

### 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 dedicated 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.

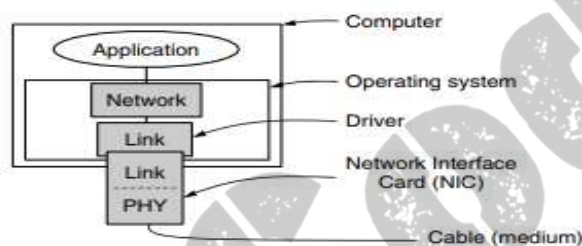


Figure 3-10. Implementation of the physical, data link, and network layers.

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.

### 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);            /* increment 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);
        if (event == frame_arrival) {
            from_physical_layer(&r);
            if (r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
            }
            s.ack = 1 - frame_expected;
            to_physical_layer(&s);
        }
    }
}

```

/\* possibilities: frame\_arrival, cksum\_err \*/  
 /\* a valid frame has arrived \*/  
 /\* go get the newly arrived frame \*/  
 /\* this is what we have been waiting for \*/  
 /\* pass the data to the network layer \*/  
 /\* next time expect the other sequence nr \*/  
 /\* tell which frame is being acked \*/  
 /\* send acknowledgement \*/

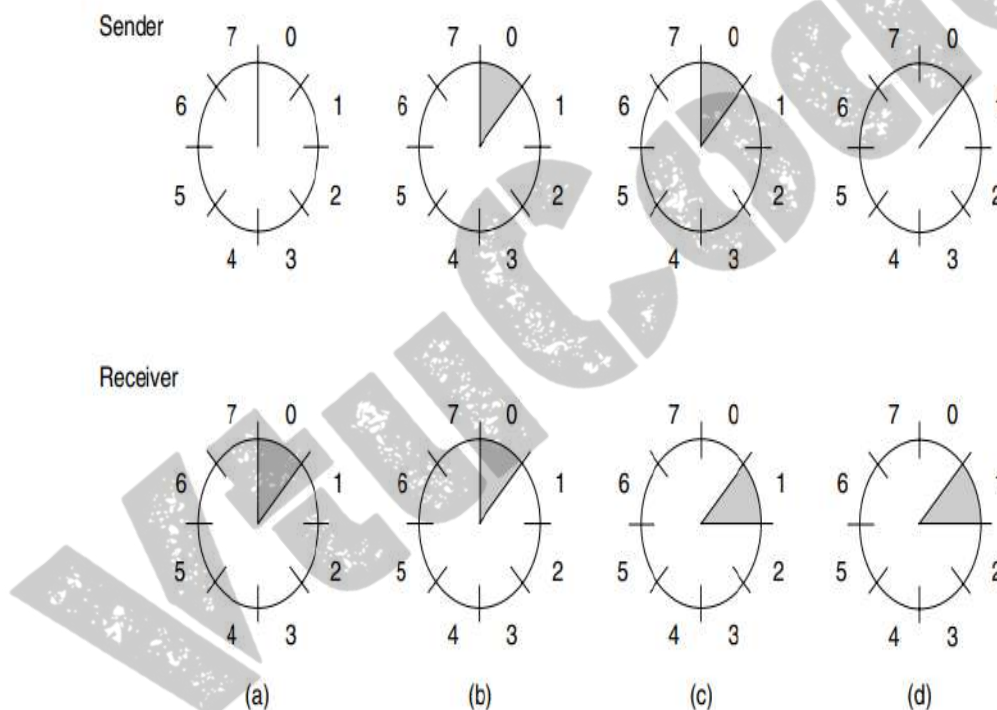
**Figure 3-14.** A positive acknowledgement with retransmission protocol.

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  $2^n - 1$  so the sequence number fits exactly in an  $n$ -bit field. The stop-and-wait sliding window protocol uses  $n = 1$ , restricting the sequence numbers to 0 and 1.

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.

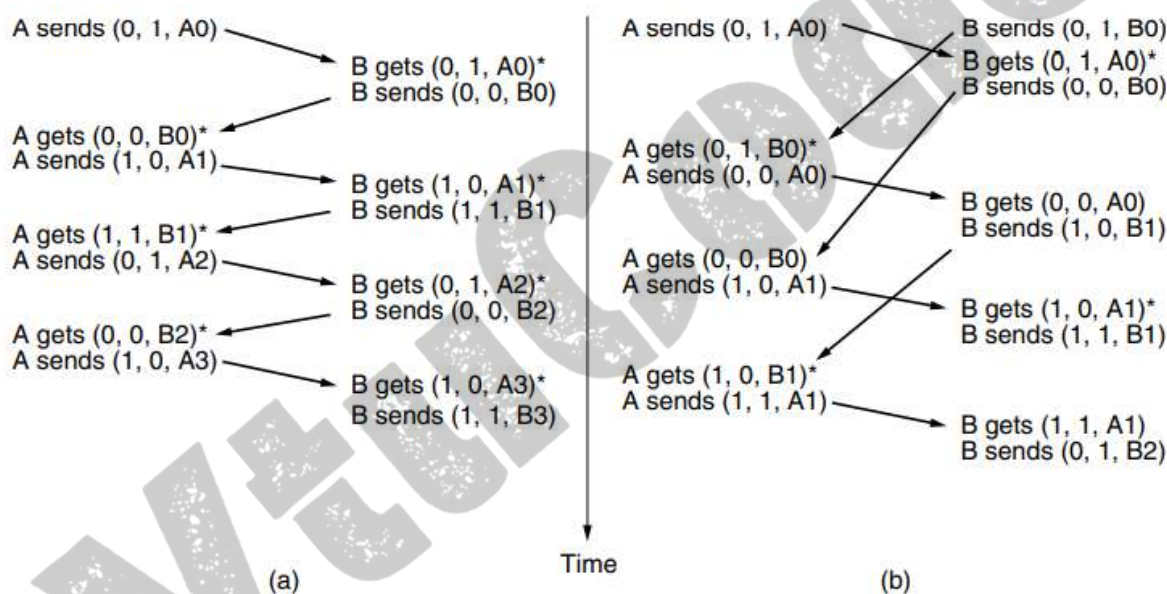


**Figure 3-15.** A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

### 3.4.1 A One-Bit Sliding Window Protocol

Consequently, A may time out repeatedly, sending a series of identical frames, all with  $\text{seq} = 0$  and  $\text{ack} = 1$ . 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  $\text{ack} = 1$  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  $\text{seq} = 0$  and  $\text{ack} = 0$ . 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.



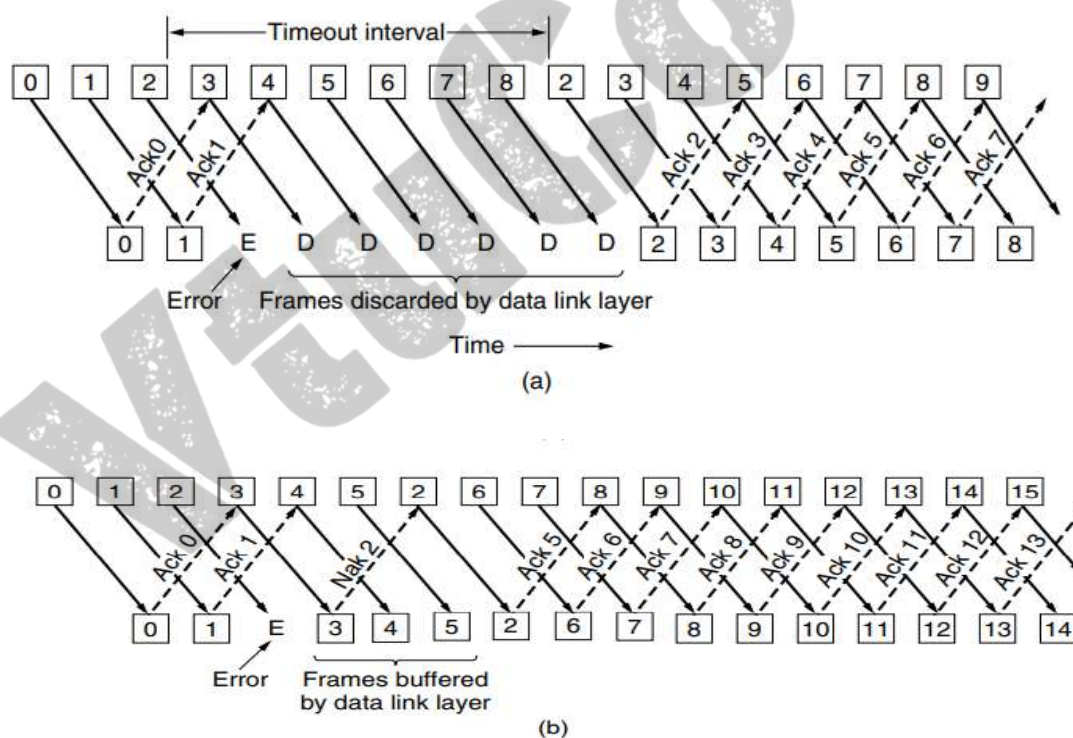
**Figure 3-17.** Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

### 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;

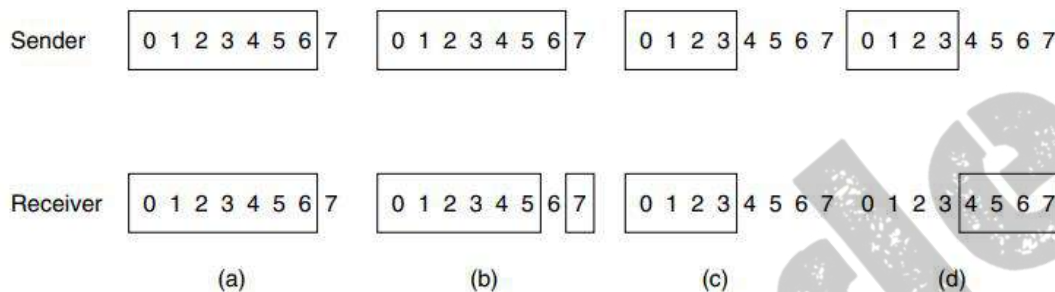
case ack_timeout:
    send_frame(ack, 0, frame_expected, out_buf); /* ack timer expired; send ack */
}
if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
}
}

```

**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 size 7. (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  $(\text{MAX SEQ} + 1)/2$ .

## 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  $\lambda$  frames/sec, and that the frames vary in length with an average length of  $1/\mu$  bits. With these parameters, the service rate of the channel is  $\mu C$  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  $N$ th time slot, if a user does not use the allocated.

#### 4.1.2 Assumptions for Dynamic Channel Allocation

1. **Independent Traffic.** The model consists of  $N$  independent 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  $\lambda \Delta t$ , where  $\lambda$  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, in which 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 remote 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.

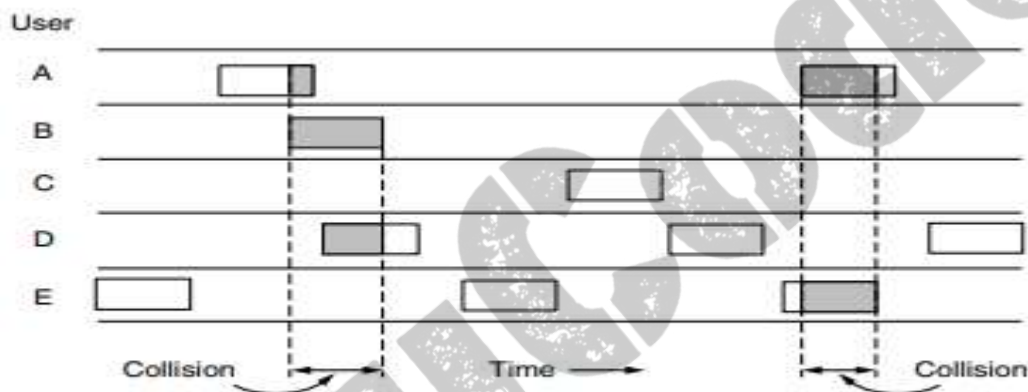


Figure 4-1. In pure ALOHA, frames are transmitted at completely arbitrary times.

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

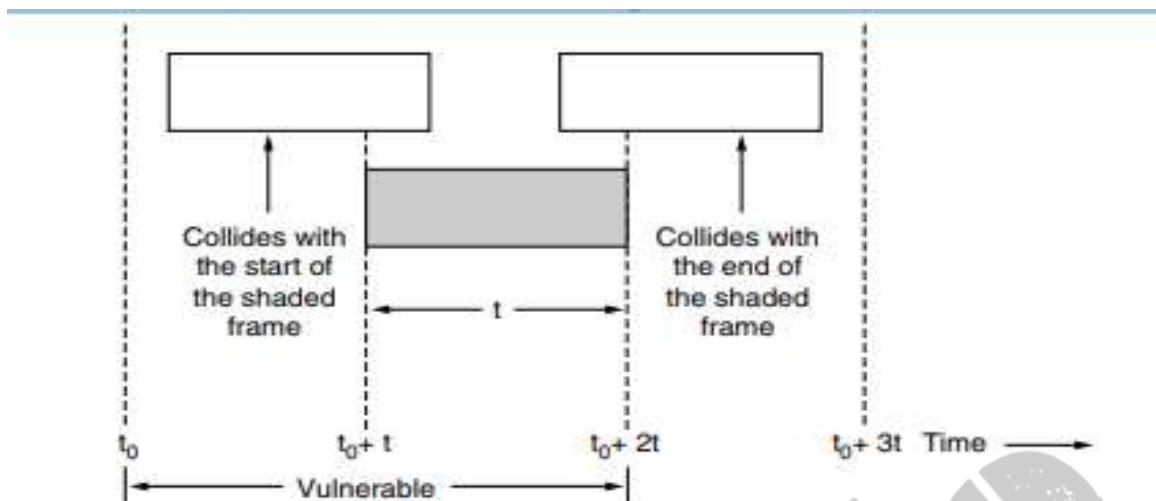


Figure 4-2. Vulnerable period for the shaded frame.

The probability that  $k$  frames are generated during a given frame time, in which  $G$  frames are expected, is given by the Poisson distribution  $\Pr[k] = \frac{G^k}{k!} e^{-G}$  (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  $P_0 = e^{-2G}$ . Using  $S = GP_0$ , 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.

$$S = G \times e^{-2G}$$

## 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.

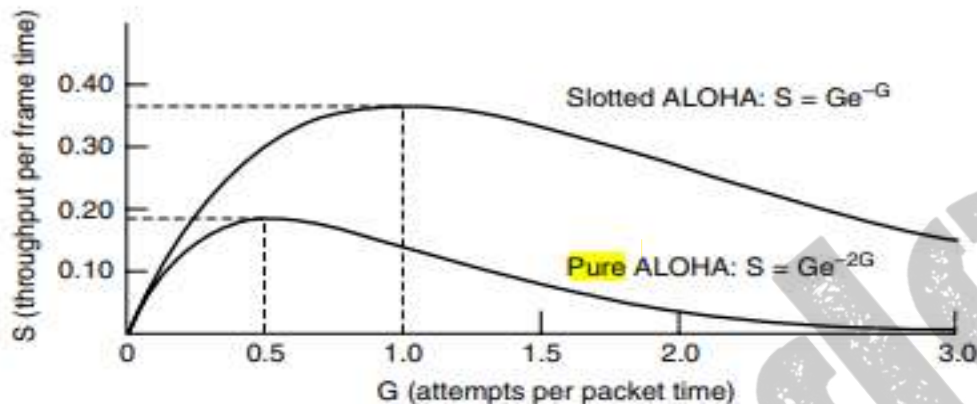
In Roberts' method, which has come to be known as slotted ALOHA—in contrast to Abramson's pure ALOHA—a station is not permitted to send when-ever the user types a line. Instead, it is required to wait for the **beginning of the next slot**. Thus, the continuous time ALOHA is turned into a discrete time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

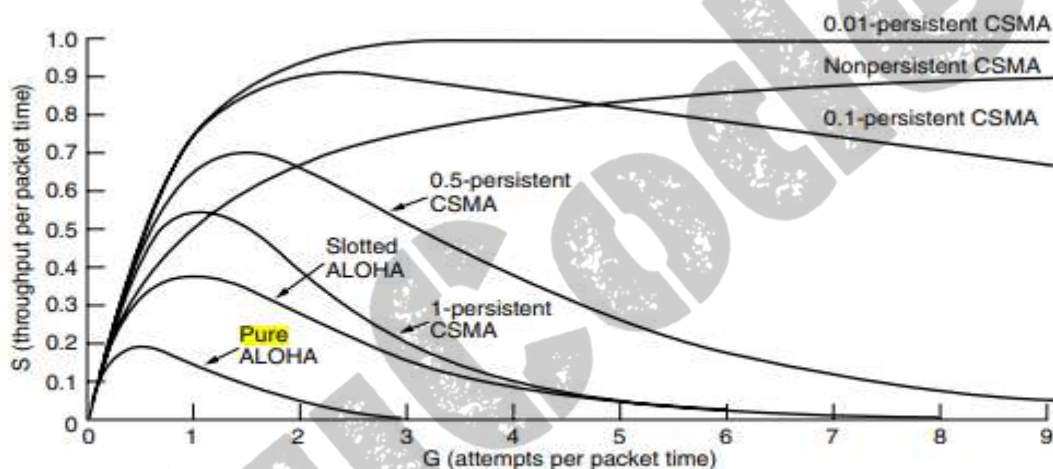


Figure 4-4. Comparison of the channel utilization versus load for various random access protocols.

## 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).

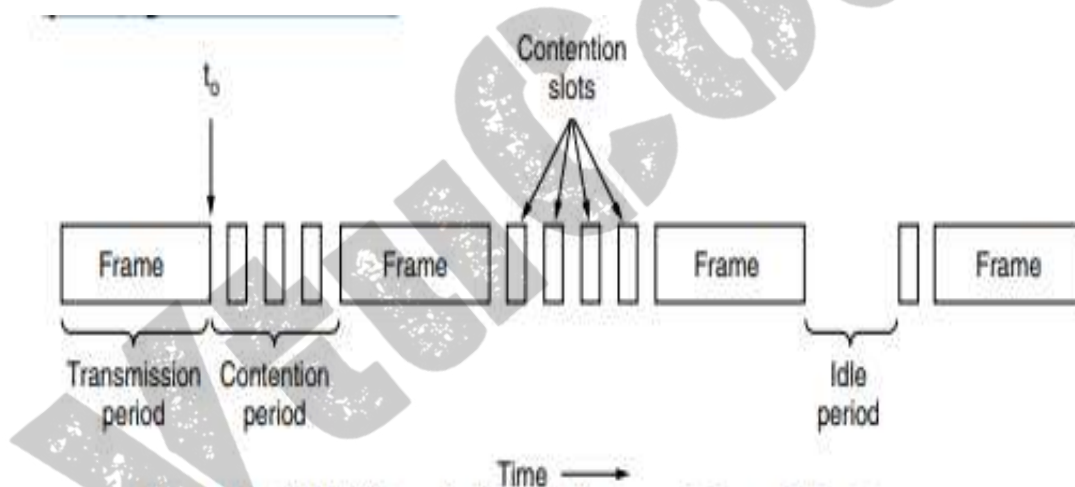


Figure 4-5. CSMA/CD can be in contention, transmission, or idle state.

### 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  $d / (d+N)$ .

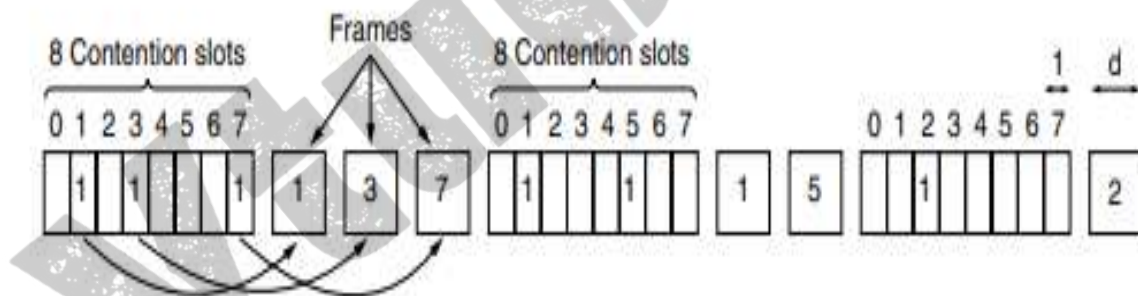


Figure 4-6. The basic bit-map protocol.

### 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.

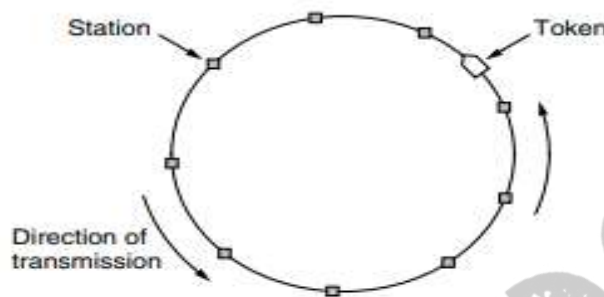


Figure 4-7. Token ring.

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 addresses with a channel that combines transmissions.
- All addresses are assumed to be the same length. The bits in each address 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 transmission 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.

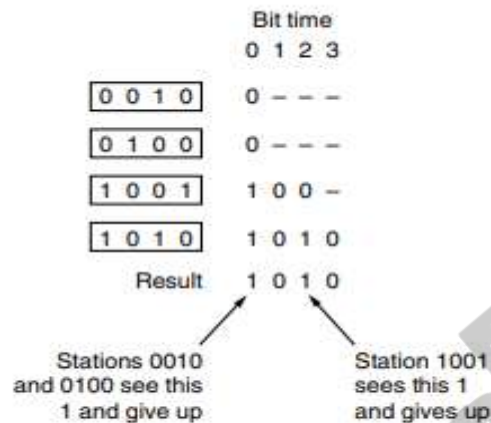
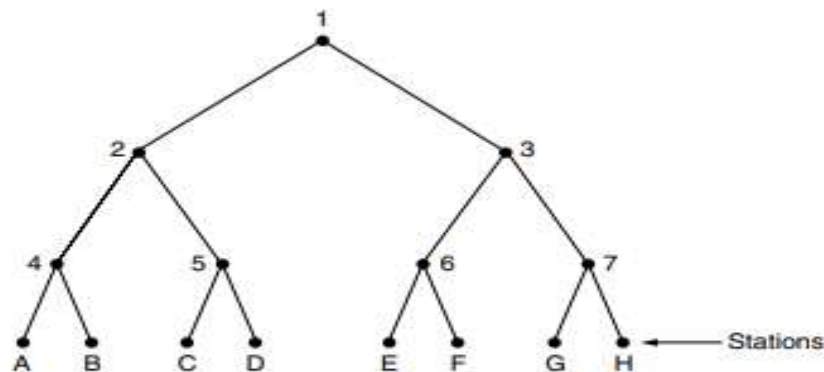


Figure 4-8. The binary countdown protocol. A dash indicates silence.

#### 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.



**Figure 4-10.** The tree for eight stations.

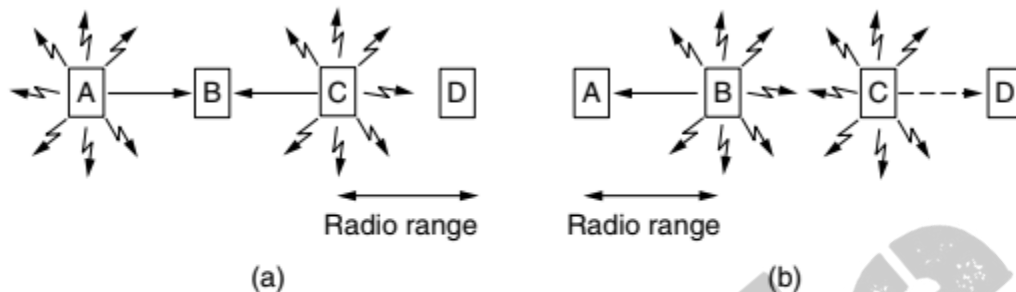
In essence, if a collision occurs during slot 0, the entire tree is searched, depth first, to locate all ready stations. Each bit slot is associated with some particular node in the tree. If a collision occurs, the search continues recursively with the node's left and right children. If a bit slot is 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  $i = \log_2 q$ .

## 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.

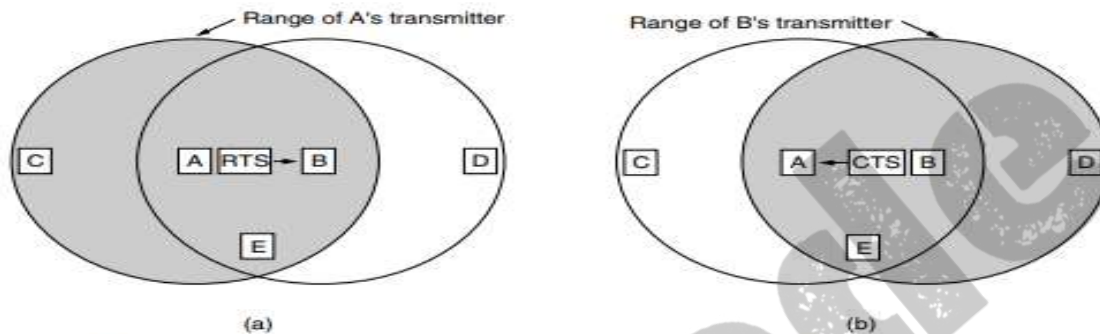


**Figure 4-11.** A wireless LAN. (a) A and C are hidden terminals when transmitting to B. (b) B and C are exposed terminals when transmitting to A and D.

- 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 because A 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.