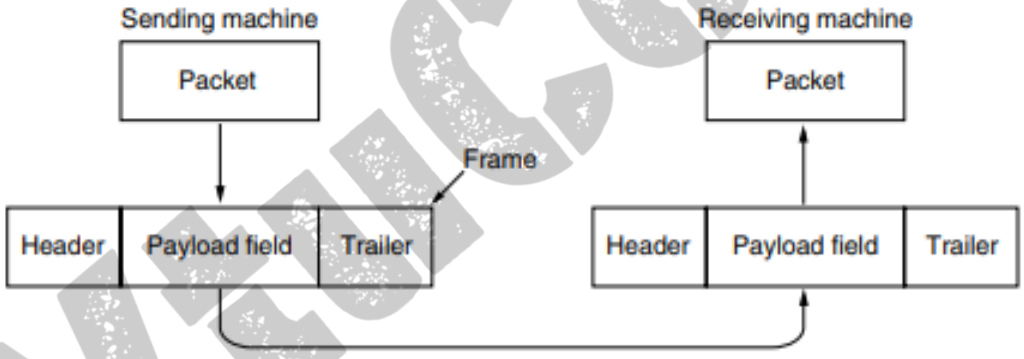


Figure 3-1. Relationship between packets and frames.

3.1.1 Services Provided to the Network Layer

The function of the data link layer is to provide services to the network layer. The principal service is transferring data from the network layer on the source machine to the network layer on the destination machine. On the source machine is an entity, call it a process, in the network layer that hands some bits to the data link layer for transmission to the destination. The job of the data link layer is to transmit the bits to the destination machine so they can be handed over to the network layer there, as shown in Fig. 3-2(a). The actual transmission follows the path of Fig. 3-2(b), but it is easier to think in terms of two data link layer processes communicating using a data link protocol

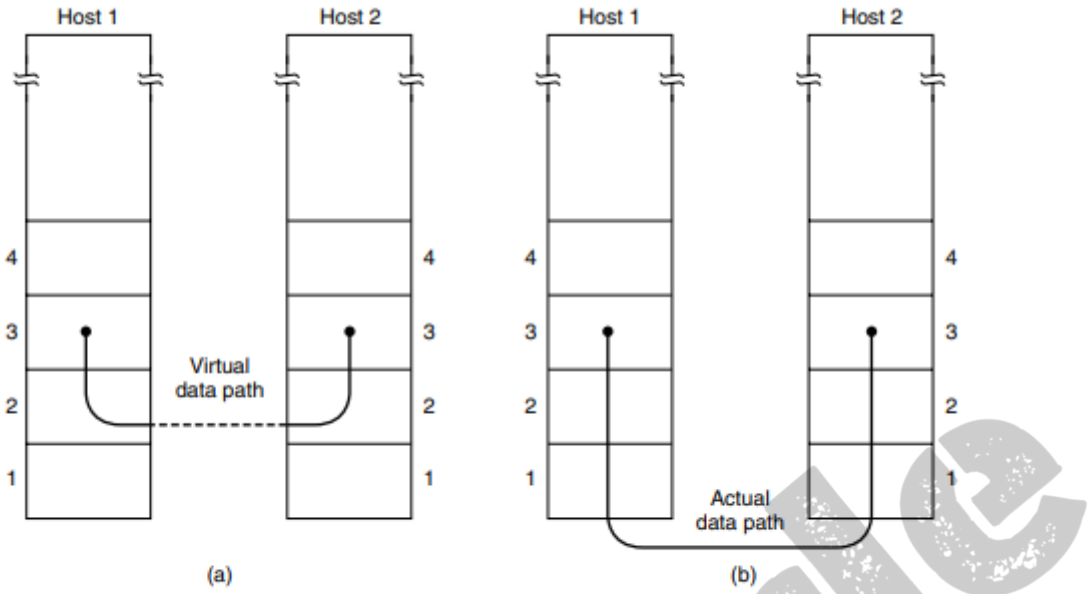


Figure 3-2. (a) Virtual communication. (b) Actual communication.

The data link layer can be designed to offer various services. The actual services that are offered vary from protocol to protocol.

Three reasonable possibilities that we will consider in turn are:

1. Unacknowledged connectionless service.

2. Acknowledged connectionless service.

3. Acknowledged connection-oriented service.

Unacknowledged connectionless service consists of having the source machine send independent frames to the destination machine without having the destination machine acknowledge them. No logical connection is established beforehand or released afterward. If a frame is lost due to noise on the line, no attempt is made to detect the loss or recover from it in the data link layer.

Acknowledged connectionless service. When this service is offered, there are still no logical connections used, but each frame sent is individually acknowledged. In this way, the sender knows whether a frame has arrived correctly or been lost. If it has not arrived within a specified time interval, it can be sent again.

The most sophisticated service the data link layer can provide to the network layer is connection-oriented service. With this service, the source and destination machines establish a connection before any data are transferred. Each frame sent over the connection is numbered, and the data link layer guarantees that each frame sent is indeed received. Furthermore, it guarantees that each frame is received exactly once and that all frames are received in the right order. Connection-oriented

service thus provides the network layer processes with the equivalent of a reliable bit stream. It is appropriate over long, unreliable links such as a satellite channel or a long-distance telephone circuit.

3.1.2 FRAMING

The bit stream received by the data link layer is not guaranteed to be error free. Some bits may have different values and the number of bits received may be less than, equal to, or more than the number of bits transmitted. It is up to the data link layer to detect and, if necessary, correct errors. The usual approach is for the data link layer to break up the bit stream into discrete frames, compute a short token called a checksum for each frame, and include the checksum in the frame when it is transmitted. When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it . A good design must make it easy for a receiver to find the start of new frames while using little of the channel bandwidth. We will look at four methods:

1. Byte count.

2. Flag bytes with byte stuffing.

3. Flag bits with bit stuffing

. 4. Physical layer coding violations.

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

The trouble with this algorithm is that the count can be garbled by a transmission error. For example, if the byte count of 5 in the second frame of Fig. 3-3(b) becomes a 7 due to a single bit flip, the destination will get out of synchronization. It will then be unable to locate the correct start of the next frame. Even if the checksum is incorrect so the destination knows that the frame is bad, it still has no way of telling where the next frame starts. Sending a frame back to the source asking for a retransmission does not help either

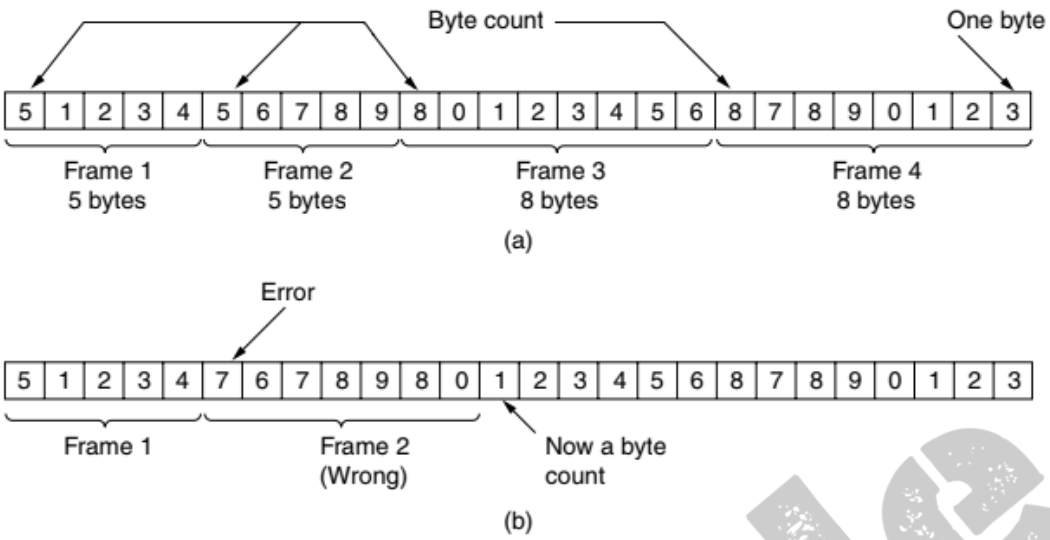


Figure 3-3. A byte stream. (a) Without errors. (b) With one error.

The second framing method gets around the problem of resynchronization after an error by having each frame start and end with special bytes. Often the same byte, called a flag byte, is used as both the starting and ending delimiter. This byte is shown in Fig. 3-4(a) as FLAG.

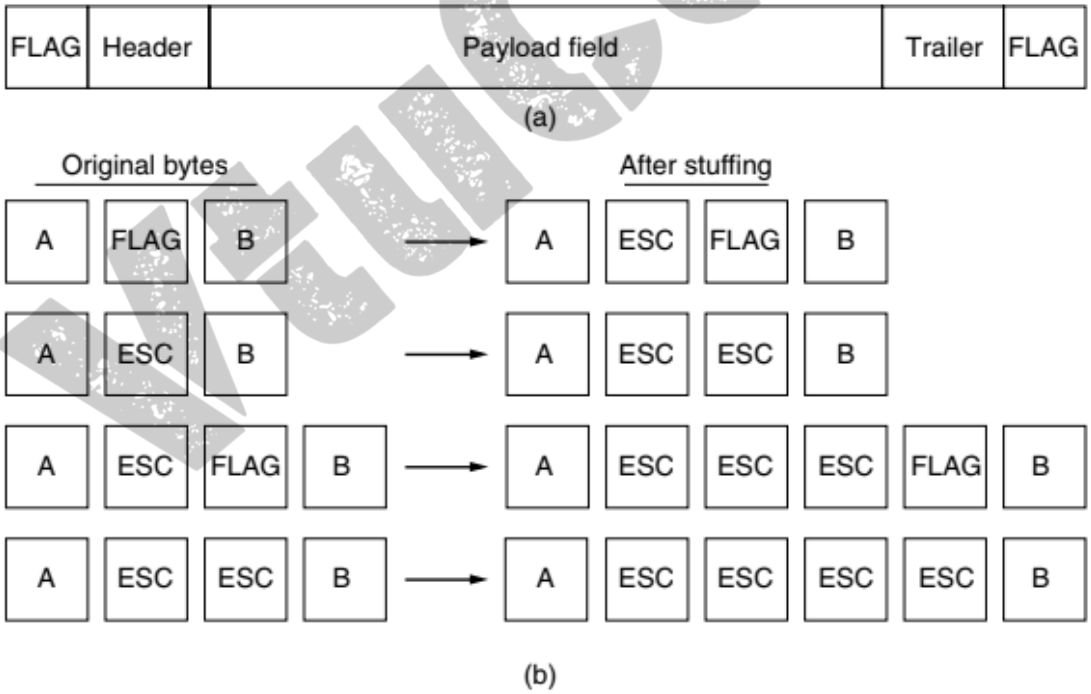


Figure 3-4. (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.

Two consecutive flag bytes indicate the end of one frame and the start of the next. Thus, if the receiver ever loses synchronization it can just search for two flag bytes to find the end of the current frame and the start of the next frame. However, there is a still a problem we have to solve. It may happen that the flag byte occurs in the data, especially when binary data such as photographs or songs are being transmitted. This situation would interfere with the framing. One way to solve this problem is to have the sender's data link layer insert a special escape byte (ESC) just before each ''accidental'' flag byte in the data. Thus, a framing flag byte can be distinguished from one in the data by the absence or presence of an escape byte before it. The data link layer on the receiving end removes the escape bytes before giving the data to the network layer. This technique is called byte stuffing. The byte-stuffing scheme depicted in Fig. 3-4 is a slight simplification of the one used in PPP (Point-to-Point Protocol), which is used to carry packets over communications links.

The third method of delimiting the bit stream gets around a disadvantage of byte stuffing, which is that it is tied to the use of 8-bit bytes. Framing can be also be done at the bit level, so frames can contain an arbitrary number of bits made up of units of any size. It was developed for the once very popular HDLC (Highlevel Data Link Control) protocol. Each frame begins and ends with a special bit pattern, 01111110 or 0x7E in hexadecimal. This pattern is a flag byte. Whenever the sender's data link layer encounters five consecutive 1s in the data, it automatically stuffs a 0 bit into the outgoing bit stream. This bit stuffing is analogous to byte stuffing, in which an escape byte is stuffed into the outgoing character stream before a flag byte in the data. It also ensures a minimum density of transitions that help the physical layer maintain synchronization. USB (Universal Serial Bus) uses bit stuffing for this reason. When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit. Just as byte stuffing is completely transparent to the network layer in both computers, so is bit stuffing. If the user data contain the flag pattern, 01111110, this flag is transmitted as 011111010 but stored in the receiver's memory as 01111110. Figure 3-5 gives an example of bit stuffing.

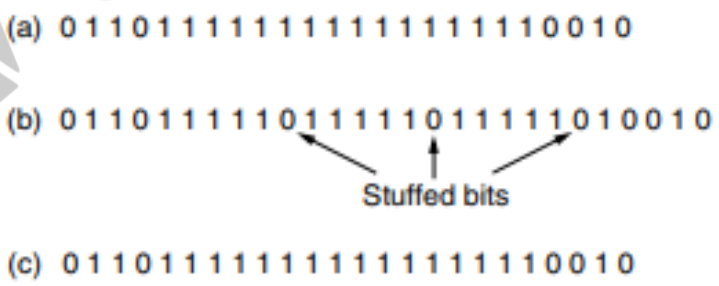


Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

3.1.3 Error Control

The usual way to ensure reliable delivery is to provide the sender with some feedback about what is happening at the other end of the line. Typically, the protocol calls for the receiver to send back special control frames bearing positive or negative acknowledgements about the incoming frames. If the sender receives a positive acknowledgement about a frame, it knows the frame has arrived safely. On the other hand, a negative acknowledgement means that something has gone wrong and the frame must be transmitted again. , if the acknowledgement frame is lost, the sender will not know how to proceed. It should be clear that a protocol in which the sender transmits a frame and then waits for an acknowledgement, positive or negative, will hang forever if a frame is ever lost due.

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

3.1.4 Flow Control

Another important design issue that occurs in the data link layer (and higher layers as well) is what to do with a sender that systematically wants to transmit frames faster than the receiver can accept them. This situation can occur when the sender is running on a fast, powerful computer and the receiver is running on a slow, low-end machine.

Two approaches are commonly used. In the first one, feedback-based flow control, the receiver sends back information to the sender giving it permission to send more data, or at least telling the sender how the receiver is doing. In the second one, rate-based flow control, the protocol has a built-in mechanism that limits the rate at which senders may transmit data, without using feedback from the receiver.

3.2 ERROR DETECTION AND CORRECTION

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

One model is that errors are caused by extreme values of thermal noise that overwhelm the signal briefly and occasionally, giving rise to isolated single-bit errors. Another model is that errors tend to come in bursts rather than singly. This model follows from the physical processes that generate

them—such as a deep fade on a wireless channel or transient electrical interference on a wired channel

Sometimes, the location of an error will be known, perhaps because the physical layer received an analog signal that was far from the expected value for a 0 or 1 and declared the bit to be lost. This situation is called an erasure channel.

3.2.1 Error-Correcting Codes

We will examine four different error-correcting codes:

1. Hamming codes.
2. Binary convolutional codes.
3. Reed-Solomon codes.
4. Low-Density Parity Check codes.

All of these codes add redundancy to the information that is sent. A frame consists of m data (i.e., message) bits and r redundant (i.e. check) bits. In a block code, the r check bits are computed solely as a function of the m data bits with which they are associated, as though the m bits were looked up in a large table to find their corresponding r check bits. In a systematic code, the m data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent. In a linear code, the r check bits are computed as a linear function of the m data bits.

Let the total length of a block be n (i.e., n = m + r). We will describe this as an (n,m) code. An n-bit unit containing data and check bits is referred to as an nbit codeword. The code rate, or simply rate, is the fraction of the codeword that carries information that is not redundant, or m/n. The rates used in practice vary widely. They might be 1/2 for a noisy channel, in which case half of the received information is redundant, or close to 1 for a high-quality channel, with only a small number of check bits added to a large message. To understand how errors can be handled, it is necessary to first look closely at what an error really is. Given any two codewords that may be transmitted or received—say, 10001001 and 10110001—it is possible to determine how many corresponding bits differ. In this case, 3 bits differ. To determine how many bits differ, just XOR the two codewords and count the number of 1 bits in the result. For example:

```
10001001
10110001
00111000
```

The number of bit positions in which two codewords differ is called the Hamming distance Its significance is that if two codewords are a Hamming distance d apart, it will require d single-bit errors to convert one into the other.

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

As a simple example of an error-detecting code, consider a code in which a single parity b appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101. A code with a single parity bit has a distance 2, since any single-bit error produces a codeword with the wrong parity. It can be used to detect single errors. As a simple example of an error-correcting code, consider a code with only four valid codewords: 0000000000, 0000011111, 1111100000, and 1111111111 This code has a distance 5, which means that it can correct double errors. If the codeword 0000000111 arrives, the receiver knows that the original must have been 0000011111. If, however, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly

In Hamming codes the bits of the codeword are numbered consecutively, starting with bit 1 at the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. This pattern is shown for an (11,7) Hamming code with 7 data bits and 4 check bits in Fig. 3-6. Each check bit forces the modulo 2 sum, or parity, of some collection of bits, including itself, to be even (or odd). A bit may be included in several check bit computations.

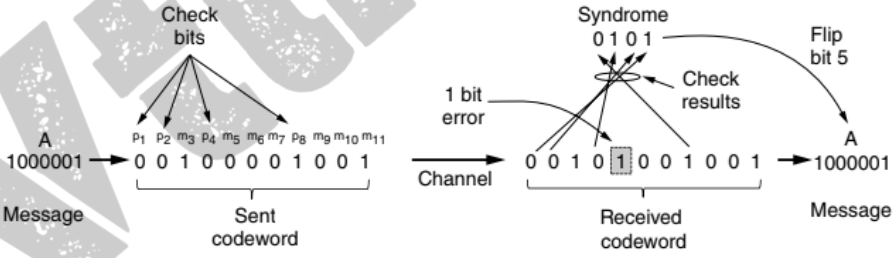


Figure 3-6. Example of an (11, 7) Hamming code correcting a single-bit error.

To see which check bits the data bit in position k contributes to, rewrite k as a sum of powers of 2. For example, 11 = 1 + 2 + 8 and 29 = 1 + 4 + 8 + 16. A bit is checked by just those check bits occurring in its expansion (e.g., bit 11 is checked by bits 1, 2, and 8). In the example, the check bits are computed for even parity sums for a message that is the ASCII letter ‘‘A.’’

Convolutional codes are widely used in deployed networks, for example, as part of the GSM mobile phone system, in satellite communications, and in 802.11. As an example, a popular

convolutional code is shown in Fig. 3-7. This code is known as the NASA convolutional code of r = 1/2 and k = 7

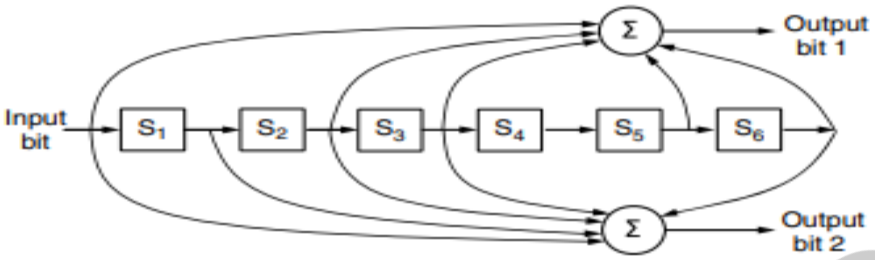


Figure 3-7. The NASA binary convolutional code used in 802.11.

In Fig. 3-7, each input bit on the left-hand side produces two output bits on the right-hand side that are XOR sums of the input and internal state. Since it deals with bits and performs linear operations, this is a binary, linear convolutional code. Since 1 input bit produces 2 output bits, the code rate is 1/2. It is not systematic since none of the output bits is simply the input bit. The internal state is kept in six memory registers. Each time another bit is input the values in the registers are shifted to the right. For example, if 111 is input and the initial state is all zeros, the internal state, written left to right, will become 100000, 110000, and 111000 after the first, second, and third bits have been input. The output bits will be 11, followed by 10, and then 01. It takes seven shifts to flush an input completely so that it does not affect the output. The constraint length of this code is thus k = 7.

3.2.2 Error-Detecting Codes

Without error-correcting codes, it would be hard to get anything through. However, over fiber or high-quality copper, the error rate is much lower, so error detection and retransmission is usually more efficient there for dealing with the occasional error. We will examine three different error-detecting codes. They are all linear, systematic block codes:

1. Parity.

2. Checksums.

3. Cyclic Redundancy Checks (CRCs)

The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). Doing this is equivalent to computing the (even) parity bit as the modulo 2 sum or XOR of the data bits. For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101. A code with a single parity bit has a distance of 2, since any single-bit error produces a codeword with the wrong parity. This means that it can detect

single-bit errors. One difficulty with this scheme is that a single parity bit can only reliably detect a single-bit error in the block. If the block is badly garbled by a long burst error, the probability that the error will be detected is only 0.5, which is hardly acceptable However, there is something else we can do that provides better protection against burst errors: we can compute the parity bits over the data in a different order than the order in which the data bits are transmitted. Doing so is called interleaving. In this case, we will compute a parity bit for each of the n columns and send all the data bits as k rows, sending the rows from top to bottom and the bits in each row from left to right in the usual manner. At the last row, we send the n parity bits. This transmission order is shown in Fig. 3-8 for n = 7 and k = 7

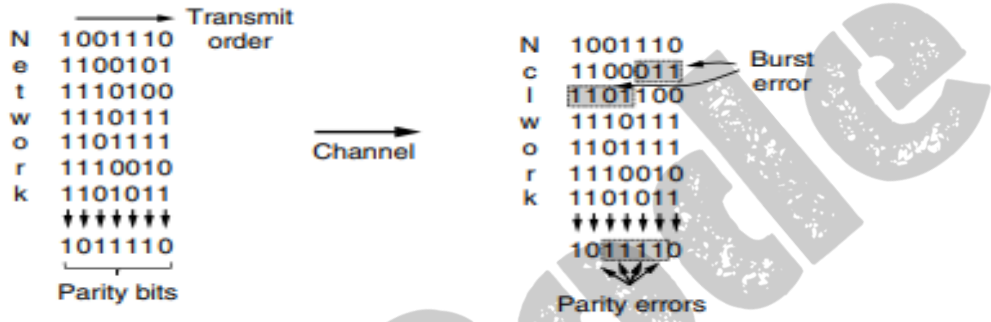


Figure 3-8. Interleaving of parity bits to detect a burst error.

In Fig. 3-8, when a burst error of length n = 7 occurs, the bits that are in error are spread across different columns. (A burst error does not imply that all the bits are wrong; it just implies that at least the first and last are wrong. In Fig. 3-8, 4 bits were flipped over a range of 7 bits.) At most 1 bit in each of the n columns will be affected, so the parity bits on those columns will detect the error. This method uses n parity bits on blocks of kn data bits to detect a single burst error of length n or less

The word ‘‘checksum’’ is often used to mean a group of check bits associated with a message, regardless of how are calculated. A group of parity bits is one example of a checksum. However, there are other, stronger checksums based on a running sum of the data bits of the message. The checksum is usually placed at the end of the message, as the complement of the sum function. This way, errors may be detected by summing the entire received codeword, both data bits and checksum. If the result comes out to be zero, no error has been detected.

One example of a checksum is the 16-bit Internet checksum used on all Internet packets as part of the IP protocol (Braden et al., 1988). This checksum is a sum of the message bits divided into 16-bit words. Because this method operates on words rather than on bits, as in parity, errors that leave the parity unchanged can still alter the sum and be detected. For example, if the lowest order bit in two different words is flipped from a 0 to a 1, a parity check across these bits would fail to detect an error. However, two 1s will be added to the 16-bit checksum to produce a different result. The error can then be detected

The CRC (Cyclic Redundancy Check), also known as a polynomial code. Polynomial codes are based upon treating bit strings as representations of polynomials with coefficients of 0 and 1 only. A k-bit frame is regarded as the coefficient list for a polynomial with k terms, ranging from x k − 1 to x 0. Such a polynomial is said to be of degree k − 1. The high-order (leftmost) bit is the coefficient of x k − 1, the next bit is the coefficient of x k − 2, and so on. For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients 1, 1, 0, 0, 0, and 1: 1x 5 + 1x 4 + 0x 3 + 0x 2 + 0x 1 + 1x 0. Polynomial arithmetic is done modulo 2, according to the rules of algebraic field theory. It does not have carries for addition or borrows for subtraction. Both addition and subtraction are identical to exclusive OR. For example:

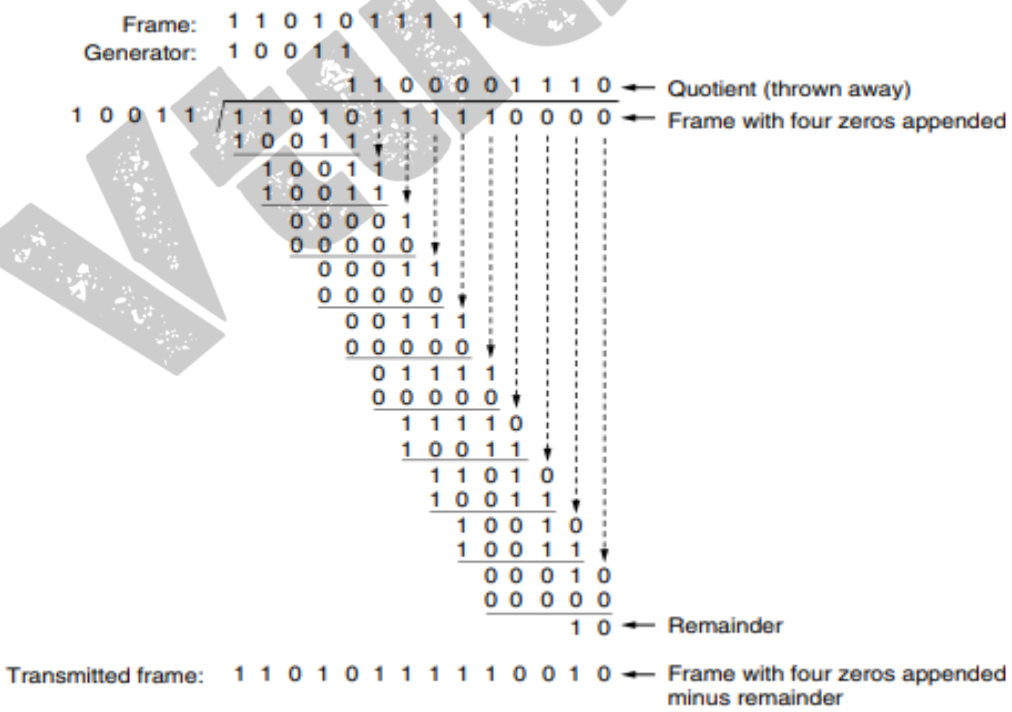
The algorithm for computing the CRC is as follows:

1. Let r be the degree of G(x). Append r zero bits to the low-order end of the frame so it now contains m + r bits and corresponds to the polynomial x r M(x)

2. Divide the bit string corresponding to G(x) into the bit string corresponding to x r M(x), using modulo 2 division.

3. Subtract the remainder (which is always r or fewer bits) from the bit string corresponding to x r M(x) using modulo 2 subtraction.

The result is the checksummed frame to be transmitted. Call its polynomial T(x). Figure 3-9 illustrates the calculation for a frame 1101011111 using the generator G(x) = x 4 + x + 1.



3.3 ELEMENTARY DATA LINK PROTOCOLS

The physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth. A common implementation is shown in Fig. 3-10. The physical layer process and some of the data link layer process run on dedicate hardware called a NIC (Network Interface Card). The rest of the link layer process and the network layer process run on the main CPU as part of the operating system, with the software for the link layer process often taking the form of a device driver. However, other implementations are also possible (e.g., three processes offloaded to dedicated hardware called a network accelerator, or three processes running on the main CPU on a software-defined ratio). Actually, the preferred implementation changes from decade to decade with technology trade-offs. In any event, treating the three layers as separate processes makes the discussion conceptually cleaner and also serves to emphasize the independence of the layers.

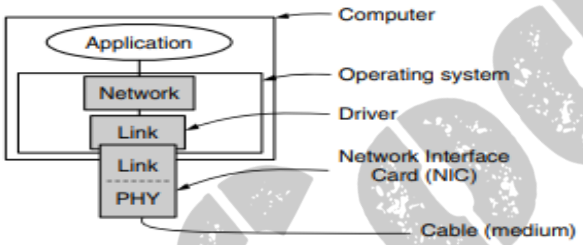


Figure 3-10. Implementation of the physical, data link, and network layers.

When the data link layer accepts a packet, it encapsulates the packet in a frame by adding a data link header and trailer to it (see Fig. 3-1). Thus, a frame consists of an embedded packet, some control information (in the header), and a checksum (in the trailer). The frame is then transmitted to the data link layer on the other machine.

```
#define MAX_PKT 1024                                        /* determines packet size in bytes */

typedef enum {false, true} boolean;                         /* boolean type */
typedef unsigned int seq_nr;                                /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;       /* packet definition */
typedef enum {data, ack, nak} frame_kind;                   /* frame_kind definition */

typedef struct {                                            /* frames are transported in this layer */
  frame_kind kind;                                          /* what kind of frame is it? */
  seq_nr seq;                                               /* sequence number */
  seq_nr ack;                                               /* acknowledgement number */
  packet info;                                              /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Figure 3-11. Some definitions needed in the protocols to follow. These definitions are located in the file *protocol.h*.

Five data structures are defined there: boolean, seq nr, packet, frame kind, and frame. A boolean is an enumerated type and can take on the values true and false. A seq nr is a small integer used to number the frames so that we can tell them apart. These sequence numbers run from 0 up to and including MAX SEQ, which is defined in each protocol needing it. A packet is the unit of information exchanged between the network layer and the data link layer on the same machine, or between network layer peers. In our model it always contains MAX PKT bytes, but more realistically it would be of variable length. A frame is composed of four fields: kind, seq, ack, and info, the first three of which contain control information and the last of which may contain actual data to be transferred. These control fields are collectively called the frame header.

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

The procedure wait for event sits in a tight loop waiting for something to happen, as mentioned earlier. The procedures to network layer and from network layer are used by the data link layer to pass packets to the network layer and accept packets from the network layer, respectively. Note that from physical layer and to physical layer pass frames between the data link layer and the physical layer. In other words, to network layer and from network layer deal with the interface between layers 2 and 3, whereas from physical layer and to physical layer deal with the interface between layers 1 and 2.

The procedure wait for event to return event = timeout. The procedures start timer and stop timer turn the timer on and off, respectively. Timeout events are possible only when the timer is running and before stop timer is called. It is explicitly permitted to call start timer while the timer is running; such a call simply resets the clock to cause the next timeout after a full timer interval has elapsed (unless it is reset or turned off). The procedures start ack timer and stop ack timer control an auxiliary timer used to generate acknowledgements under certain conditions. The procedures enable network layer and disable network layer are used in the more sophisticated protocols

Frame sequence numbers are always in the range 0 to MAX SEQ (inclusive), where MAX SEQ is different for the different protocols. It is frequently necessary to advance a sequence number by 1 circularly (i.e., MAX SEQ is followed by 0). The macro inc performs this incrementing. It has been defined as a macro because it is used in-line within the critical path.

3.3.1 A Utopian Simplex Protocol

The protocol consists of two distinct procedures, a sender and a receiver. The sender runs in the data link layer of the source machine, and the receiver runs in the data link layer of the destination machine. No sequence numbers or acknowledgements are used here, so MAX SEQ is not needed. The only event type possible is frame arrival . The sender is in an infinite while loop just pumping data out onto the line as fast as it can. The body of the loop consists of three actions: go fetch a packet from the (always obliging) network layer, construct an outbound frame using the variable s, and send the frame on its way. Only the info field of the frame is used by this protocol, because the other fields have to do with error and flow control and there are no errors or flow control restrictions here. The receiver is equally simple. Initially, it waits for something to happen, the only possibility being the arrival of an undamaged frame. Eventually, the frame arrives and the procedure wait for event returns, with event set to frame arrival (which is ignored anyway). The call to from physical layer removes the newly arrived frame from the hardware buffer and puts it in the variable r, where the receiver code can get at it.

```
/* Protocol 1 (Utopia) provides for data transmission in one direction only, from
   sender to receiver. The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

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

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

  while (true) {
      from_network_layer(&buffer);      /* go get something to send */
      s.info = buffer;                  /* copy it into s for transmission */
      to_physical_layer(&s);            /* send it on its way */
  }                                     /* Tomorrow, and tomorrow, and tomorrow,
                                           Creeps in this petty pace from day to day
                                           To the last syllable of recorded time.
                                              – Macbeth, V, v */
}

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

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

Figure 3-12. A utopian simplex protocol.

The utopia protocol is unrealistic because it does not handle either flow control or error correction. Its processing is close to that of an unacknowledged connectionless service that relies on higher layers to solve these problems, though even an unacknowledged connectionless service would do some error detection.

3.3.2 A Simplex Stop-and-Wait Protocol for an Error-Free Channel

Protocols in which the sender sends one frame and then waits for an acknowledgement before proceeding are called stop-and-wait. Figure 3-13 gives an example of a simplex stop-and-wait protocol.This protocol entails a strict alternation of flow: first the sender sends a frame, then the receiver sends a frame, then the sender sends another frame, then the receiver sends another one, and so on. A halfduplex physical channel would suffice here. As in protocol 1, the sender starts out by fetching a packet from the network layer, using it to construct a frame, and sending it on its way. But now, unlike in protocol 1, the sender must wait until an acknowledgement frame arrives before looping back and fetching the next packet from the network layer. The sending data link layer need not even inspect the incoming frame as there is only one possibility. The incoming frame is always an acknowledgement. The only difference between receiver1 and receiver2 is that after delivering a packet to the network layer, receiver2 sends an acknowledgement frame back to the sender before entering the wait loop again.

```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */

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

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

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

void receiver2(void)
{
  frame r, s;                       /* buffers for frames */
  event_type event;                 /* frame_arrival is the only possibility */
  while (true) {
      wait_for_event(&event);       /* only possibility is frame_arrival */
      from_physical_layer(&r);      /* go get the inbound frame */
      to_network_layer(&r.info);    /* pass the data to the network layer */
      to_physical_layer(&s);        /* send a dummy frame to awaken sender */
  }
```

3.3.3 A Simplex Stop-and-Wait Protocol for a Noisy Channel

Consider the following scenario:

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

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

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

4. The duplicate frame also arrives intact at the data link layer on B and is unwittingly passed to the network layer there. If A is sending a file to B, part of the file will be duplicated (i.e., the copy of the file made by B will be incorrect and the error will not have been detected). In other words, the protocol will fail.

Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called ARQ (Automatic Repeat reQuest) or PAR (Positive Acknowledgement with Retransmission). Like protocol 2, this one also transmits data only in one direction. Protocol 3 differs from its predecessors in that both sender and receiver have a variable whose value is remembered while the data link layer is in the wait state. The sender remembers the sequence number of the next frame to send in next frame to send; the receiver remembers the sequence number of the next frame expected in frame expected.

Each protocol has a short initialization phase before entering the infinite loop .After transmitting a frame and starting the timer, the sender waits for something exciting to happen. Only three possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires. If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet. It also advances the sequence number. If a damaged frame arrives or the timer expires, neither the buffer nor the sequence number is changed so that a duplicate can be sent. In all cases, the contents of the buffer (either the next packet or a duplicate) are then sent.

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                       /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

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

  next_frame_to_send = 0;               /* initialize outbound sequence numbers */
  from_network_layer(&buffer);          /* fetch first packet */
  while (true) {
      s.info = buffer;                  /* construct a frame for transmission */
      s.seq = next_frame_to_send;       /* insert sequence number in frame */
      to_physical_layer(&s);            /* send it on its way */
      start_timer(s.seq);               /* if answer takes too long, time out */
      wait_for_event(&event);           /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);      /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);        /* turn the timer off */
              from_network_layer(&buffer); /* get the next one to send */
              inc(next_frame_to_send);  /* invert next_frame_to_send */
          }
      }
  }
}

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

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);           /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {     /* a valid frame has arrived */
          from_physical_layer(&r);      /* go get the newly arrived frame */
          if (r.seq == frame_expected) { /* this is what we have been waiting for */
              to_network_layer(&r.info); /* pass the data to the network layer */
              inc(frame_expected);      /* next time expect the other sequence nr */
          }
          s.ack = 1 - frame_expected;   /* tell which frame is being acked */
          to_physical_layer(&s);        /* send acknowledgement */
      }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

3.4 SLIDING WINDOW PROTOCOLS

In this model the data frames from A to B are intermixed with the acknowledgement frames from A to B. By looking at the kind field in the header of an incoming frame, the receiver can tell whether the frame is data or an acknowledgement. When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the network layer passes it the next packet. The acknowledgement is attached to the outgoing data frame (using the ack field in the frame header). In effect, the acknowledgement gets a free ride on the next outgoing data frame. The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame is known as piggybacking.

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

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

 next_frame_to_send = 0;                            /* initialize outbound sequence numbers */
 from_network_layer(&buffer);                       /* fetch first packet */
 while (true) {
     s.info = buffer;                               /* construct a frame for transmission */
     s.seq = next_frame_to_send;                    /* insert sequence number in frame */
     to_physical_layer(&s);                         /* send it on its way */
     start_timer(s.seq);                            /* if answer takes too long, time out */
     wait_for_event(&event);                        /* frame_arrival, cksum_err, timeout */
     if (event == frame_arrival) {
          from_physical_layer(&s);                  /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
               stop_timer(s.ack);                   /* turn the timer off */
               from_network_layer(&buffer);         /* get the next one to send */
               inc(next_frame_to_send);             /* invert next_frame_to_send */
```

```
            }
        }
    }
}

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

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);                 /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {           /* a valid frame has arrived */
          from_physical_layer(&r);            /* go get the newly arrived frame */
          if (r.seq == frame_expected) {      /* this is what we have been waiting for */
              to_network_layer(&r.info);      /* pass the data to the network layer */
              inc(frame_expected);            /* next time expect the other sequence nr */
          }
          s.ack = 1 – frame_expected;         /* tell which frame is being acked */
          to_physical_layer(&s);              /* send acknowledgement */
      }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

If a new packet arrives quickly, the acknowledgement is piggybacked onto it. Otherwise, if no new packet has arrived by the end of this time period, the data link layer just sends a separate acknowledgement frame. The next three protocols are bidirectional protocols that belong to a class called sliding window protocols. The three differ among themselves in terms of efficiency, complexity, and buffer requirements. In these, as in all sliding window protocols, each outbound frame contains a sequence number, ranging from 0 up to some maximum. The maximum is usually $2n - 1$ so the sequence number fits exactly in an n-bit field. The stop-and-wait sliding window protocol uses $n = 1$, restricting the sequence numbers to 0 and 1.

The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send. These frames are said to

fall within the sending window. Similarly, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept. The sender's window and the receiver's window need not have the same lower and upper limits or even have the same size.

The sequence numbers within the sender's window represent frames that have been sent or can be sent but are as yet not acknowledged. Whenever a new packet arrives from the network layer, it is given the next highest sequence number, and the upper edge of the window is advanced by one. When an acknowledgement comes in, the lower edge is advanced by one. In this way the window continuously maintains a list of unacknowledged frames. Figure 3-15 shows an example with a maximum window size of 1. Initially, no frames are outstanding, so the lower and upper edges of the sender's window are equal, but as time goes on, the situation progresses as shown. Unlike the sender's window, the receiver's window always remains at its initial size, rotating as the next frame is accepted and delivered to the network layer.

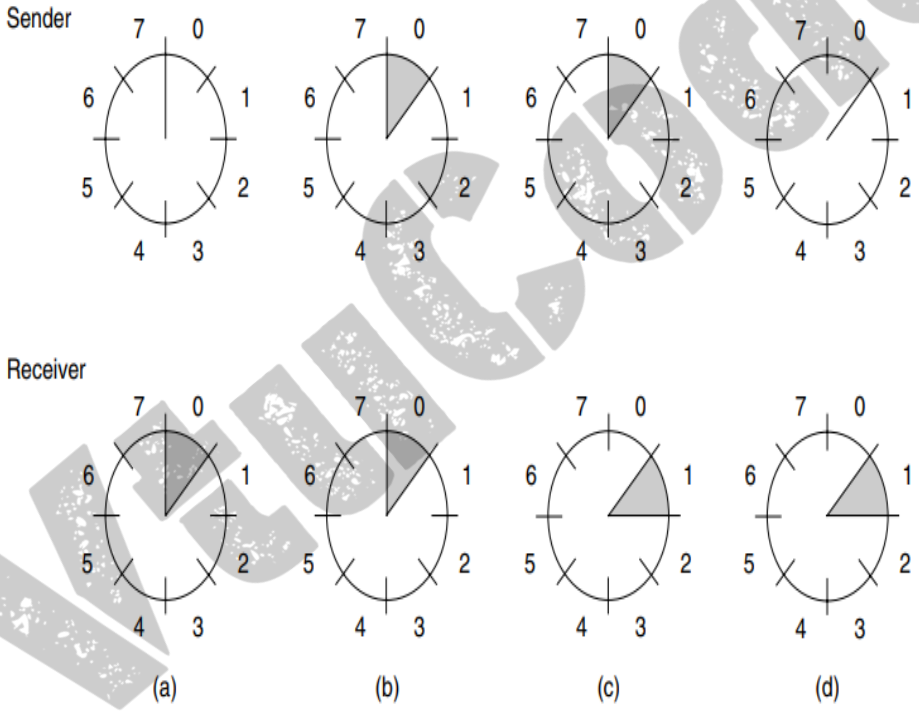


Figure 3-15. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

3.4.1 A One-Bit Sliding Window Protocol

A sliding window protocol with a window size of 1. Such a protocol uses stop-and-wait since the sender transmits a frame and waits for its acknowledgement before sending the next one. Figure 3-16 depicts such a protocol. Like the others, it starts out by defining some variables. Next frame to send tells which frame the sender is trying to send. Similarly, frame expected tells which frame the receiver is expecting. In both cases, 0 and 1 are the only possibilities.

```
/* Protocol 4 (Sliding window) is bidirectional. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
 seq_nr next_frame_to_send;                         /* 0 or 1 only */
 seq_nr frame_expected;                             /* 0 or 1 only */
 frame r, s;                                        /* scratch variables */
 packet buffer;                                     /* current packet being sent */
 event_type event;
 next_frame_to_send = 0;                            /* next frame on the outbound stream */
 frame_expected = 0;                                /* frame expected next */
 from_network_layer(&buffer);                       /* fetch a packet from the network layer */
 s.info = buffer;                                   /* prepare to send the initial frame */
 s.seq = next_frame_to_send;                        /* insert sequence number into frame */
 s.ack = 1 – frame_expected;                        /* piggybacked ack */
 to_physical_layer(&s);                             /* transmit the frame */
 start_timer(s.seq);                                /* start the timer running */

  while (true) {
      wait_for_event(&event);                       /* frame_arrival, cksum_err, or timeout */
      if (event == frame_arrival) {                 /* a frame has arrived undamaged */
          from_physical_layer(&r);                  /* go get it */
          if (r.seq == frame_expected) {            /* handle inbound frame stream */
              to_network_layer(&r.info);            /* pass packet to network layer */
              inc(frame_expected);                  /* invert seq number expected next */
          }
          if (r.ack == next_frame_to_send) {        /* handle outbound frame stream */
              stop_timer(r.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* fetch new pkt from network layer */
              inc(next_frame_to_send);              /* invert sender's sequence number */
          }
      }
      s.info = buffer;                              /* construct outbound frame */
      s.seq = next_frame_to_send;                   /* insert sequence number into it */
      s.ack = 1 – frame_expected;                   /* seq number of last received frame */
      to_physical_layer(&s);                        /* transmit a frame */
      start_timer(s.seq);                           /* start the timer running */
  }
}
```

Assume that computer A is trying to send its frame 0 to computer B and that B is trying to send its frame 0 to A. Suppose that A sends a frame to B, but A's timeout interval is a little too short.

Consequently, A may time out repeatedly, sending a series of identical frames, all with seq = 0 and ack = 1. When the first valid frame arrives at computer B, it will be accepted and frame expected will be set to a value of 1. All the subsequent frames received will be rejected because B is now expecting frames with sequence number 1, not 0. Furthermore, since all the duplicates will have ack = 1 and B is still waiting for an acknowledgement of 0, B will not go and fetch a new packet from its network layer. After every rejected duplicate comes in, B will send A a frame containing seq = 0 and ack = 0. Eventually, one of these will arrive correctly at A, causing A to begin sending the next packet. No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, to skip a packet, or to deadlock. The protocol is correct.

This synchronization difficulty is illustrated by Fig. 3-17. In part (a), the normal operation of the protocol is shown. In (b) the peculiarity is illustrated. If B waits for A's first frame before sending one of its own, the sequence is as shown in (a), and every frame is accepted.

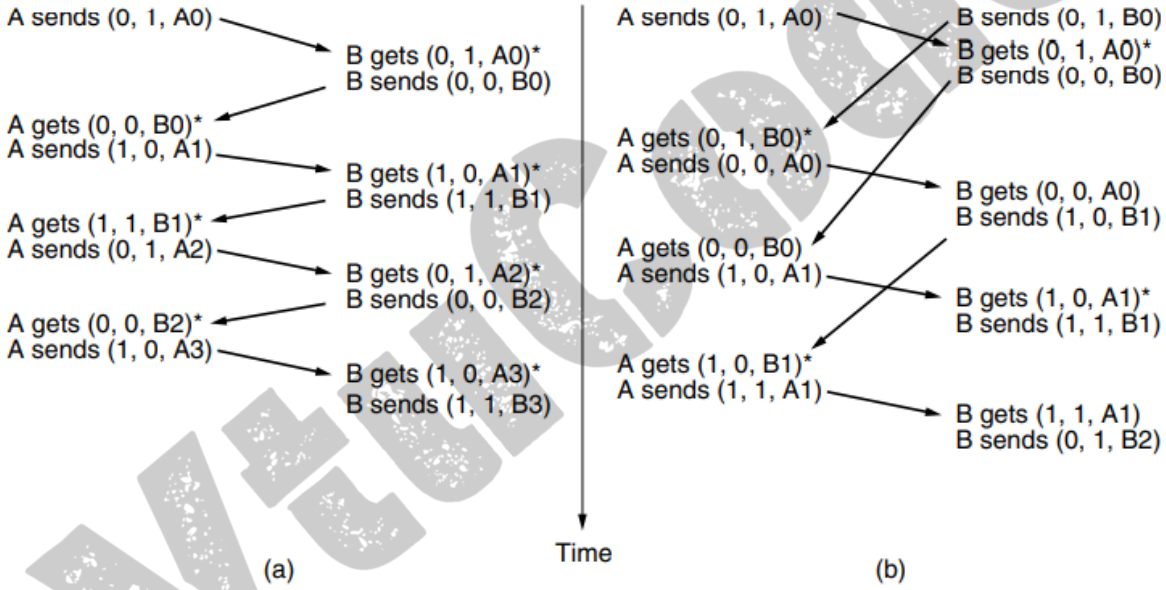


Figure 3-17. Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

3.4.2 A Protocol Using Go-Back-N

For smaller window sizes, the utilization of the link will be less than 100% since the sender will be blocked sometimes. We can write the utilization as the fraction of time that the sender is not blocked:

$$\text{link utilization} \leq \frac{w}{1 + 2BD}$$

This value is an upper bound because it does not allow for any frame processing time and treats the acknowledgement frame as having zero length, since it is usually short. The equation shows the need for having a large window w whenever the bandwidth-delay product is large. If the delay is high, the sender will rapidly exhaust its window even for a moderate bandwidth, as in the satellite example. If the bandwidth is high, even for a moderate delay the sender will exhaust its window quickly unless it has a large window (e.g., a 1-Gbps link with 1-msec delay holds 1 megabit). With stop-and-wait for which w = 1, if there is even one frame's worth of propagation delay the efficiency will be less than 50%. This technique of keeping multiple frames in flight is an example of pipelining. Two basic approaches are available for dealing with errors in the presence of pipelining, both of which are shown in Fig. 3-18.

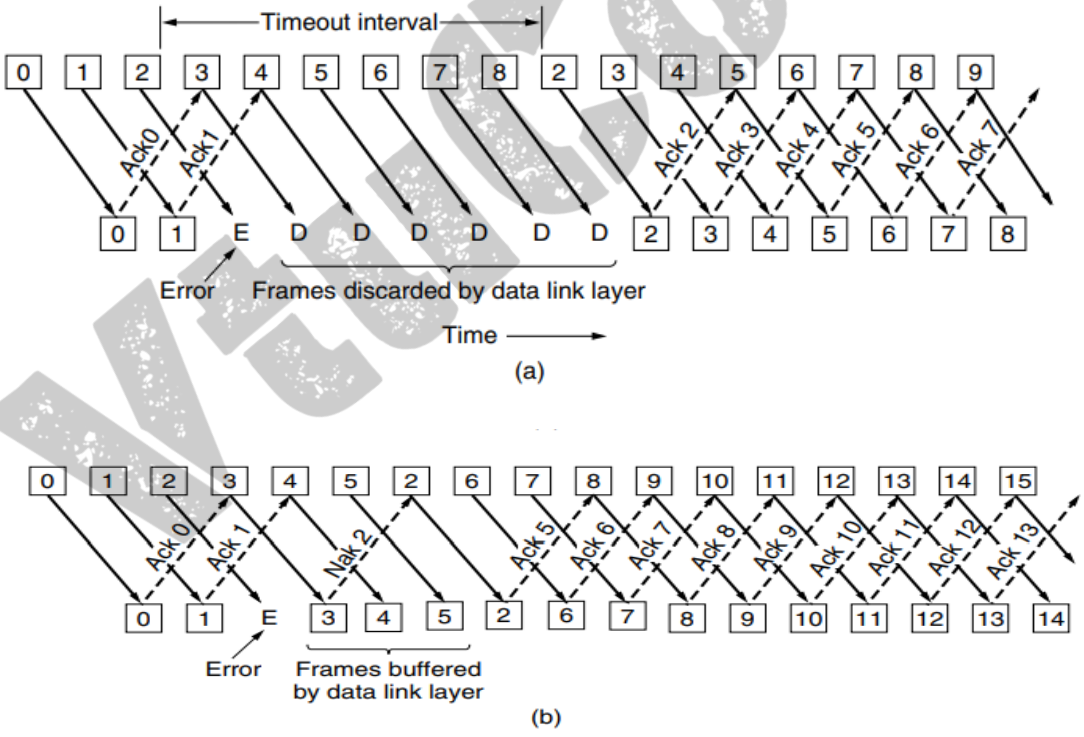


Figure 3-18. Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1 and (b) receiver's window size is large.

One option, called go-back-n, is for the receiver simply to discard all subsequent frames, sending no acknowledgements for the discarded frames. This strategy corresponds to a receive window of

size 1. In other words, the data link layer refuses to accept any frame except the next one it must give to the network layer. If the sender's window fills up before the timer runs out, the pipeline will begin to empty. Eventually, the sender will time out and retransmit all unacknowledged frames in order, starting with the damaged or lost one. This approach can waste a lot of bandwidth if the error rate is high.

In Fig. 3-18(b) we see go-back-n for the case in which the receiver's window is large. Frames 0 and 1 are correctly received and acknowledged. Frame 2, however, is damaged or lost. The sender, unaware of this problem, continues to send frames until the timer for frame 2 expires. Then it backs up to frame 2 and starts over with it, sending 2, 3, 4, etc. all over again

The other general strategy for handling errors when frames are pipelined is called selective repeat. When it is used, a bad frame that is received is discarded, but any good frames received after it are accepted and buffered. When the sender times out, only the oldest unacknowledged frame is retransmitted. If that frame arrives correctly, the receiver can deliver to the network layer, in sequence, all the frames it has buffered. Selective repeat corresponds to a receiver window larger than 1. This approach can require large amounts of data link layer memory if the window is large. Selective repeat is often combined with having the receiver send a negative acknowledgement (NAK) when it detects an error, for example, when it receives a checksum error or a frame out of sequence. NAKs stimulate retransmission before the corresponding timer expires and thus improve performance. In Fig. 3-18(b), frames 0 and 1 are again correctly received and acknowledged and frame 2 is lost. When frame 3 arrives at the receiver, the data link layer there notices that it has missed a frame, so it sends back a NAK for 2 but buffers 3. When frames 4 and 5 arrive, they, too, are buffered by the data link layer instead of being passed to the network layer. Eventually, the NAK 2 gets back to the sender, which immediately resends frame 2. When that arrives, the data link layer now has 2, 3, 4, and 5 and can pass all of them to the network layer in the correct order.

```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike in the previous
   protocols, the network layer is not assumed to have a new packet all the time. Instead,
   the network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

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

```
static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data frame. */
  frame s;                                  /* scratch variable */

  s.info = buffer[frame_nr];                /* insert packet into frame */
  s.seq = frame_nr;                         /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
  to_physical_layer(&s);                    /* transmit the frame */
  start_timer(frame_nr);                    /* start the timer running */
}

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

 enable_network_layer();                    /* allow network_layer_ready events */
 ack_expected = 0;                          /* next ack expected inbound */
 next_frame_to_send = 0;                    /* next frame going out */
 frame_expected = 0;                        /* number of frame expected inbound */
 nbuffered = 0;                             /* initially no packets are buffered */

 while (true) {
   wait_for_event(&event);                  /* four possibilities: see event_type above */



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

     case frame_arrival:                    /* a data or control frame has arrived */
          from_physical_layer(&r);          /* get incoming frame from physical layer */

          if (r.seq == frame_expected) {
               /* Frames are accepted only in order. */
               to_network_layer(&r.info);   /* pass packet to network layer */
               inc(frame_expected);         /* advance lower edge of receiver's window */
           }
```

```
            /* Ack n implies n – 1, n – 2, etc.  Check for this. */
            while (between(ack_expected, r.ack, next_frame_to_send)) {
                /* Handle piggybacked ack. */
                nbuffered = nbuffered – 1;          /* one frame fewer buffered */
                stop_timer(ack_expected);           /* frame arrived intact; stop timer */
                inc(ack_expected);                  /* contract sender's window */
            }
            break;

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

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

      }


    }

    if (nbuffered < MAX_SEQ)
            enable_network_layer();
    else
            disable_network_layer();
  }
}
```

Figure 3-19. A sliding window protocol using go-back-n.

Because protocol has multiple outstanding frames, it logically needs multiple timers, one per outstanding frame. Each frame times out independently of all the other ones. However, all of these timers can easily be simulated in software using a single hardware clock that causes interrupts periodically. The pending timeouts form a linked list, with each node of the list containing the number of clock ticks until the timer expires, the frame being timed, and a pointer to the next node. As an illustration of how the timers could be implemented, consider the example of Fig. 3-20(a). Assume that the clock ticks once every 1 msec. Initially, the real time is 10:00:00.000; three timeouts are pending, at 10:00:00.005, 10:00:00.013, and 10:00:00.019. Every time the hardware clock ticks, the real time is updated and the tick counter at the head of the list is decremented. When the tick counter becomes zero, a timeout is caused and the node is removed from the list, as shown in Fig. 3-20(b). Although this organization requires the list to be scanned when start timer or stop timer is called, it does not require much work per tick.

3.4.3 A Protocol Using Selective Repeat

The go-back-n protocol works well if errors are rare, but if the line is poor it wastes a lot of bandwidth on retransmitted frames. An alternative strategy, the selective repeat protocol, is to allow the receiver to accept and buffer the frames following a damaged or lost one. Whenever a frame arrives, its sequence number is checked by the function between to see if it falls within the window. If so and if it has not already been received, it is accepted and stored. This action is taken without regard to whether or not the frame contains the next packet expected by the network layer. Of course, it must be kept within the data link layer and not passed to the network layer until all the lower-numbered frames have already been delivered to the network layer in the correct order. A protocol using this algorithm is given in Fig. 3-21.

```
/* Protocol 6 (Selective repeat) accepts frames out of order but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7                                   /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                              /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1;                  /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol 5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                          /* scratch variable */
```

```
  s.kind = fk;                                  /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                             /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                /* one nak per frame, please */
  to_physical_layer(&s);                        /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                             /* no need for separate ack frame */
}

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


enable_network_layer();                         /* initialize */
ack_expected = 0;                               /* next ack expected on the inbound stream */
next_frame_to_send = 0;                         /* number of next outgoing frame */
frame_expected = 0;
too_far = NR_BUFS;
nbuffered = 0;                                  /* initially no packets are buffered */
for (i = 0; i < NR_BUFS; i++) arrived[i] = false;


 while (true) {
   wait_for_event(&event);                      /* five possibilities: see event_type above */
   switch(event) {
     case network_layer_ready:                  /* accept, save, and transmit a new frame */
          nbuffered = nbuffered + 1;            /* expand the window */
          from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
          send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
          inc(next_frame_to_send);              /* advance upper window edge */
          break;
```

```
case frame_arrival:                          /* a data or control frame has arrived */
     from_physical_layer(&r);                /* fetch incoming frame from physical layer */
     if (r.kind == data) {
          /* An undamaged frame has arrived. */
          if ((r.seq != frame_expected) && no_nak)
             send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
          if (between(frame_expected,r.seq,too_far) && (arrived[r.seq%NR_BUFS]==false)) {
               /* Frames may be accepted in any order. */
               arrived[r.seq % NR_BUFS] = true;      /* mark buffer as full */
               in_buf[r.seq % NR_BUFS] = r.info;     /* insert data into buffer */


                  while (arrived[frame_expected % NR_BUFS]) {
                       /* Pass frames and advance window. */
                       to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                       no_nak = true;
                       arrived[frame_expected % NR_BUFS] = false;
                       inc(frame_expected);     /* advance lower edge of receiver's window */
                       inc(too_far);            /* advance upper edge of receiver's window */
                       start_ack_timer();       /* to see if a separate ack is needed */
                  }
               }
          }
          if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
               send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

          while (between(ack_expected, r.ack, next_frame_to_send)) {
               nbuffered = nbuffered – 1;           /* handle piggybacked ack */
               stop_timer(ack_expected % NR_BUFS);     /* frame arrived intact */
               inc(ack_expected);                   /* advance lower edge of sender's window */
          }
          break;

          case cksum_err:
               if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* damaged frame */
               break;
          case timeout:
               send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
               break;
          case ack_timeout:
               send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
          }
          if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
     }
}
```

Figure 3-21. A sliding window protocol using selective repeat.

Suppose that we have a 3-bit sequence number, so that the sender is permitted to transmit up to seven frames before being required to wait for an acknowledgement. Initially, the sender's and receiver's windows are as shown in Fig. 3-22(a). The sender now transmits frames 0 through 6. The receiver's window allows it to accept any frame with a sequence number between 0 and 6 inclusive. All seven frames arrive correctly, so the receiver acknowledges them and advances its window to allow receipt of 7, 0, 1, 2, 3, 4, or 5, as shown in Fig. 3-22(b). All seven buffers are marked empty.

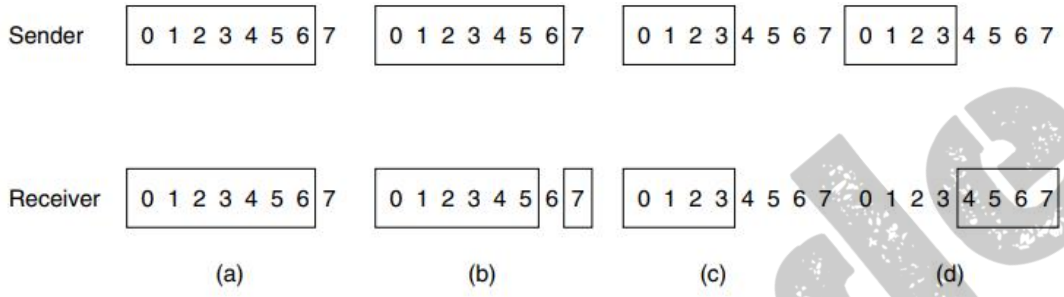


Figure 3-22. (a) Initial situation with a window of size7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

To ensure that there is no overlap, the maximum window size should be at most half the range of the sequence numbers. This situation is shown in Fig. 3-22(c) and Fig. 3-22(d). With 3 bits, the sequence numbers range from 0 to 7. Only four unacknowledged frames should be outstanding at any instant. That way, if the receiver has just accepted frames 0 through 3 and advanced its window to permit acceptance of frames 4 through 7, it can unambiguously tell if subsequent frames are retransmissions (0 through 3) or new ones (4 through 7). In general, the window size for protocol 6 will be (MAX SEQ + 1)/2.

MEDIUM ACCESS CONTROL SUBLAYER

Network links can be divided into two categories: those using point-to-point connections and those using broadcast channels. The MAC sub layer is the bottom part of the data link layer, so logically we should have studied it examining all the point-to-point protocols. it is easier to understand protocols involving multiple parties after two-party protocols.

4.1 THE CHANNEL ALLOCATION PROBLEM

The central theme of this chapter is how to allocate a single broadcast channel among competing users. The channel might be a portion of the wireless spectrum in a geographic region, or a single

wire or optical fiber to which multiple nodes are connected. It does not matter. In both cases, the channel connects each user to all other users and any user who makes full use of the channel interferes with other users who also wish to use the channel.

4.1.1 Static Channel Allocation

The traditional way of allocating a single channel, such as a telephone trunk, among multiple competing users is to chop up its capacity by using one of the multiplexing schemes. If there are N users, the bandwidth is divided into N equal-sized portions, with each user being assigned one portion. Since each user has a private frequency band, there is now no interference among users. When there is only a small and constant number of users, each of which has a steady stream or a heavy load of traffic, this division is a simple and efficient allocation mechanism.

A wireless example is FM radio stations. Each station gets a portion of the FM band and uses it most of the time to broadcast its signal. The poor performance of static FDM can easily be seen with a simple queueing theory calculation. Let us start by finding the mean time delay, T, to send a frame onto a channel of capacity C bps. We assume that the frames arrive randomly with an average arrival rate of λ frames/sec, and that the frames vary in length with an average length of 1/μ bits. With these parameters, the service rate of the channel is μC frames/sec.

A standard queueing theory result is

$$T = \frac{1}{\mu C - \lambda}$$

(For the curious, this result is for an __M/M/1‘‘ queue. It requires that the randomness of the times between frame arrivals and the frame lengths follow an exponential distribution, or equivalently be the result of a Poisson process.) Precisely the same arguments that apply to FDM also apply to other ways of statically dividing the channel. If we were to use time division multiplexing (TDM) and allocate each user every Nth time slot, if a user does not use the allocated.

4.1.2 Assumptions for Dynamic Channel Allocation

1. **Independent Traffic**. The model consists ofNindependent stations (e.g., computers, telephones), each with a program or user that generates frames for transmission. The expected number of frames generated in an interval of length t is λΔt, where λ is a constant (the arrival rate of new frames). Once a frame has been generated, the station is blocked and does nothing until the frame has been successfully transmitted.
2. **Single Channel**. A single channel is available for all communication. All stations can transmit on it and all can receive from it. The stations are assumed to be equally capable, though protocols may assign them different roles (e.g., priorities).

3. **Observable Collisions**. If two frames are transmitted simultaneously, they overlap in time and the resulting signal is garbled. This event is called a collision. All stations can detect that a collision has occurred. A collided frame must be transmitted again later. No errors other than those generated by collisions occur.
4. **Continuous or Slotted Time**. Time may be assumed continuous, inwhich case frame transmission can begin at any instant. Alternatively, time may be slotted or divided into discrete intervals (called slots). Frame transmissions must then begin at the start of a slot. A slot may contain 0, 1, or more frames, corresponding to an idle slot, a successful transmission, or a collision, respectively.
5. **Carrier Sense or No Carrier Sense**. With the carrier sense assumption, stations can tell if the channel is in use before trying to use it. No station will attempt to use the channel while it is sensed as busy. If there is no carrier sense, stations cannot sense the channel before trying to use it. They just go ahead and transmit. Only later can they determine whether the transmission was successful.

4.2 MULTIPLE ACCESS PROTOCOLS

Many algorithms for allocating a multiple access channel are known. In the following sections, we will study a small sample of the more interesting ones and give some examples of how they are commonly used in practice.

4.2.1 ALOHA

- In this case, "pristine" can be interpreted as "not having a working telephone system." This did not make life more pleasant for researcher Norman Abramson and his colleagues at the University of Hawaii who were trying to connect users on re-mote islands to the main computer in Honolulu. Stringing their own cables under the Pacific Ocean was not in the cards, so they looked for a different solution.
- The one they found used short-range radios, with each user terminal sharing the same upstream frequency to send frames to the central computer.
- It included a simple and elegant method to solve the channel allocation problem. Their work has been extended by many researchers since then (Schwartz and Abramson, 2009). Although Abramson's work, called the ALOHA system, used ground-based radio broadcasting, the basic idea is applicable to any system in which uncoordinated users are competing for the use of a single shared channel.

Pure ALOHA

- The basic idea of an ALOHA system is simple: let users transmit whenever they have data to be sent. There will be collisions, of course, and the colliding frames will be damaged. Senders need some way to find out if this is the case
- In the ALOHA system, after each station has sent its frame to the central computer, this computer rebroadcasts the frame to all of the stations. A sending station can thus listen for the broadcast from the hub to see if its frame has gotten through. In other systems, such as wired
- LANs, the sender might be able to listen for collisions while transmitting.
- If the frame was destroyed, the sender just waits a random amount of time and sends it again. The waiting time must be random or the same frames will collide over and over, in lockstep. Systems in which multiple users share a common channel in a way that can lead to conflicts are known as contention systems.

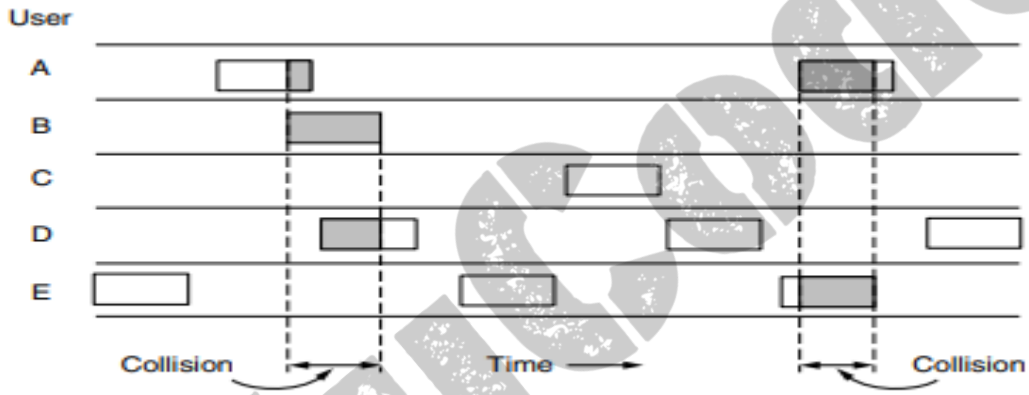


Figure 4-1. In pure ALOHA, frames are transmitted at completely arbitrary times.

A user is always in one of two states: typing or waiting. Initially, all users are in the typing state. When a line is finished, the user stops typing, waiting for a response. . The station then transmits a frame containing the line over the shared channel to the central computer and checks the channel to see if it was successful. If so, the user sees the reply and goes back to typing. If not, the user continues

If any other user has generated a frame between time t 0 and t 0 + t, the end of that frame will collide with the beginning of the shaded one. In fact, the shaded frame's fate was already sealed even before the first bit was sent, but since in pure ALOHA a station does not listen to the channel before transmitting, it has no way of knowing that another frame was already underway. Similarly, any other frame started between t 0 + t and t 0 + 2t will bump into the end of the shaded frame

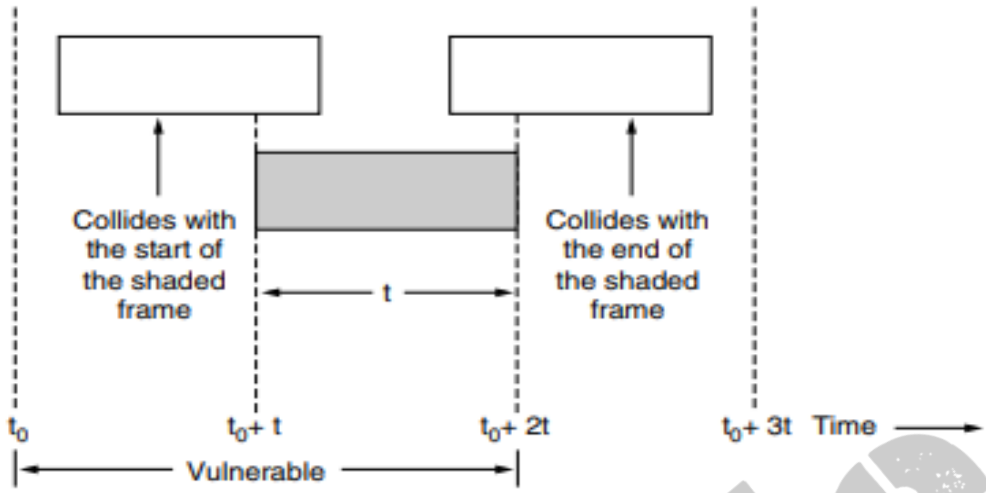


Figure 4-2. Vulnerable period for the shaded frame.

The probability that k frames are generated during a given frame time, in which G frames are expected, is given by the Poisson distribution Pr[k] = k! Gk e −G (4-2) so the probability of zero frames is just e −G. In an interval two frame times long, the mean number of frames generated is 2G. The probability of no frames being initiated during the entire vulnerable period is thus given by P0 = e −2G. Using S = GP0, we get S = Ge −2G The relation between the offered traffic and the throughput is shown in Fig. 4-3. The maximum throughput occurs at G = 0.5, with S = 1/2e, which is about 0.184. In other words, the best we can hope for is a channel utilization of 18%. This result is not very encouraging, but with everyone transmitting at will, we could hardly have expected a 100% success rate.

Slotted ALOHA

Soon after ALOHA came onto the scene, Roberts (1972) published a method for doubling the capacity of an ALOHA system. His proposal was to divide time into discrete intervals called slots, each interval corresponding to one frame.

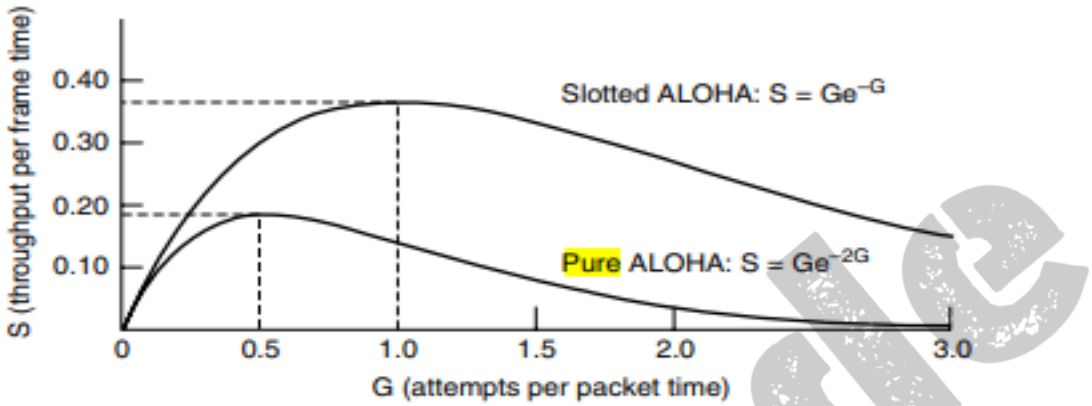
In Roberts‘ method, which has come to be known as slotted ALOHA—in contrast to Abramson‘s pure ALOHA—a station is not permitted to send when-ever the user types a line. Instead, it is required to wait for the beginning of the next slot. Thus, the continuous time ALOHA is turned into a discrete time one. This halves the vulnerable period.

The probability that it will avoid a collision is e −G , which is the probability that all the other stations are silent in that slot. The probability of a collision is then just 1 −e −G . The probability of a transmission requiring exactly k attempts (i.e., k− 1 collisions followed by one success) is

$$P_k = e^{-G}(1 - e^{-G})^{k-1}$$

The expected number of transmissions, E, per line typed at a terminal.

$$E = \sum_{k=1}^{\infty} kP_k = \sum_{k=1}^{\infty} ke^{-G}(1 - e^{-G})^{k-1} = e^{G}$$



4.2.2 CARRIER SENSE MULTIPLE ACCESS PROTOCOLS

Protocols in which stations listen for a carrier (i.e., a transmission) and act accordingly are called carrier sense protocols. A number of them have been proposed, and they were long ago analyzed in detail. These networks can achieve a much better utilization than 1/ e. In this section, we will discuss some protocols for improving performance

Persistent and Non persistent CSMA

- The first carrier sense protocol that we will study here is called 1-persistentCSMA (Carrier Sense Multiple Access). That is a bit of a mouthful for the simplest CSMA scheme. When a station has data to send, it first listens to the channel to see if anyone else is transmitting at that moment. If the channel is idle, the stations sends its data. Otherwise, if the channel is busy, the station just waits until it becomes idle. Then the station transmits a frame
- If a collision occurs, the station waits a random amount of time and starts all over again. The protocol is called 1-persistent because the station transmits with a probability of 1 when it finds the channel idle.
- This chance depends on the number of frames that fit on the channel, or the bandwidth-delay product of the channel. If only a tiny fraction of a frame fits on the channel, which is the case in most LANs since the propagation delay is small, the chance of a collision happening is small. The larger the bandwidth-delay product, the more important this effect becomes, and the worse the performance of the protocol.

- Even so, this protocol has better performance than pure ALOHA because both stations have the decency to desist from interfering with the third station‘s frame. Exactly the same holds for slotted ALOHA.
- A second carrier sense protocol is non persistent CSMA. In this protocol, a conscious attempt is made to be less greedy than in the previous one. As before, a station senses the channel when it wants to send a frame, and if no one else is sending, the station begins doing so itself. However, if the channel is already in use, the station does not continually sense it for the purpose of seizing it immediately upon detecting the end of the previous transmission.

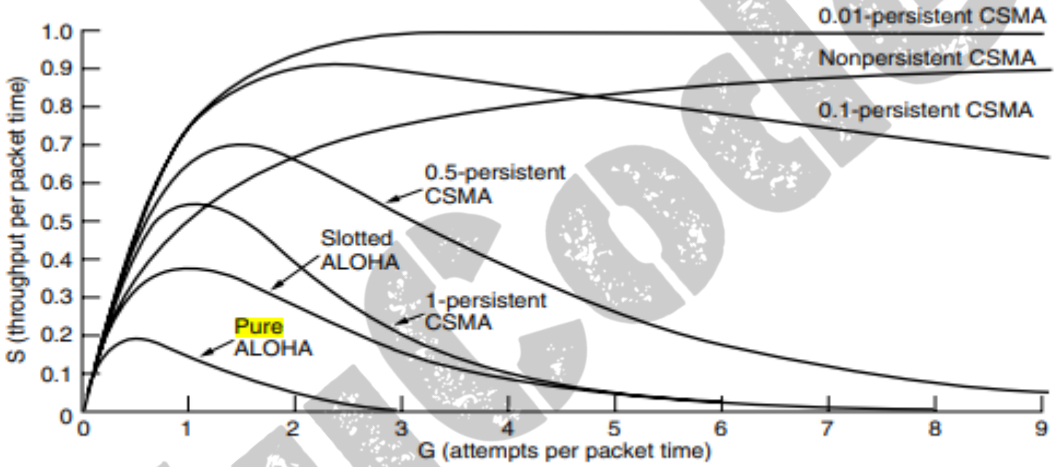


Figure 4-4. Comparison of the channel utilization versus load for various random access protocols.

CSMA with Collision Detection

- Persistent and non persistent CSMA protocols are definitely an improvement over ALOHA because they ensure that no station begins to transmit while the channel is busy. However, if two stations sense the channel to be idle and begin transmitting simultaneously, their signals will still collide.

- Another improvement is for the stations to quickly detect the collision and abruptly stop transmitting, (rather than finishing them) since they are irretrievably garbled anyway. This strategy saves time and bandwidth.
- This protocol, known as CSMA/CD (CSMA with Collision Detection), is the basis of the classic Ethernet LAN, so it is worth devoting some time to looking at it in detail. It is important to realize that collision detection is an analog process.
- The station‘s hardware must listen to the channel while it is transmitting. If the signal it reads back is different from the signal it is putting out, it knows that a collision is occurring.

CSMA/CD, as well as many other LAN protocols, uses the conceptual model of Fig. 4-5. At the point marked t 0, a station has finished transmitting its frame. Any other station having a frame to send may now attempt to do so. If two or more stations decide to transmit simultaneously, there will be a collision. If a station detects a collision, it aborts its transmission, waits a random period of time, and then tries again Therefore, our model for CSMA/CD will consist of alternating contention and transmission periods, with idle periods occurring when all stations are quiet (e.g., for lack of work)

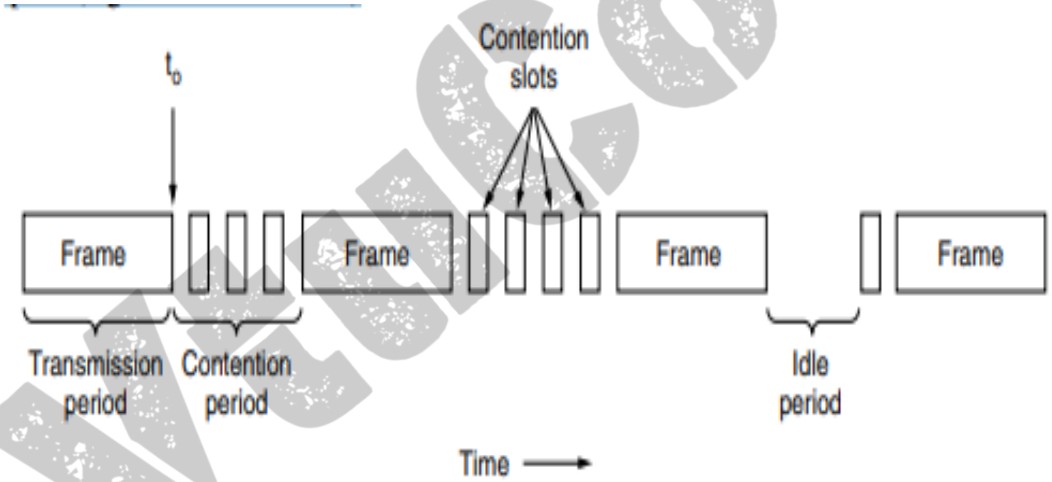


Figure 4-5. CSMA/CD can be in contention, transmission, or idle state.

4.2.3 COLLISION-FREE PROTOCOLS

In the protocols to be described, we assume that there are exactly N stations, each programmed with a unique address from 0 to N− 1. It does not matter that some stations may be inactive part of the time. We also assume that propagation delay is negligible.

Most of these protocols are not currently used in major systems, but in a rapidly changing field, having some protocols with excellent properties available for future systems is often a good thing

A Bit-Map Protocol

- In our first collision-free protocol, the basic bit-map method, each contention period consists of exactly N slots. If station 0 has a frame to send, it transmits a 1 bit during the slot 0.
- No other station is allowed to transmit during this slot. Regardless of what station 0 does, station 1 gets the opportunity to transmit a 1 bit during slot 1, but only if it has a frame queued. Protocols like this in which the desire to transmit is broadcast before the actual transmission are called reservation protocols because they reserve channel ownership in advance and prevent collisions.
- Let us briefly analyze the performance of this protocol. For convenience, we will measure time in units of the contention bit slot, with data frames consisting of d time units.
- The channel efficiency at low load is easy to compute. The overhead per frame is N bits and the amount of data is d bits, for an efficiency of d / (d+N).

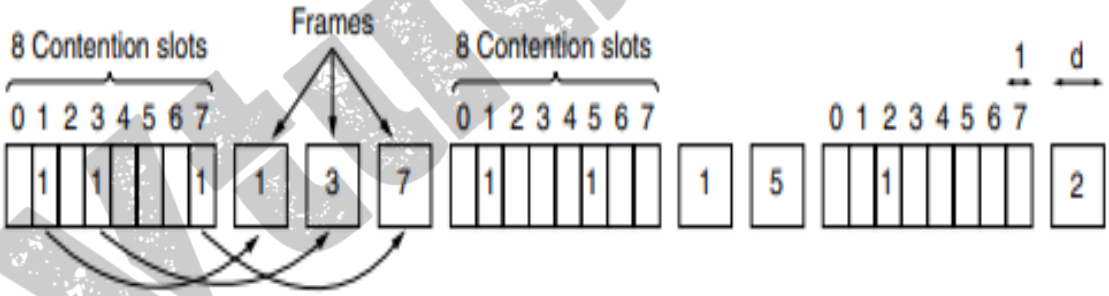


Figure 4-6. The basic bit-map protocol.

Token Passing

- The essence of the bit-map protocol is that it lets every station transmit a frame in turn in a predefined order. Another way to accomplish the same thing is to pass a small message called a token from one station to the next in the same predefined order.
- The token represents permission to send. If a station has a frame queued for transmission when it receives the token, it can send that frame before it passes the token to the next station. If it has no queued frame, it simply passes the token.
- In a token ring protocol, the topology of the network is used to define the order in which stations send. The stations are connected one to the next in a single ring.

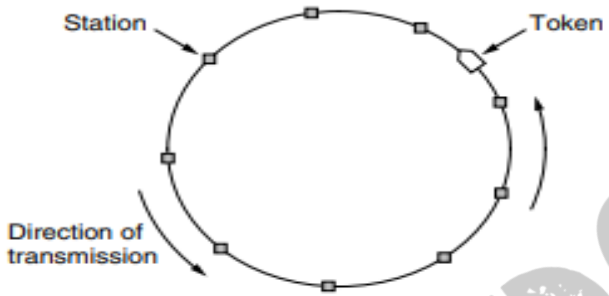


Figure 4-7. Token ring.

The performance of token passing is similar to that of the bit-map protocol, though the contention slots and frames of one cycle are now intermingled.

The channel connecting the stations might instead be a single long bus. Each station then uses the bus to send the token to the next station in the predefined sequence. Possession of the token allows a station to use the bus to send one frame, as before. This protocol is called token bus.

Binary Countdown

- A problem with the basic bit-map protocol, and by extension token passing, is that the overhead is 1 bit per station, so it does not scale well to networks with thousands of stations. We can do better than that by using binary station ad-dresses with a channel that combines transmissions.
- All addresses are assumed to be the same length. The bits in each ad-dress position from different stations are BOOLEAN OR together by the channel when they are sent at the same time. We will call this protocol binary count-down
- It implicitly assumes that the trans-mission delays are negligible so that all stations see asserted bits essentially instantaneously.
- To avoid conflicts, an arbitration rule must be applied: as soon as a station sees that a high-order bit position that is 0 in its address has been overwritten with a 1, it gives up.

The protocol is illustrated in Fig. 4-8. It has the property that higher-numbered stations have a higher priority than lower-numbered stations, which may be either good or bad, depending on the context.

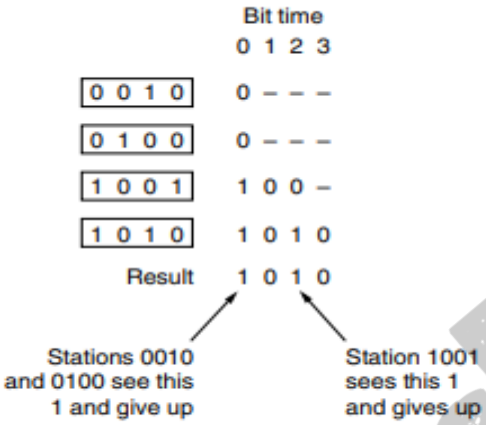


Figure 4-8. The binary countdown protocol. A dash indicates silence.

4.2.4 LIMITED-CONTENTION PROTOCOLS

- Each strategy can be rated as to how well it does with respect to the two important performance measures, delay at low load and channel efficiency at high load.
- Under conditions of light load, contention (i.e., pure or slotted ALOHA) is preferable due to its low delay (since collisions are rare). As the load increases, contention becomes increasingly less attractive because the overhead associated with channel arbitration becomes greater
- Such protocols, which we will call limited-contention protocols, do in fact exist, and will conclude our study of carrier sense networks.

Before looking at the asymmetric protocols, let us quickly review the performance of the symmetric case. Suppose that k stations are contending for channel access. Each has a probability p of transmitting during each slot. The probability that some station successfully acquires the channel during a given slot is the probability that any one station transmits, with probability p, and all other $k-1$ stations defer, each with probability $1-p$. This value is $kp(1-p)^{k-1}$. To find the optimal value of p, we differentiate with respect to p, set the result to zero, and solve for p. Doing so, we find that the best value of p is $1/k$. Substituting $p = 1/k$, we get

$$\Pr[\text{success with optimal } p] = \left[\frac{k-1}{k}\right]^{k-1} \quad (4\text{-}4)$$

This probability is plotted in Fig. 4-9. For small numbers of stations, the chances of success are good, but as soon as the number of stations reaches even five, the probability has dropped close to its asymptotic value of $1/e$.

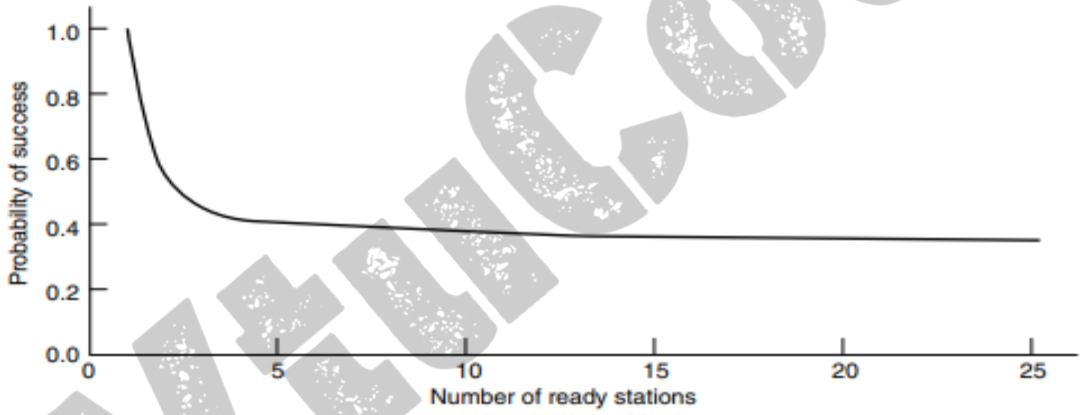


Figure 4-9. Acquisition probability for a symmetric contention channel.

The Adaptive Tree Walk Protocol

This mixed sample was then tested for antibodies. If none were found, all the soldiers in the group were declared healthy. If antibodies were present, two new mixed samples were prepared, one from soldiers 1 through N / 2 and one from the rest. The proc-ess was repeated recursively until the infected soldiers were determined

.If, on the other hand, two or more stations under node 2 want to transmit, there will be a collision during slot 1, in which case it is node 4‘s turn during slot 2.

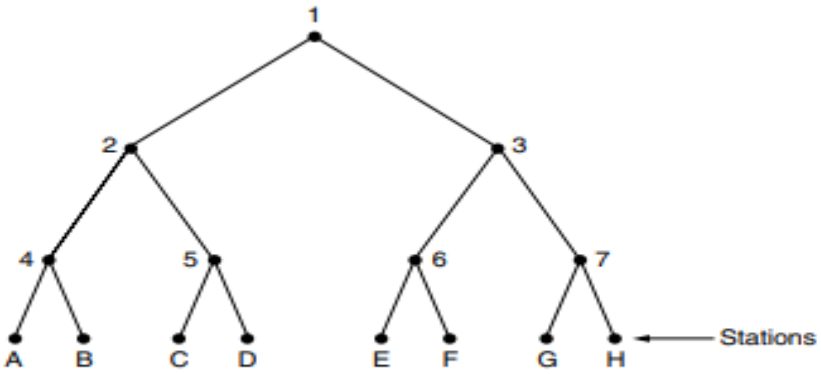


Figure 4-10. The tree for eight stations.

In essence, if a collision occurs during slot 0, the entire tree is searched, depth first, to locate all ready stations. Each bit slot is associated with some particular node in the tree. If a collision occurs, the search continues recursively with the node‘s left and right children. If a bit slot is idle or if only one station transmits in it, the searching of its node can stop because all ready stations have been located. (Were there more than one, there would have been a collision.) When the load on the system is heavy, it is hardly worth the effort to dedicate slot 0 to node 1 because that makes sense only in the unlikely event that precisely one station has a frame to send.

To proceed, let us number the levels of the tree from the top, with node 1 in Fig. 4-10 at level 0, nodes 2 and 3 at level 1, etc. Notice that each node at level I has a fraction 2−i of the stations below it. If the q ready stations are uniformly distributed, the expected number of them below a specific node at level i is just 2−i q. Intuitively, we would expect the optimal level to begin searching the tree to be the one at which the mean number of contending stations per slot is 1, that is, the level at which 2−i q = 1. Solving this equation, we find that i = log2 q.

4.2.5 Wireless LAN Protocols

- Such a LAN is an example of a broadcast channel. It also has somewhat different properties than a wired LAN, which leads to different MAC protocols. In this section, we will examine some of these protocols. In Sec. 4.4, we will look at 802.11 (WiFi) in detail.
- A common configuration for a wireless LAN is an office building with access points (APs) strategically placed around the building. The APs are wired together using copper or fiber and provide connectivity to the stations that talk to them
- A naive approach to using a wireless LAN might be to try CSMA: just listen for other transmissions and only transmit if no one else is doing so. The trouble is, this protocol is not really a good way to think about wireless because what matters for reception is interference at the receiver, not at the sender.

- To see the nature of the problem, consider where four wireless stations are illustrated. For our purposes, it does not matter which are APs and which are laptops. The radio range is such that A and B are within each other's range and can potentially interfere with one another. C can also potentially interfere with both B and D, but not with A

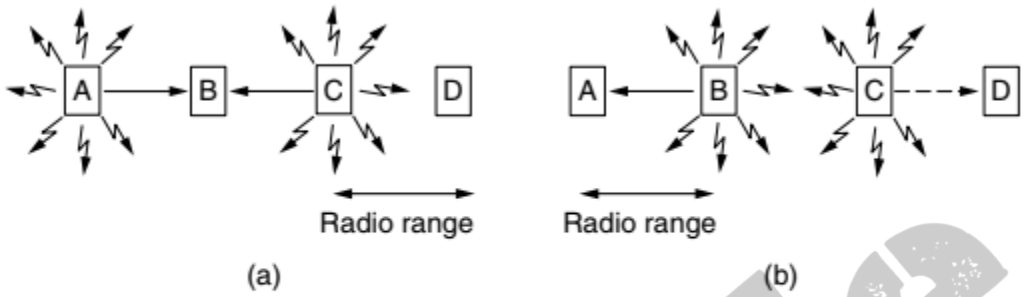


Figure 4-11. A wireless LAN. (a) *A* and *C* are hidden terminals when transmitting to *B*. (b) *B* and *C* are exposed terminals when transmitting to *A* and *D*.

- First consider what happens when A and C transmit to B, as depicted in Fig. 4-11(a). If A sends and then C immediately senses the medium, it will not hear A becauseA is out of range. Thus C will falsely conclude that it can transmit to B. If C does start transmitting, it will interfere at B, wiping out the frame from A. (We assume here that no CDMA-type scheme is used to provide multiple channels, so collisions garble the signal and destroy both frames.) We want a MAC protocol that will prevent this kind of collision from happening because it wastes bandwidth.
- The problem of a station not being able to detect a potential competitor for the medium because the competitor is too far away is called the hidden terminal problem.
- Now let us look at a different situation: B transmitting to A at the same time that C wants to transmit to D, as shown in Fig. 4-11(b). If C senses the medium, it will hear a transmission and falsely conclude that it may not send to D (shown as a dashed line). In fact, such a transmission would cause bad reception only in the zone between B and C, where neither of the intended receivers is located. We want a MAC protocol that prevents this kind of deferral from happening because it wastes bandwidth. The problem is called the exposed terminal problem
- The difficulty is that, before starting a transmission, a station really wants to know whether there is radio activity around the receiver.
- An early and influential protocol that tackles these problems for wireless LANs is MACA (Multiple Access with Collision Avoidance) (Karn, 1990). The basic idea behind it is for the sender to stimulate the receiver into outputting a short frame, so stations nearby can

detect this transmission and avoid transmitting for the duration of the upcoming (large) data frame. This technique is used instead of carrier sense.

MACA is illustrated in Fig. 4-12. Let us see how A sends a frame to B. A starts by sending an RTS (Request To Send) frame to B, as shown in Fig. 4-12(a). This short frame (30 bytes) contains the length of the data frame that will eventually follow. Then B replies with a CTS (Clear To Send) frame, as shown in Fig. 4-12(b). The CTS frame contains the data length (copied from the RTS frame). Upon receipt of the CTS frame, A begins transmission

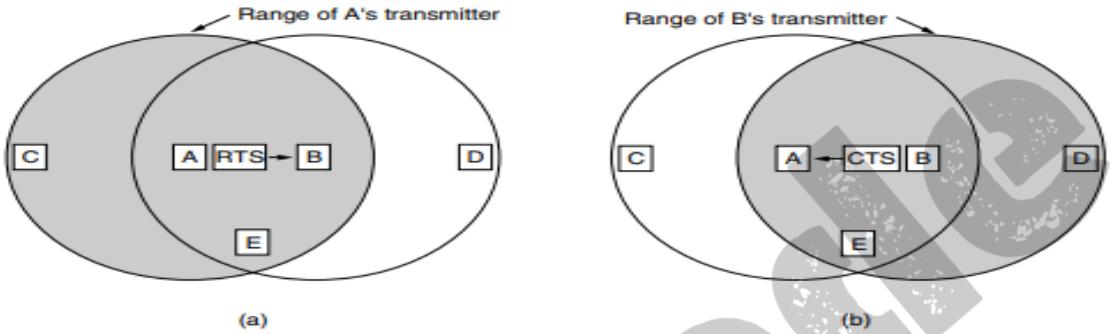


Figure 4-12. The MACA protocol. (a) *A* sending an RTS to *B*. (b) *B* responding with a CTS to *A*.