

```

/* 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.

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

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

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

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

The sequence numbers within the sender's window represent frames 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.

Since frames currently within the sender's window may ultimately be lost or damaged in transit, the sender must keep all of these frames in its memory for possible retransmission. Thus, if the maximum window size is n , the sender needs n buffers to hold the unacknowledged frames. If the window ever grows to its

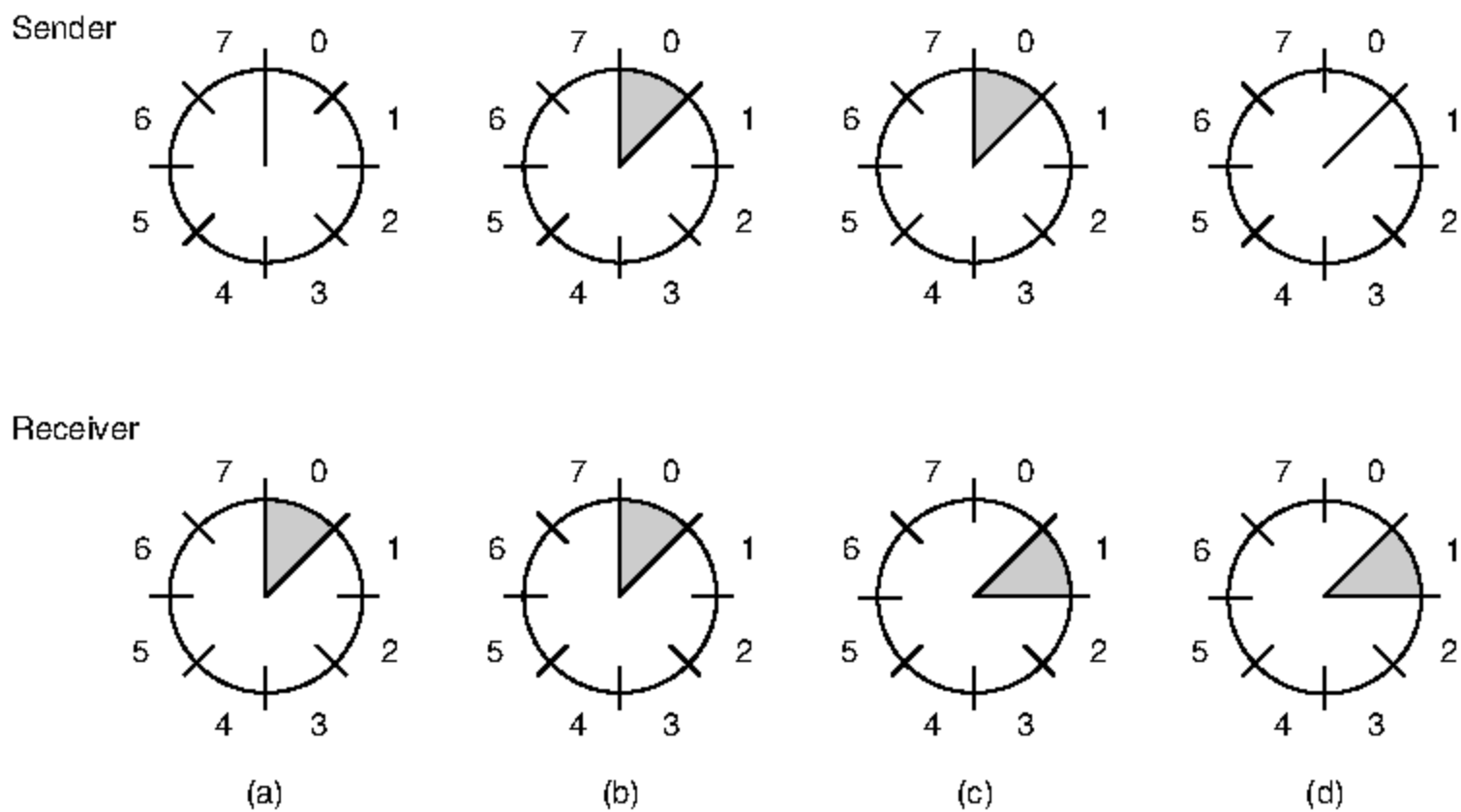


Figure 3-15. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

maximum size, the sending data link layer must forcibly shut off the network layer until another buffer becomes free.

The receiving data link layer's window corresponds to the frames it may accept. Any frame falling within the window is put in the receiver's buffer. When a frame whose sequence number is equal to the lower edge of the window is received, it is passed to the network layer and the window is rotated by one. Any frame falling outside the window is discarded. In all of these cases, a subsequent acknowledgement is generated so that the sender may work out how to proceed. Note that a window size of 1 means that the data link layer only accepts frames in order, but for larger windows this is not so. The network layer, in contrast, is always fed data in the proper order, regardless of the data link layer's window size.

Figure 3-15 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.

3.4.1 A One-Bit Sliding Window Protocol

Before tackling the general case, let us examine 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 */
    }
}
```

Figure 3-16. A 1-bit sliding window protocol.

Under normal circumstances, one of the two data link layers goes first and transmits the first frame. In other words, only one of the data link layer programs should contain the *to_physical_layer* and *start_timer* procedure calls outside the main loop. The starting machine fetches the first packet from its network layer, builds a frame from it, and sends it. When this (or any) frame arrives, the receiving data link layer checks to see if it is a duplicate, just as in protocol 3. If the frame is the one expected, it is passed to the network layer and the receiver's window is slid up.

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

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

When the first valid frame arrives at computer *B*, it will be accepted and *frame_expected* will be set to 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.

However, to show how subtle protocol interactions can be, we note that a peculiar situation arises if both sides simultaneously send an initial packet. 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.

However, if *A* and *B* simultaneously initiate communication, their first frames cross, and the data link layers then get into situation (b). In (a) each frame arrival brings a new packet for the network layer; there are no duplicates. In (b) half of the frames contain duplicates, even though there are no transmission errors. Similar situations can occur as a result of premature timeouts, even when one side clearly starts first. In fact, if multiple premature timeouts occur, frames may be sent three or more times, wasting valuable bandwidth.

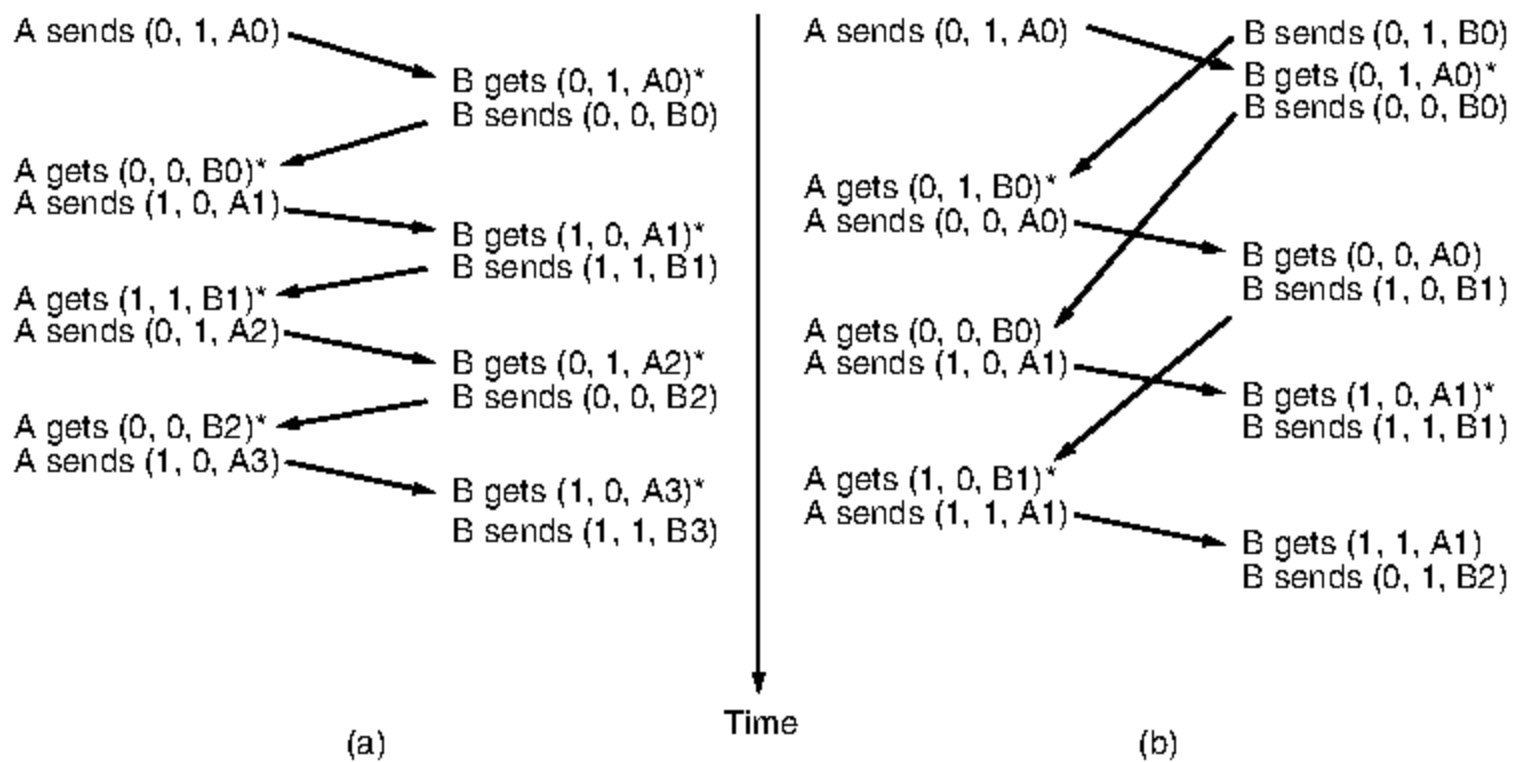


Figure 3-17. Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

3.4.2 A Protocol Using Go-Back-N

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

The problem described here can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame. If we relax that restriction, much better efficiency can be achieved. Basically, the solution lies in allowing the sender to transmit up to w frames before blocking, instead of just 1. With a large enough choice of w the sender will be able to continuously transmit frames since the acknowledgements will arrive for previous frames before the window becomes full, preventing the sender from blocking.

To find an appropriate value for w we need to know how many frames can fit inside the channel as they propagate from sender to receiver. This capacity is determined by the bandwidth in bits/sec multiplied by the one-way transit time, or the **bandwidth-delay product** of the link. We can divide this quantity by the number of bits in a frame to express it as a number of frames. Call this quantity BD . Then w should be set to $2BD + 1$. Twice the bandwidth-delay is the number of frames that can be outstanding if the sender continuously sends frames when the round-trip time to receive an acknowledgement is considered. The “+1” is because an acknowledgement frame will not be sent until after a complete frame is received.

For the example link with a bandwidth of 50 kbps and a one-way transit time of 250 msec, the bandwidth-delay product is 12.5 kbit or 12.5 frames of 1000 bits each. $2BD + 1$ is then 26 frames. Assume the sender begins sending frame 0 as before and sends a new frame every 20 msec. By the time it has finished sending 26 frames, at $t = 520$ msec, the acknowledgement for frame 0 will have just arrived. Thereafter, acknowledgements will arrive every 20 msec, so the sender will always get permission to continue just when it needs it. From then onwards, 25 or 26 unacknowledged frames will always be outstanding. Put in other terms, the sender's maximum window size is 26.

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

Two basic approaches are available for dealing with errors in the presence of pipelining, both of which are shown in Fig. 3-18.

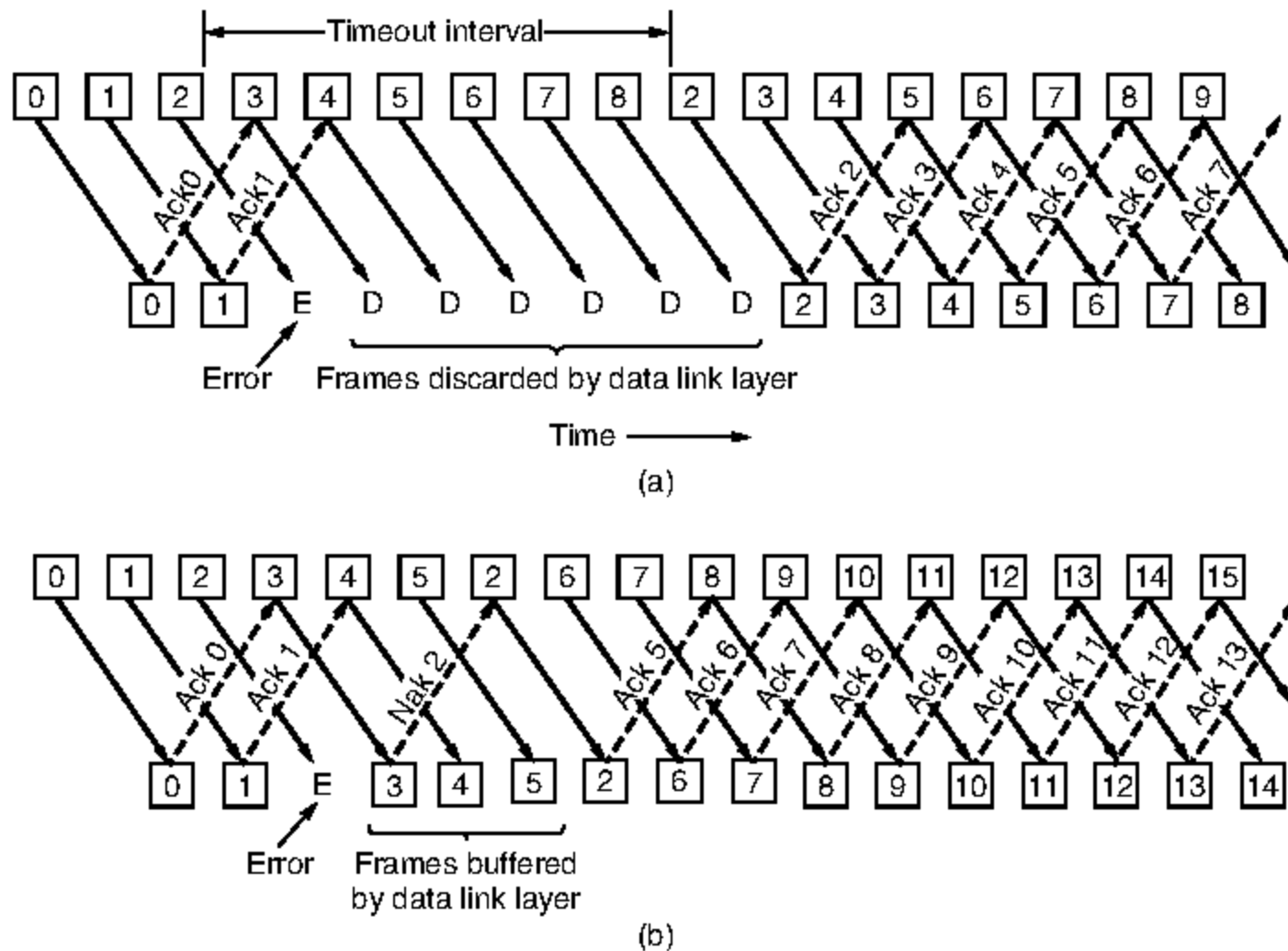


Figure 3-18. Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1 and (b) receiver's window size is large.

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. It can also acknowledge all frames up to and including 5, as shown in the figure. If the NAK should get lost, eventually the sender will time out for frame 2 and send it (and only it) of its own accord, but that may be a quite a while later.

These two alternative approaches are trade-offs between efficient use of bandwidth and data link layer buffer space. Depending on which resource is scarcer, one or the other can be used. Figure 3-19 shows a go-back-n protocol in which the receiving data link layer only accepts frames in order; frames following an error are discarded. In this protocol, for the first time we have dropped the assumption that the network layer always has an infinite supply of packets to send. When the network layer has a packet it wants to send, it can cause a *network_layer_ready* event to happen. However, to enforce the flow control limit on the sender window or the number of unacknowledged frames that may be outstanding at any time, the data link layer must be able to keep the network layer from bothering it with more work. The library procedures *enable_network_layer* and *disable_network_layer* do this job.

The maximum number of frames that may be outstanding at any instant is not the same as the size of the sequence number space. For go-back-n, MAX_SEQ frames may be outstanding at any instant, even though there are $MAX_SEQ + 1$ distinct sequence numbers (which are 0, 1, ..., MAX_SEQ). We will see an even tighter restriction for the next protocol, selective repeat. To see why this restriction is required, consider the following scenario with $MAX_SEQ = 7$:

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

/* 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.

The question is this: did all eight frames belonging to the second batch arrive successfully, or did all eight get lost (counting discards following an error as lost)? In both cases the receiver would be sending frame 7 as the acknowledgement.

The sender has no way of telling. For this reason the maximum number of outstanding frames must be restricted to *MAX_SEQ*.

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

For this protocol, we assume that there is always reverse traffic on which to piggyback acknowledgements. Protocol 4 does not need this assumption since it sends back one frame every time it receives a frame, even if it has already sent that frame. In the next protocol we will solve the problem of one-way traffic in an elegant way.

Because protocol 5 has multiple outstanding frames, it logically needs multiple timers, one per outstanding frame. Each frame times out independently of all the other ones. 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.

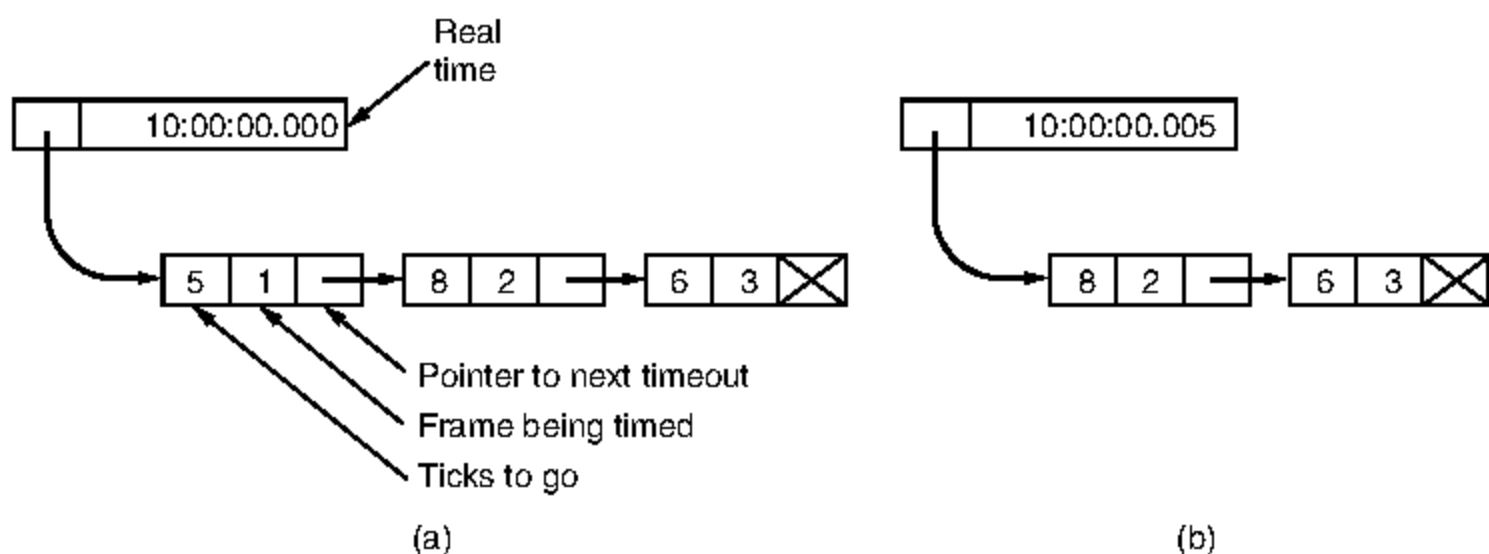


Figure 3-20. Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

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. In protocol 5, both of these routines have been given a parameter indicating which frame is to be timed.

3.4.3 A Protocol Using Selective Repeat

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.

In this protocol, both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively. The sender's window size starts out at 0 and grows to some predefined maximum. The receiver's window, in contrast, is always fixed in size and equal to the predetermined maximum. The receiver has a buffer reserved for each sequence number within its fixed window. Associated with each buffer is a bit (*arrived*) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function *between* to see if it falls within the window. If so and if it has not already been received, it is accepted and stored. This action is taken without regard to whether or not 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.

Nonsequential receive introduces further constraints on frame sequence numbers compared to protocols in which frames are only accepted in order. We can illustrate the trouble most easily with an example. Suppose that we have a 3-bit sequence number, so that the sender is permitted to transmit up to seven frames before being required to wait for an acknowledgement. Initially, the sender'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.

It is at this point that disaster strikes in the form of a lightning bolt hitting the telephone pole and wiping out all the acknowledgements. The protocol should operate correctly despite this disaster. The sender eventually times out and retransmits frame 0. When this frame arrives at the receiver, a check is made to see if it falls within the receiver's window. Unfortunately, in Fig. 3-22(b) frame 0 is

/* 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.

within the new window, so it is accepted as a new frame. The receiver also sends a (piggybacked) acknowledgement for frame 6, since 0 through 6 have been received.

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

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

The way out of this dilemma lies in making sure that after the receiver has advanced its window there is no overlap with the original window. To ensure that there is no overlap, the maximum window size should be at most half the range of the sequence numbers. 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 $(MAX_SEQ + 1)/2$.

An interesting question is: how many buffers must the receiver have? Under no conditions will it ever accept frames whose sequence numbers are below the lower edge of the window or frames whose sequence numbers are above the upper edge of the window. Consequently, the number of buffers needed is equal to the window size, not to the range of sequence numbers. In the preceding example of a 3-bit sequence number, four buffers, numbered 0 through 3, are needed. When frame i arrives, it is put in buffer $i \bmod 4$. Notice that although i and $(i + 4) \bmod 4$ are “competing” for the same buffer, they are never within the window at the same time, because that would imply a window size of at least 5.

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

Protocol 6 also relaxes the implicit assumption that the channel is heavily loaded. We made this assumption in protocol 5 when we relied on frames being sent in the reverse direction on which to piggyback acknowledgements. If the reverse traffic is light, the acknowledgements may be held up for a long period of time, which can cause problems. In the extreme, if there is a lot of traffic in one

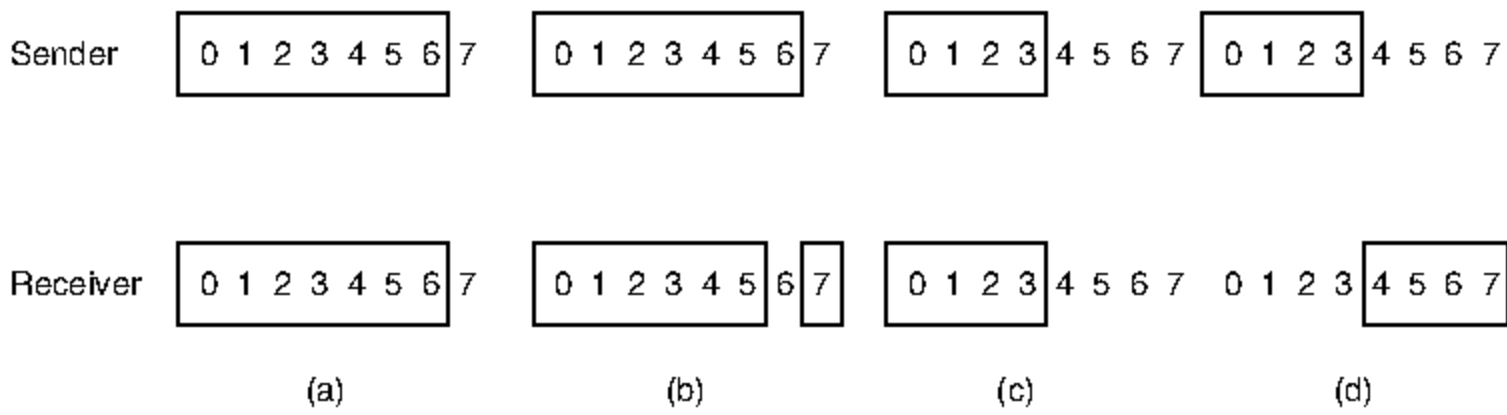


Figure 3-22. (a) Initial situation with a window of size 7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

direction and no traffic in the other direction, the protocol will block when the sender window reaches its maximum.

To relax this assumption, an auxiliary timer is started by *start_ack_timer* after an in-sequence data frame arrives. If no reverse traffic has presented itself before this timer expires, a separate acknowledgement frame is sent. An interrupt due to the auxiliary timer is called an *ack_timeout* event. With this arrangement, traffic flow in only one direction is possible because the lack of reverse data frames onto which acknowledgements can be piggybacked is no longer an obstacle. Only one auxiliary timer exists, and if *start_ack_timer* is called while the timer is running, it has no effect. The timer is not reset or extended since its purpose is to provide some minimum rate of acknowledgements.

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

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

In some situations, the time required for a frame to propagate to the destination, be processed there, and have the acknowledgement come back is (nearly) constant. In these situations, the sender can adjust its timer to be “tight,” just slightly larger than the normal time interval expected between sending a frame and receiving its acknowledgement. NAKs are not useful in this case.

However, in other situations the time can be highly variable. For example, if the reverse traffic is sporadic, the time before acknowledgement will be shorter when there is reverse traffic and longer when there is not. The sender is faced with the choice of either setting the interval to a small value (and risking unnecessary retransmissions), or setting it to a large value (and going idle for a long period after an error). Both choices waste bandwidth. In general, if the standard deviation of the acknowledgement interval is large compared to the interval itself, the timer is set “loose” to be conservative. NAKs can then appreciably speed up retransmission of lost or damaged frames.

Closely related to the matter of timeouts and NAKs is the question of determining which frame caused a timeout. In protocol 5, it is always *ack_expected*, because it is always the oldest. In protocol 6, there is no trivial way to determine who timed out. Suppose that frames 0 through 4 have been transmitted, meaning that the list of outstanding frames is 01234, in order from oldest to youngest. Now imagine that 0 times out, 5 (a new frame) is transmitted, 1 times out, 2 times out, and 6 (another new frame) is transmitted. At this point the list of outstanding frames is 3405126, from oldest to youngest. If all inbound traffic (i.e., acknowledgement-bearing frames) is lost for a while, the seven outstanding frames will time out in that order.

To keep the example from getting even more complicated than it already is, we have not shown the timer administration. Instead, we just assume that the variable *oldest_frame* is set upon timeout to indicate which frame timed out.

3.5 EXAMPLE DATA LINK PROTOCOLS

Within a single building, LANs are widely used for interconnection, but most wide-area network infrastructure is built up from point-to-point lines. In Chap. 4, we will look at LANs. Here we will examine the data link protocols found on point-to-point lines in the Internet in two common situations. The first situation is when packets are sent over SONET optical fiber links in wide-area networks. These links are widely used, for example, to connect routers in the different locations of an ISP’s network.

The second situation is for ADSL links running on the local loop of the telephone network at the edge of the Internet. These links connect millions of individuals and businesses to the Internet.

The Internet needs point-to-point links for these uses, as well as dial-up modems, leased lines, and cable modems, and so on. A standard protocol called **PPP**

(**Point-to-Point Protocol**) is used to send packets over these links. PPP is defined in RFC 1661 and further elaborated in RFC 1662 and other RFCs (Simpson, 1994a, 1994b). SONET and ADSL links both apply PPP, but in different ways.

3.5.1 Packet over SONET

SONET, which we covered in Sec. 2.6.4, is the physical layer protocol that is most commonly used over the wide-area optical fiber links that make up the backbone of communications networks, including the telephone system. It provides a bitstream that runs at a well-defined rate, for example 2.4 Gbps for an OC-48 link. This bitstream is organized as fixed-size byte payloads that recur every 125 μ sec, whether or not there is user data to send.

To carry packets across these links, some framing mechanism is needed to distinguish occasional packets from the continuous bitstream in which they are transported. PPP runs on IP routers to provide this mechanism, as shown in Fig. 3-23.

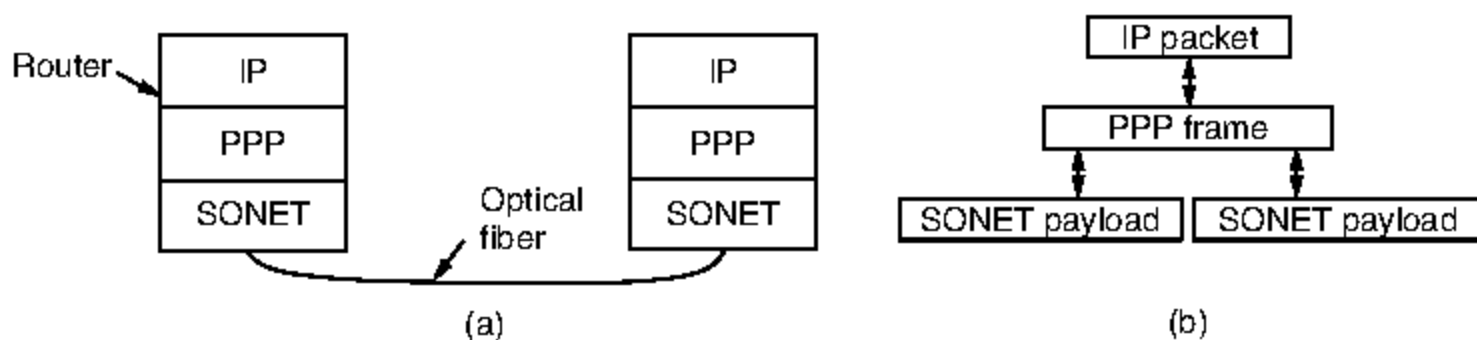


Figure 3-23. Packet over SONET. (a) A protocol stack. (b) Frame relationships.

PPP improves on an earlier, simpler protocol called **SLIP** (**Serial Line Internet Protocol**) and is used to handle error detection link configuration, support multiple protocols, permit authentication, and more. With a wide set of options, PPP provides three main features:

1. A framing method that unambiguously delineates the end of one frame and the start of the next one. The frame format also handles error detection.
2. A link control protocol for bringing lines up, testing them, negotiating options, and bringing them down again gracefully when they are no longer needed. This protocol is called **LCP** (**Link Control Protocol**).
3. A way to negotiate network-layer options in a way that is independent of the network layer protocol to be used. The method chosen is to have a different **NCP** (**Network Control Protocol**) for each network layer supported.

The PPP frame format was chosen to closely resemble the frame format of **HDLC (High-level Data Link Control)**, a widely used instance of an earlier family of protocols, since there was no need to reinvent the wheel.

The primary difference between PPP and HDLC is that PPP is byte oriented rather than bit oriented. In particular, PPP uses byte stuffing and all frames are an integral number of bytes. HDLC uses bit stuffing and allows frames of, say, 30.25 bytes.

There is a second major difference in practice, however. HDLC provides reliable transmission with a sliding window, acknowledgements, and timeouts in the manner we have studied. PPP can also provide reliable transmission in noisy environments, such as wireless networks; the exact details are defined in RFC 1663. However, this is rarely done in practice. Instead, an “unnumbered mode” is nearly always used in the Internet to provide connectionless unacknowledged service.

The PPP frame format is shown in Fig. 3-24. All PPP frames begin with the standard HDLC flag byte of 0x7E (01111110). The flag byte is stuffed if it occurs within the *Payload* field using the escape byte 0x7D. The following byte is the escaped byte XORed with 0x20, which flips the 5th bit. For example, 0x7D 0x5E is the escape sequence for the flag byte 0x7E. This means the start and end of frames can be searched for simply by scanning for the byte 0x7E since it will not occur elsewhere. The destuffing rule when receiving a frame is to look for 0x7D, remove it, and XOR the following byte with 0x20. Also, only one flag byte is needed between frames. Multiple flag bytes can be used to fill the link when there are no frames to be sent.

After the start-of-frame flag byte comes the *Address* field. This field is always set to the binary value 11111111 to indicate that all stations are to accept the frame. Using this value avoids the issue of having to assign data link addresses.

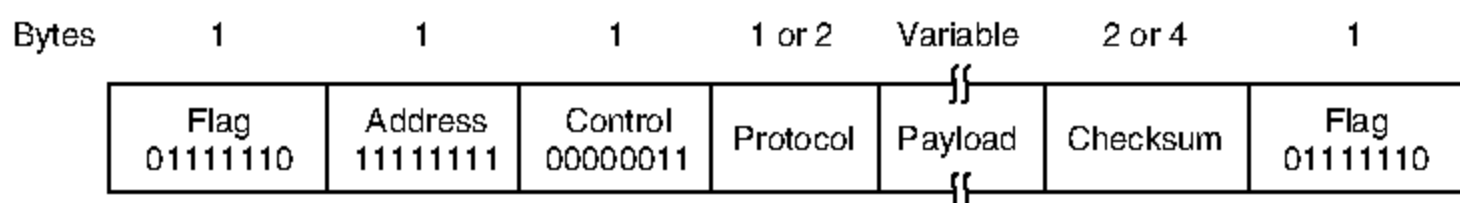


Figure 3-24. The PPP full frame format for unnumbered mode operation.

The *Address* field is followed by the *Control* field, the default value of which is 00000011. This value indicates an unnumbered frame.

Since the *Address* and *Control* fields are always constant in the default configuration, LCP provides the necessary mechanism for the two parties to negotiate an option to omit them altogether and save 2 bytes per frame.

The fourth PPP field is the *Protocol* field. Its job is to tell what kind of packet is in the *Payload* field. Codes starting with a 0 bit are defined for IP version 4, IP version 6, and other network layer protocols that might be used, such as IPX and

AppleTalk. Codes starting with a 1 bit are used for PPP configuration protocols, including LCP and a different NCP for each network layer protocol supported. The default size of the *Protocol* field is 2 bytes, but it can be negotiated down to 1 byte using LCP. The designers were perhaps overly cautious in thinking that someday there might be more than 256 protocols in use.

The *Payload* field is variable length, up to some negotiated maximum. If the length is not negotiated using LCP during line setup, a default length of 1500 bytes is used. Padding may follow the payload if it is needed.

After the *Payload* field comes the *Checksum* field, which is normally 2 bytes, but a 4-byte checksum can be negotiated. The 4-byte checksum is in fact the same 32-bit CRC whose generator polynomial is given at the end of Sec. 3.2.2. The 2-byte checksum is also an industry-standard CRC.

PPP is a framing mechanism that can carry the packets of multiple protocols over many types of physical layers. To use PPP over SONET, the choices to make are spelled out in RFC 2615 (Malis and Simpson, 1999). A 4-byte checksum is used, since this is the primary means of detecting transmission errors over the physical, link, and network layers. It is recommended that the *Address*, *Control*, and *Protocol* fields not be compressed, since SONET links already run at relatively high rates.

There is also one unusual feature. The PPP payload is scrambled (as described in Sec. 2.5.1) before it is inserted into the SONET payload. Scrambling XORs the payload with a long pseudorandom sequence before it is transmitted. The issue is that the SONET bitstream needs frequent bit transitions for synchronization. These transitions come naturally with the variation in voice signals, but in data communication the user chooses the information that is sent and might send a packet with a long run of 0s. With scrambling, the likelihood of a user being able to cause problems by sending a long run of 0s is made extremely low.

Before PPP frames can be carried over SONET lines, the PPP link must be established and configured. The phases that the link goes through when it is brought up, used, and taken down again are shown in Fig. 3-25.

The link starts in the *DEAD* state, which means that there is no connection at the physical layer. When a physical layer connection is established, the link moves to *ESTABLISH*. At this point, the PPP peers exchange a series of LCP packets, each carried in the *Payload* field of a PPP frame, to select the PPP options for the link from the possibilities mentioned above. The initiating peer proposes options, and the responding peer either accepts or rejects them, in whole or part. The responder can also make alternative proposals.

If LCP option negotiation is successful, the link reaches the *AUTHENTICATE* state. Now the two parties can check each other's identities, if desired. If authentication is successful, the *NETWORK* state is entered and a series of NCP packets are sent to configure the network layer. It is difficult to generalize about the NCP protocols because each one is specific to some network layer protocol and allows configuration requests to be made that are specific to that protocol.

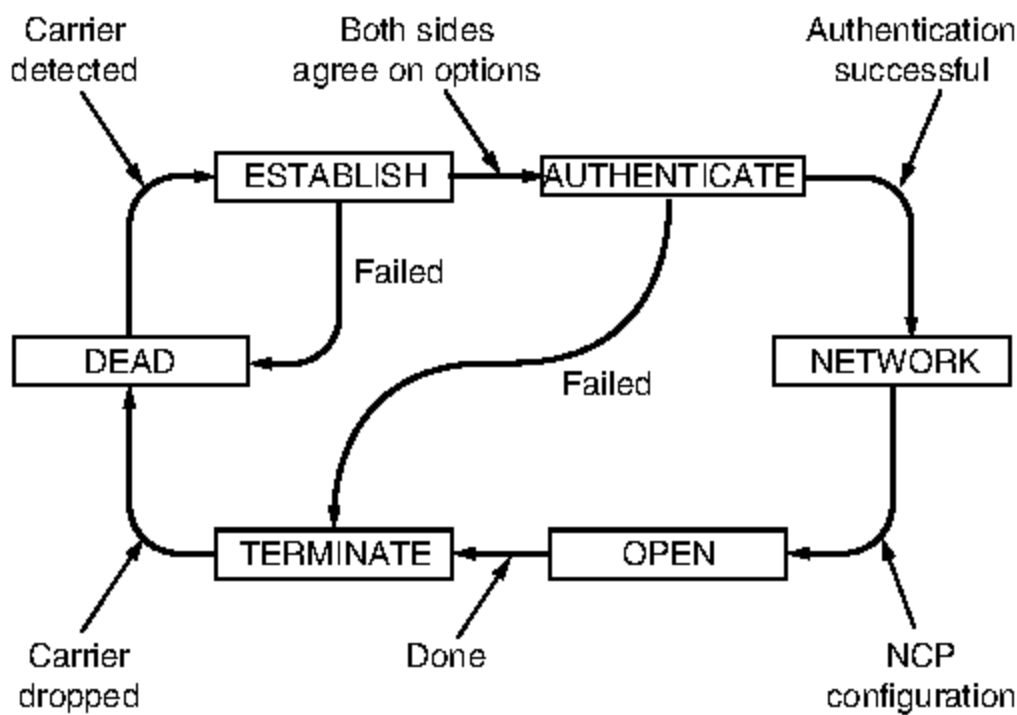


Figure 3-25. State diagram for bringing a PPP link up and down.

For IP, for example, the assignment of IP addresses to both ends of the link is the most important possibility.

Once *OPEN* is reached, data transport can take place. It is in this state that IP packets are carried in PPP frames across the SONET line. When data transport is finished, the link moves into the *TERMINATE* state, and from there it moves back to the *DEAD* state when the physical layer connection is dropped.

3.5.2 ADSL (Asymmetric Digital Subscriber Loop)

ADSL connects millions of home subscribers to the Internet at megabit/sec rates over the same telephone local loop that is used for plain old telephone service. In Sec. 2.5.3, we described how a device called a DSL modem is added on the home side. It sends bits over the local loop to a device called a DSLAM (DSL Access Multiplexer), pronounced “dee-slam,” in the telephone company’s local office. Now we will explore in more detail how packets are carried over ADSL links.

The overall picture for the protocols and devices used with ADSL is shown in Fig. 3-26. Different protocols are deployed in different networks, so we have chosen to show the most popular scenario. Inside the home, a computer such as a PC sends IP packets to the DSL modem using a link layer like Ethernet. The DSL modem then sends the IP packets over the local loop to the DSLAM using the protocols that we are about to study. At the DSLAM (or a router connected to it depending on the implementation) the IP packets are extracted and enter an ISP network so that they may reach any destination on the Internet.

The protocols shown over the ADSL link in Fig. 3-26 start at the bottom with the ADSL physical layer. They are based on a digital modulation scheme called

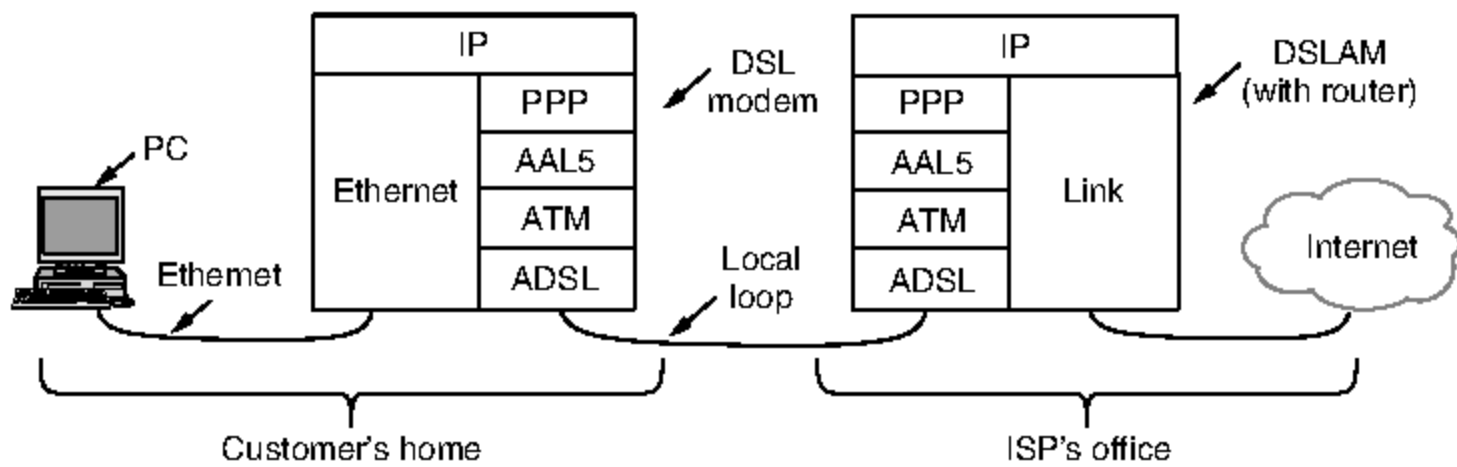


Figure 3-26. ADSL protocol stacks.

orthogonal frequency division multiplexing (also known as discrete multitone), as we saw in Sec 2.5.3. Near the top of the stack, just below the IP network layer, is PPP. This protocol is the same PPP that we have just studied for packet over SONET transports. It works in the same way to establish and configure the link and carry IP packets.

In between ADSL and PPP are ATM and AAL5. These are new protocols that we have not seen before. **ATM (Asynchronous Transfer Mode)** was designed in the early 1990s and launched with incredible hype. It promised a network technology that would solve the world's telecommunications problems by merging voice, data, cable television, telegraph, carrier pigeon, tin cans connected by strings, tom toms, and everything else into an integrated system that could do everything for everyone. This did not happen. In large part, the problems of ATM were similar to those we described concerning the OSI protocols, that is, bad timing, technology, implementation, and politics. Nevertheless, ATM was much more successful than OSI. While it has not taken over the world, it remains widely used in niches including broadband access lines such as DSL, and WAN links inside telephone networks.

ATM is a link layer that is based on the transmission of fixed-length **cells** of information. The "Asynchronous" in its name means that the cells do not always need to be sent in the way that bits are continuously sent over synchronous lines, as in SONET. Cells only need to be sent when there is information to carry. ATM is a connection-oriented technology. Each cell carries a **virtual circuit** identifier in its header and devices use this identifier to forward cells along the paths of established connections.

The cells are each 53 bytes long, consisting of a 48-byte payload plus a 5-byte header. By using small cells, ATM can flexibly divide the bandwidth of a physical layer link among different users in fine slices. This ability is useful when, for example, sending both voice and data over one link without having long data packets that would cause large variations in the delay of the voice samples. The unusual choice for the cell length (e.g., compared to the more natural choice of a

power of 2) is an indication of just how political the design of ATM was. The 48-byte size for the payload was a compromise to resolve a deadlock between Europe, which wanted 32-byte cells, and the U.S., which wanted 64-byte cells. A brief overview of ATM is given by Siu and Jain (1995).

To send data over an ATM network, it needs to be mapped into a sequence of cells. This mapping is done with an ATM adaptation layer in a process called segmentation and reassembly. Several adaptation layers have been defined for different services, ranging from periodic voice samples to packet data. The main one used for packet data is **AAL5 (ATM Adaptation Layer 5)**.

An AAL5 frame is shown in Fig. 3-27. Instead of a header, it has a trailer that gives the length and has a 4-byte CRC for error detection. Naturally, the CRC is the same one used for PPP and IEEE 802 LANs like Ethernet. Wang and Crowcroft (1992) have shown that it is strong enough to detect nontraditional errors such as cell reordering. As well as a payload, the AAL5 frame has padding. This rounds out the overall length to be a multiple of 48 bytes so that the frame can be evenly divided into cells. No addresses are needed on the frame as the virtual circuit identifier carried in each cell will get it to the right destination.

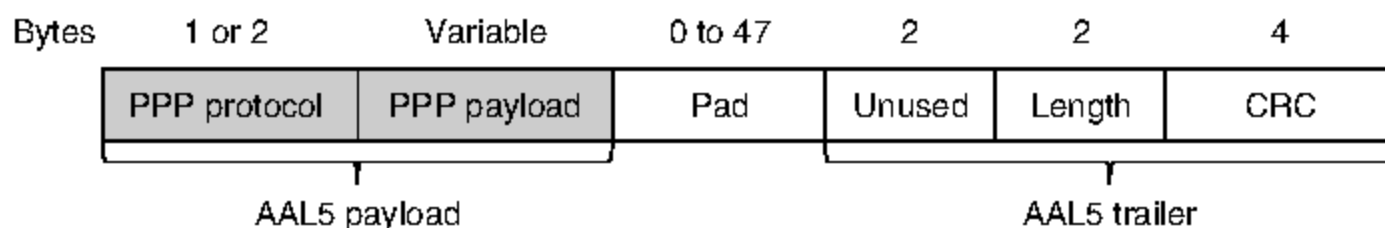


Figure 3-27. AAL5 frame carrying PPP data.

Now that we have described ATM, we have only to describe how PPP makes use of ATM in the case of ADSL. It is done with yet another standard called **PPPoA (PPP over ATM)**. This standard is not really a protocol (so it does not appear in Fig. 3-26) but more a specification of how to work with both PPP and AAL5 frames. It is described in RFC 2364 (Gross et al., 1998).

Only the PPP protocol and payload fields are placed in the AAL5 payload, as shown in Fig. 3-27. The protocol field indicates to the DSLAM at the far end whether the payload is an IP packet or a packet from another protocol such as LCP. The far end knows that the cells contain PPP information because an ATM virtual circuit is set up for this purpose.

Within the AAL5 frame, PPP framing is not needed as it would serve no purpose; ATM and AAL5 already provide the framing. More framing would be worthless. The PPP CRC is also not needed because AAL5 already includes the very same CRC. This error detection mechanism supplements the ADSL physical layer coding of a Reed-Solomon code for error correction and a 1-byte CRC for the detection of any remaining errors not otherwise caught. This scheme has a much more sophisticated error-recovery mechanism than when packets are sent over a SONET line because ADSL is a much noisier channel.

3.6 SUMMARY

The task of the data link layer is to convert the raw bit stream offered by the physical layer into a stream of frames for use by the network layer. The link layer can present this stream with varying levels of reliability, ranging from connectionless, unacknowledged service to reliable, connection-oriented service.

Various framing methods are used, including byte count, byte stuffing, and bit stuffing. Data link protocols can provide error control to detect or correct damaged frames and to retransmit lost frames. To prevent a fast sender from overrunning a slow receiver, the data link protocol can also provide flow control. The sliding window mechanism is widely used to integrate error control and flow control in a simple way. When the window size is 1 packet, the protocol is stop-and-wait.

Codes for error correction and detection add redundant information to messages by using a variety of mathematical techniques. Convolutional codes and Reed-Solomon codes are widely deployed for error correction, with low-density parity check codes increasing in popularity. The codes for error detection that are used in practice include cyclic redundancy checks and checksums. All these codes can be applied at the link layer, as well as at the physical layer and higher layers.

We examined a series of protocols that provide a reliable link layer using acknowledgements and retransmissions, or ARQ (Automatic Repeat reQuest), under more realistic assumptions. Starting from an error-free environment in which the receiver can handle any frame sent to it, we introduced flow control, followed by error control with sequence numbers and the stop-and-wait algorithm. Then we used the sliding window algorithm to allow bidirectional communication and introduce the concept of piggybacking. The last two protocols pipeline the transmission of multiple frames to prevent the sender from blocking on a link with a long propagation delay. The receiver can either discard all frames other than the next one in sequence, or buffer out-of-order frames and send negative acknowledgements for greater bandwidth efficiency. The former strategy is a go-back-n protocol, and the latter strategy is a selective repeat protocol.

The Internet uses PPP as the main data link protocol over point-to-point lines. It provides a connectionless unacknowledged service, using flag bytes to delimit frames and a CRC for error detection. It is used to carry packets across a range of links, including SONET links in wide-area networks and ADSL links for the home.

PROBLEMS

1. An upper-layer packet is split into 10 frames, each of which has an 80% chance of arriving undamaged. If no error control is done by the data link protocol, how many times must the message be sent on average to get the entire thing through?

2. The following character encoding is used in a data link protocol:
A: 01000111 B: 11100011 FLAG: 01111110 ESC: 11100000
Show the bit sequence transmitted (in binary) for the four-character frame A B ESC FLAG when each of the following framing methods is used:
 - (a) Byte count.
 - (b) Flag bytes with byte stuffing.
 - (c) Starting and ending flag bytes with bit stuffing.
3. The following data fragment occurs in the middle of a data stream for which the byte-stuffing algorithm described in the text is used: A B ESC C ESC FLAG FLAG D. What is the output after stuffing?
4. What is the maximum overhead in byte-stuffing algorithm?
5. One of your classmates, Scrooge, has pointed out that it is wasteful to end each frame with a flag byte and then begin the next one with a second flag byte. One flag byte could do the job as well, and a byte saved is a byte earned. Do you agree?
6. A bit string, 011110111110111110, needs to be transmitted at the data link layer. What is the string actually transmitted after bit stuffing?
7. Can you think of any circumstances under which an open-loop protocol (e.g., a Hamming code) might be preferable to the feedback-type protocols discussed throughout this chapter?
8. To provide more reliability than a single parity bit can give, an error-detecting coding scheme uses one parity bit for checking all the odd-numbered bits and a second parity bit for all the even-numbered bits. What is the Hamming distance of this code?
9. Sixteen-bit messages are transmitted using a Hamming code. How many check bits are needed to ensure that the receiver can detect and correct single-bit errors? Show the bit pattern transmitted for the message 1101001100110101. Assume that even parity is used in the Hamming code.
10. A 12-bit Hamming code whose hexadecimal value is 0xE4F arrives at a receiver. What was the original value in hexadecimal? Assume that not more than 1 bit is in error.
11. One way of detecting errors is to transmit data as a block of n rows of k bits per row and add parity bits to each row and each column. The bit in the lower-right corner is a parity bit that checks its row and its column. Will this scheme detect all single errors? Double errors? Triple errors? Show that this scheme cannot detect some four-bit errors.
12. Suppose that data are transmitted in blocks of sizes 1000 bits. What is the maximum error rate under which error detection and retransmission mechanism (1 parity bit per block) is better than using Hamming code? Assume that bit errors are independent of one another and no bit error occurs during retransmission.
13. A block of bits with n rows and k columns uses horizontal and vertical parity bits for error detection. Suppose that exactly 4 bits are inverted due to transmission errors. Derive an expression for the probability that the error will be undetected.

14. Using the convolutional coder of Fig. 3-7, what is the output sequence when the input sequence is 10101010 (left to right) and the internal state is initially all zero?
15. Suppose that a message 1001 1100 1010 0011 is transmitted using Internet Checksum (4-bit word). What is the value of the checksum?
16. What is the remainder obtained by dividing $x^7 + x^5 + 1$ by the generator polynomial $x^3 + 1$?
17. A bit stream 10011101 is transmitted using the standard CRC method described in the text. The generator polynomial is $x^3 + 1$. Show the actual bit string transmitted. Suppose that the third bit from the left is inverted during transmission. Show that this error is detected at the receiver's end. Give an example of bit errors in the bit string transmitted that will not be detected by the receiver.
18. A 1024-bit message is sent that contains 992 data bits and 32 CRC bits. CRC is computed using the IEEE 802 standardized, 32-degree CRC polynomial. For each of the following, explain whether the errors during message transmission will be detected by the receiver:
 - (a) There was a single-bit error.
 - (b) There were two isolated bit errors.
 - (c) There were 18 isolated bit errors.
 - (d) There were 47 isolated bit errors.
 - (e) There was a 24-bit long burst error.
 - (f) There was a 35-bit long burst error.
19. In the discussion of ARQ protocol in Section 3.3.3, a scenario was outlined that resulted in the receiver accepting two copies of the same frame due to a loss of acknowledgement frame. Is it possible that a receiver may accept multiple copies of the same frame when none of the frames (message or acknowledgement) are lost?
20. A channel has a bit rate of 4 kbps and a propagation delay of 20 msec. For what range of frame sizes does stop-and-wait give an efficiency of at least 50%?
21. In protocol 3, is it possible for the sender to start the timer when it is already running? If so, how might this occur? If not, why is it impossible?
22. A 3000-km-long T1 trunk is used to transmit 64-byte frames using protocol 5. If the propagation speed is 6 μ sec/km, how many bits should the sequence numbers be?
23. Imagine a sliding window protocol using so many bits for sequence numbers that wraparound never occurs. What relations must hold among the four window edges and the window size, which is constant and the same for both the sender and the receiver?
24. If the procedure *between* in protocol 5 checked for the condition $a \leq b \leq c$ instead of the condition $a \leq b < c$, would that have any effect on the protocol's correctness or efficiency? Explain your answer.
25. In protocol 6, when a data frame arrives, a check is made to see if the sequence number differs from the one expected and *no_nak* is true. If both conditions hold, a NAK is sent. Otherwise, the auxiliary timer is started. Suppose that the *else* clause were omitted. Would this change affect the protocol's correctness?

26. Suppose that the three-statement while loop near the end of protocol 6 was removed from the code. Would this affect the correctness of the protocol or just the performance? Explain your answer.
27. The distance from earth to a distant planet is approximately 9×10^{10} m. What is the channel utilization if a stop-and-wait protocol is used for frame transmission on a 64 Mbps point-to-point link? Assume that the frame size is 32 KB and the speed of light is 3×10^8 m/s.
28. In the previous problem, suppose a sliding window protocol is used instead. For what send window size will the link utilization be 100%? You may ignore the protocol processing times at the sender and the receiver.
29. In protocol 6, the code for *frame_arrival* has a section used for NAKs. This section is invoked if the incoming frame is a NAK and another condition is met. Give a scenario where the presence of this other condition is essential.
30. Consider the operation of protocol 6 over a 1-Mbps perfect (i.e., error-free) line. The maximum frame size is 1000 bits. New packets are generated 1 second apart. The timeout interval is 10 msec. If the special acknowledgement timer were eliminated, unnecessary timeouts would occur. How many times would the average message be transmitted?
31. In protocol 6, $MAX_SEQ = 2^n - 1$. While this condition is obviously desirable to make efficient use of header bits, we have not demonstrated that it is essential. Does the protocol work correctly for $MAX_SEQ = 4$, for example?
32. Frames of 1000 bits are sent over a 1-Mbps channel using a geostationary satellite whose propagation time from the earth is 270 msec. Acknowledgements are always piggybacked onto data frames. The headers are very short. Three-bit sequence numbers are used. What is the maximum achievable channel utilization for
 - (a) Stop-and-wait?
 - (b) Protocol 5?
 - (c) Protocol 6?
33. Compute the fraction of the bandwidth that is wasted on overhead (headers and re-transmissions) for protocol 6 on a heavily loaded 50-kbps satellite channel with data frames consisting of 40 header and 3960 data bits. Assume that the signal propagation time from the earth to the satellite is 270 msec. ACK frames never occur. NAK frames are 40 bits. The error rate for data frames is 1%, and the error rate for NAK frames is negligible. The sequence numbers are 8 bits.
34. Consider an error-free 64-kbps satellite channel used to send 512-byte data frames in one direction, with very short acknowledgements coming back the other way. What is the maximum throughput for window sizes of 1, 7, 15, and 127? The earth-satellite propagation time is 270 msec.
35. A 100-km-long cable runs at the T1 data rate. The propagation speed in the cable is $2/3$ the speed of light in vacuum. How many bits fit in the cable?
36. Give at least one reason why PPP uses byte stuffing instead of bit stuffing to prevent accidental flag bytes within the payload from causing confusion.

37. What is the minimum overhead to send an IP packet using PPP? Count only the overhead introduced by PPP itself, not the IP header overhead. What is the maximum overhead?
38. A 100-byte IP packet is transmitted over a local loop using ADSL protocol stack. How many ATM cells will be transmitted? Briefly describe their contents.
39. The goal of this lab exercise is to implement an error-detection mechanism using the standard CRC algorithm described in the text. Write two programs, *generator* and *verifier*. The *generator* program reads from standard input a line of ASCII text containing an n -bit message consisting of a string of 0s and 1s. The second line is the k -bit polynomial, also in ASCII. It outputs to standard output a line of ASCII text with $n + k$ 0s and 1s representing the message to be transmitted. Then it outputs the polynomial, just as it read it in. The verifier program reads in the output of the generator program and outputs a message indicating whether it is correct or not. Finally, write a program, *alter*, that inverts 1 bit on the first line depending on its argument (the bit number counting the leftmost bit as 1) but copies the rest of the two lines correctly. By typing

```
generator <file | verifier
```

you should see that the message is correct, but by typing

```
generator <file | alter arg | verifier
```

you should get the error message.

This page intentionally left blank

4

THE MEDIUM ACCESS CONTROL SUBLAYER

Network links can be divided into two categories: those using point-to-point connections and those using broadcast channels. We studied point-to-point links in Chap. 2; this chapter deals with broadcast links and their protocols.

In any broadcast network, the key issue is how to determine who gets to use the channel when there is competition for it. To make this point, consider a conference call in which six people, on six different telephones, are all connected so that each one can hear and talk to all the others. It is very likely that when one of them stops speaking, two or more will start talking at once, leading to chaos. In a face-to-face meeting, chaos is avoided by external means. For example, at a meeting, people raise their hands to request permission to speak. When only a single channel is available, it is much harder to determine who should go next. Many protocols for solving the problem are known. They form the contents of this chapter. In the literature, broadcast channels are sometimes referred to as **multiaccess channels** or **random access channels**.

The protocols used to determine who goes next on a multiaccess channel belong to a sublayer of the data link layer called the **MAC (Medium Access Control)** sublayer. The MAC sublayer is especially important in LANs, particularly wireless ones because wireless is naturally a broadcast channel. WANs, in contrast, use point-to-point links, except for satellite networks. Because multiaccess channels and LANs are so closely related, in this chapter we will discuss LANs in

general, including a few issues that are not strictly part of the MAC sublayer, but the main subject here will be control of the channel.

Technically, the MAC sublayer is the bottom part of the data link layer, so logically we should have studied it before examining all the point-to-point protocols in Chap. 3. Nevertheless, for most people, it is easier to understand protocols involving multiple parties after two-party protocols are well understood. For that reason we have deviated slightly from a strict bottom-up order of presentation.

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.

We will first look at the shortcomings of static allocation schemes for bursty traffic. Then, we will lay out the key assumptions used to model the dynamic schemes that we examine in the following sections.

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 we described in Sec. 2.5, such as FDM (Frequency Division Multiplexing). 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.

However, when the number of senders is large and varying or the traffic is bursty, FDM presents some problems. If the spectrum is cut up into N regions and fewer than N users are currently interested in communicating, a large piece of valuable spectrum will be wasted. And if more than N users want to communicate, some of them will be denied permission for lack of bandwidth, even if some of the users who have been assigned a frequency band hardly ever transmit or receive anything.

Even assuming that the number of users could somehow be held constant at N , dividing the single available channel into some number of static subchannels is

inherently inefficient. The basic problem is that when some users are quiescent, their bandwidth is simply lost. They are not using it, and no one else is allowed to use it either. A static allocation is a poor fit to most computer systems, in which data traffic is extremely bursty, often with peak traffic to mean traffic ratios of 1000:1. Consequently, most of the channels will be idle most of the time.

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 $1/\mu$ 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.)

In our example, if C is 100 Mbps, the mean frame length, $1/\mu$, is 10,000 bits, and the frame arrival rate, λ , is 5000 frames/sec, then $T = 200 \mu\text{sec}$. Note that if we ignored the queueing delay and just asked how long it takes to send a 10,000-bit frame on a 100-Mbps network, we would get the (incorrect) answer of 100 μsec . That result only holds when there is no contention for the channel.

Now let us divide the single channel into N independent subchannels, each with capacity C/N bps. The mean input rate on each of the subchannels will now be λ/N . Recomputing T , we get

$$T_N = \frac{1}{\mu(C/N) - (\lambda/N)} = \frac{N}{\mu C - \lambda} = NT \quad (4-1)$$

The mean delay for the divided channel is N times worse than if all the frames were somehow magically arranged orderly in a big central queue. This same result says that a bank lobby full of ATM machines is better off having a single queue feeding all the machines than a separate queue in front of each machine.

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 slot, it would just lie fallow. The same would hold if we split up the networks physically. Using our previous example again, if we were to replace the 100-Mbps network with 10 networks of 10 Mbps each and statically allocate each user to one of them, the mean delay would jump from 200 μsec to 2 msec.

Since none of the traditional static channel allocation methods work well at all with bursty traffic, we will now explore dynamic methods.

4.1.2 Assumptions for Dynamic Channel Allocation

Before we get to the first of the many channel allocation methods in this chapter, it is worthwhile to carefully formulate the allocation problem. Underlying all the work done in this area are the following five key assumptions:

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 Δt is $\lambda \Delta t$, 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, 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.

Some discussion of these assumptions is in order. The first one says that frame arrivals are independent, both across stations and at a particular station, and that frames are generated unpredictably but at a constant rate. Actually, this assumption is not a particularly good model of network traffic, as it is well known that packets come in bursts over a range of time scales (Paxson and Floyd, 1995; and Leland et al., 1994). Nonetheless, **Poisson models**, as they are frequently called, are useful because they are mathematically tractable. They help us analyze

protocols to understand roughly how performance changes over an operating range and how it compares with other designs.

The single-channel assumption is the heart of the model. No external ways to communicate exist. Stations cannot raise their hands to request that the teacher call on them, so we will have to come up with better solutions.

The remaining three assumptions depend on the engineering of the system, and we will say which assumptions hold when we examine a particular protocol.

The collision assumption is basic. Stations need some way to detect collisions if they are to retransmit frames rather than let them be lost. For wired channels, node hardware can be designed to detect collisions when they occur. The stations can then terminate their transmissions prematurely to avoid wasting capacity. This detection is much harder for wireless channels, so collisions are usually inferred after the fact by the lack of an expected acknowledgement frame. It is also possible for some frames involved in a collision to be successfully received, depending on the details of the signals and the receiving hardware. However, this situation is not the common case, so we will assume that all frames involved in a collision are lost. We will also see protocols that are designed to prevent collisions from occurring in the first place.

The reason for the two alternative assumptions about time is that slotted time can be used to improve performance. However, it requires the stations to follow a master clock or synchronize their actions with each other to divide time into discrete intervals. Hence, it is not always available. We will discuss and analyze systems with both kinds of time. For a given system, only one of them holds.

Similarly, a network may have carrier sensing or not have it. Wired networks will generally have carrier sense. Wireless networks cannot always use it effectively because not every station may be within radio range of every other station. Similarly, carrier sense will not be available in other settings in which a station cannot communicate directly with other stations, for example a cable modem in which stations must communicate via the cable headend. Note that the word “carrier” in this sense refers to a signal on the channel and has nothing to do with the common carriers (e.g., telephone companies) that date back to the days of the Pony Express.

To avoid any misunderstanding, it is worth noting that no multiaccess protocol guarantees reliable delivery. Even in the absence of collisions, the receiver may have copied some of the frame incorrectly for various reasons. Other parts of the link layer or higher layers provide reliability.

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

The story of our first MAC starts out in pristine Hawaii in the early 1970s. 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.

We will discuss two versions of ALOHA here: pure and slotted. They differ with respect to whether time is continuous, as in the pure version, or divided into discrete slots into which all frames must fit.

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.

A sketch of frame generation in an ALOHA system is given in Fig. 4-1. We have made the frames all the same length because the throughput of ALOHA systems is maximized by having a uniform frame size rather than by allowing variable-length frames.

Whenever two frames try to occupy the channel at the same time, there will be a collision (as seen in Fig. 4-1) and both will be garbled. If the first bit of a new frame overlaps with just the last bit of a frame that has almost finished, both frames will be totally destroyed (i.e., have incorrect checksums) and both will have to be retransmitted later. The checksum does not (and should not) distinguish between a total loss and a near miss. Bad is bad.

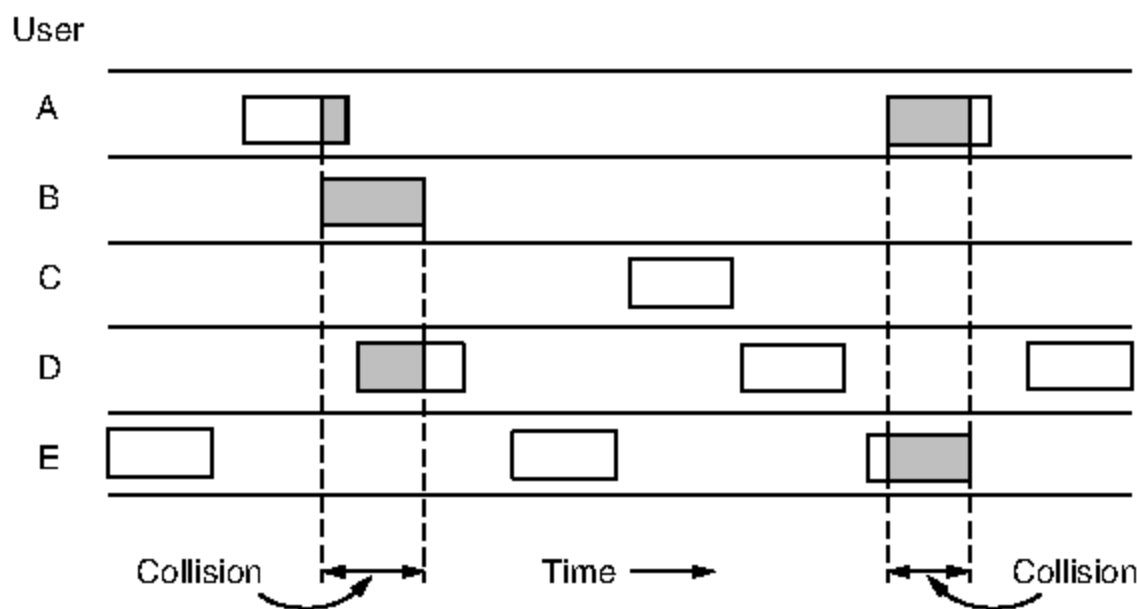


Figure 4-1. In pure ALOHA, frames are transmitted at completely arbitrary times.

An interesting question is: what is the efficiency of an ALOHA channel? In other words, what fraction of all transmitted frames escape collisions under these chaotic circumstances? Let us first consider an infinite collection of users typing at their terminals (stations). 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 to wait while the station retransmits the frame over and over until it has been successfully sent.

Let the “frame time” denote the amount of time needed to transmit the standard, fixed-length frame (i.e., the frame length divided by the bit rate). At this point, we assume that the new frames generated by the stations are well modeled by a Poisson distribution with a mean of N frames per frame time. (The infinite-population assumption is needed to ensure that N does not decrease as users become blocked.) If $N > 1$, the user community is generating frames at a higher rate than the channel can handle, and nearly every frame will suffer a collision. For reasonable throughput, we would expect $0 < N < 1$.

In addition to the new frames, the stations also generate retransmissions of frames that previously suffered collisions. Let us further assume that the old and new frames combined are well modeled by a Poisson distribution, with mean of G frames per frame time. Clearly, $G \geq N$. At low load (i.e., $N \approx 0$), there will be few collisions, hence few retransmissions, so $G \approx N$. At high load, there will be many collisions, so $G > N$. Under all loads, the throughput, S , is just the offered load, G , times the probability, P_0 , of a transmission succeeding—that is, $S = GP_0$, where P_0 is the probability that a frame does not suffer a collision.

A frame will not suffer a collision if no other frames are sent within one frame time of its start, as shown in Fig. 4-2. Under what conditions will the

shaded frame arrive undamaged? Let t be the time required to send one frame. 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.

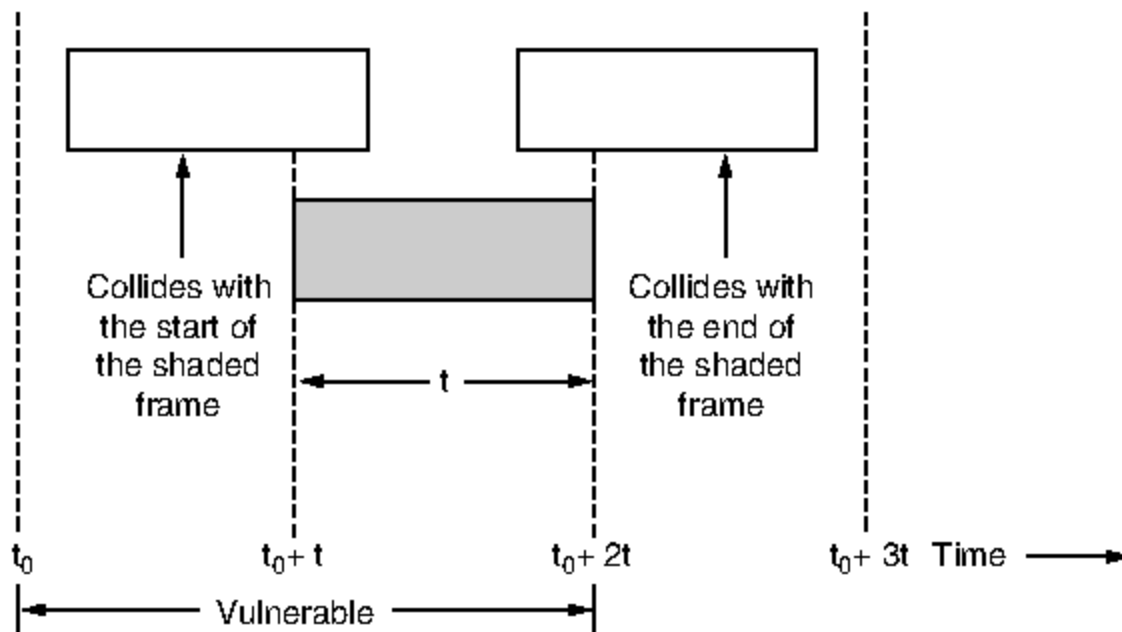


Figure 4-2. Vulnerable period for the shaded frame.

The probability that k frames are generated during a given frame time, in which G frames are expected, is given by the Poisson distribution

$$\Pr[k] = \frac{G^k e^{-G}}{k!} \quad (4-2)$$

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.

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. This

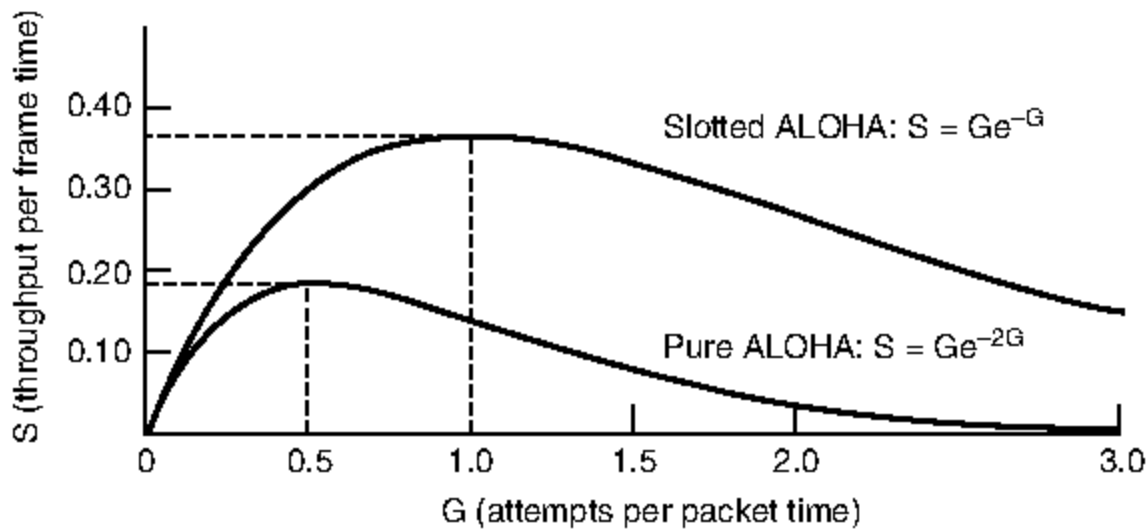


Figure 4-3. Throughput versus offered traffic for ALOHA systems.

approach requires the users to agree on slot boundaries. One way to achieve synchronization would be to have one special station emit a pip at the start of each interval, like a clock.

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 whenever 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. To see this, look at Fig. 4-3 and imagine the collisions that are now possible. The probability of no other traffic during the same slot as our test frame is then e^{-G} , which leads to

$$S = Ge^{-G} \quad (4-3)$$

As you can see from Fig. 4-3, slotted ALOHA peaks at $G = 1$, with a throughput of $S = 1/e$ or about 0.368, twice that of pure ALOHA. If the system is operating at $G = 1$, the probability of an empty slot is 0.368 (from Eq. 4-2). The best we can hope for using slotted ALOHA is 37% of the slots empty, 37% successes, and 26% collisions. Operating at higher values of G reduces the number of empties but increases the number of collisions exponentially. To see how this rapid growth of collisions with G comes about, consider the transmission of a test frame. 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 is then

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

As a result of the exponential dependence of E upon G , small increases in the channel load can drastically reduce its performance.

Slotted ALOHA is notable for a reason that may not be initially obvious. It was devised in the 1970s, used in a few early experimental systems, then almost forgotten. When Internet access over the cable was invented, all of a sudden there was a problem of how to allocate a shared channel among multiple competing users. Slotted ALOHA was pulled out of the garbage can to save the day. Later, having multiple RFID tags talk to the same RFID reader presented another variation on the same problem. Slotted ALOHA, with a dash of other ideas mixed in, again came to the rescue. It has often happened that protocols that are perfectly valid fall into disuse for political reasons (e.g., some big company wants everyone to do things its way) or due to ever-changing technology trends. Then, years later some clever person realizes that a long-discarded protocol solves his current problem. For this reason, in this chapter we will study a number of elegant protocols that are not currently in widespread use but might easily be used in future applications, provided that enough network designers are aware of them. Of course, we will also study many protocols that are in current use as well.

4.2.2 Carrier Sense Multiple Access Protocols

With slotted ALOHA, the best channel utilization that can be achieved is $1/e$. This low result is hardly surprising, since with stations transmitting at will, without knowing what the other stations are doing there are bound to be many collisions. In LANs, however, it is often possible for stations to detect what other stations are doing, and thus adapt their behavior accordingly. These networks can achieve a much better utilization than $1/e$. In this section, we will discuss some protocols for improving performance.

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. For example, see Kleinrock and Tobagi (1975). Below we will look at several versions of carrier sense protocols.

Persistent and Nonpersistent 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 station 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.

You might expect that this scheme avoids collisions except for the rare case of simultaneous sends, but in fact it does not. If two stations become ready in the middle of a third station's transmission, both will wait politely until the transmission ends, and then both will begin transmitting exactly simultaneously, resulting in a collision. If they were not so impatient, there would be fewer collisions.

More subtly, the propagation delay has an important effect on collisions. There is a chance that just after a station begins sending, another station will become ready to send and sense the channel. If the first station's signal has not yet reached the second one, the latter will sense an idle channel and will also begin sending, resulting in a collision. 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 **nonpersistent 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. Instead, it waits a random period of time and then repeats the algorithm. Consequently, this algorithm leads to better channel utilization but longer delays than 1-persistent CSMA.

The last protocol is **p-persistent CSMA**. It applies to slotted channels and works as follows. When a station becomes ready to send, it senses the channel. If it is idle, it transmits with a probability p . With a probability $q = 1 - p$, it defers until the next slot. If that slot is also idle, it either transmits or defers again, with probabilities p and q . This process is repeated until either the frame has been transmitted or another station has begun transmitting. In the latter case, the unlucky station acts as if there had been a collision (i.e., it waits a random time and starts again). If the station initially senses that the channel is busy, it waits until the next slot and applies the above algorithm. IEEE 802.11 uses a refinement of p-persistent CSMA that we will discuss in Sec. 4.4.

Figure 4-4 shows the computed throughput versus offered traffic for all three protocols, as well as for pure and slotted ALOHA.

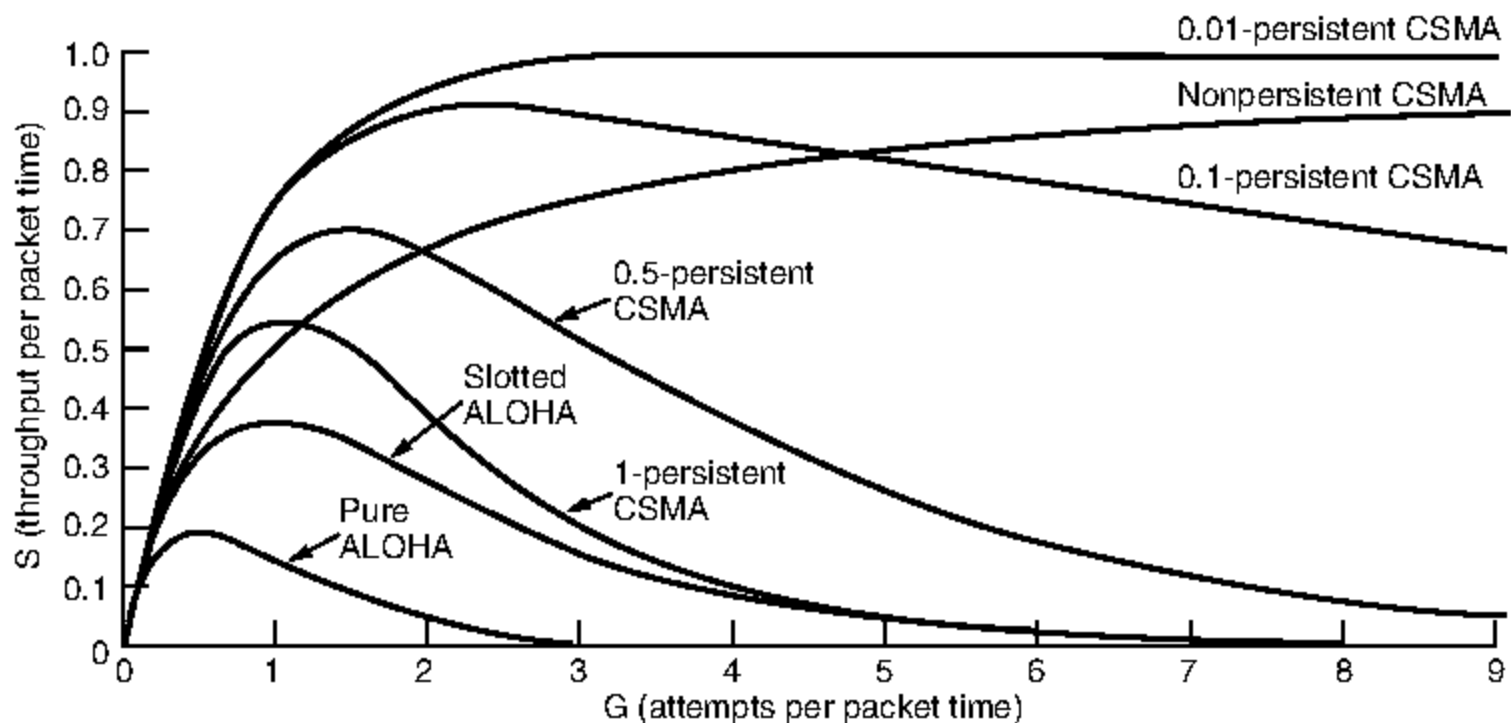


Figure 4-4. Comparison of the channel utilization versus load for various random access protocols.

CSMA with Collision Detection

Persistent and nonpersistent 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. The implications are that a received signal must not be tiny compared to the transmitted signal (which is difficult for wireless, as received signals may be 1,000,000 times weaker than transmitted signals) and that the modulation must be chosen to allow collisions to be detected (e.g., a collision of two 0-volt signals may well be impossible to detect).

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 (assuming that no other station has started transmitting in the

meantime). 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).

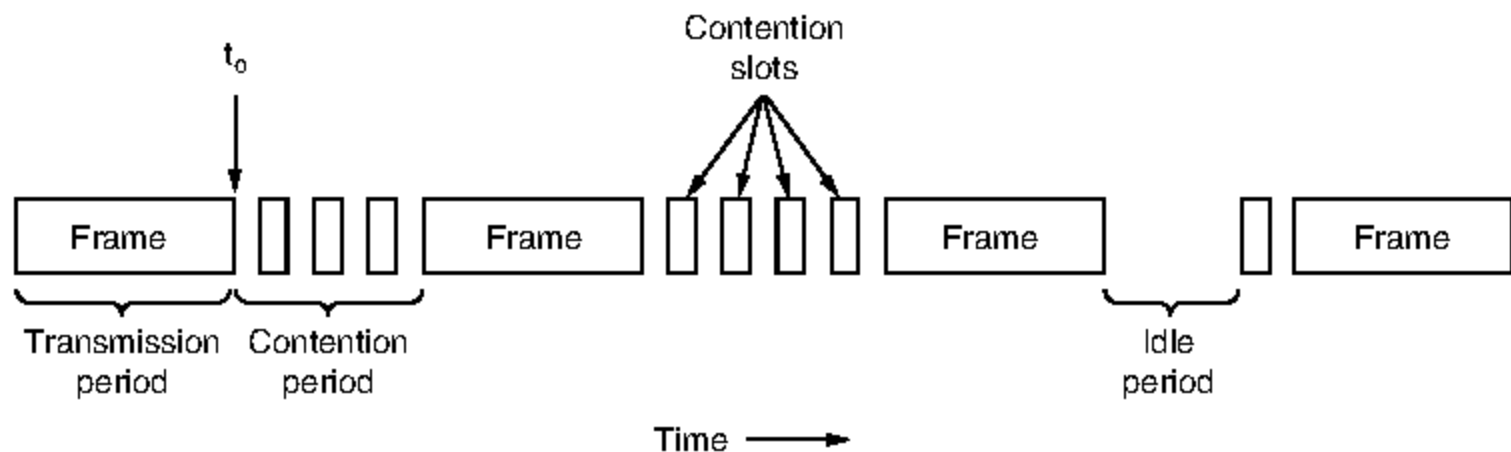


Figure 4-5. CSMA/CD can be in contention, transmission, or idle state.

Now let us look at the details of the contention algorithm. Suppose that two stations both begin transmitting at exactly time t_0 . How long will it take them to realize that they have collided? The answer is vital to determining the length of the contention period and hence what the delay and throughput will be.

The minimum time to detect the collision is just the time it takes the signal to propagate from one station to the other. Based on this information, you might think that a station that has not heard a collision for a time equal to the full cable propagation time after starting its transmission can be sure it has seized the cable. By “seized,” we mean that all other stations know it is transmitting and will not interfere. This conclusion is wrong.

Consider the following worst-case scenario. Let the time for a signal to propagate between the two farthest stations be τ . At t_0 , one station begins transmitting. At $t_0 + \tau - \epsilon$, an instant before the signal arrives at the most distant station, that station also begins transmitting. Of course, it detects the collision almost instantly and stops, but the little noise burst caused by the collision does not get back to the original station until time $2\tau - \epsilon$. In other words, in the worst case a station cannot be sure that it has seized the channel until it has transmitted for 2τ without hearing a collision.

With this understanding, we can think of CSMA/CD contention as a slotted ALOHA system with a slot width of 2τ . On a 1-km long coaxial cable, $\tau \approx 5 \mu\text{sec}$. The difference for CSMA/CD compared to slotted ALOHA is that slots in which only one station transmits (i.e., in which the channel is seized) are followed by the rest of a frame. This difference will greatly improve performance if the frame time is much longer than the propagation time.

4.2.3 Collision-Free Protocols

Although collisions do not occur with CSMA/CD once a station has unambiguously captured the channel, they can still occur during the contention period. These collisions adversely affect the system performance, especially when the

bandwidth-delay product is large, such as when the cable is long (i.e., large τ) and the frames are short. Not only do collisions reduce bandwidth, but they make the time to send a frame variable, which is not a good fit for real-time traffic such as voice over IP. CSMA/CD is also not universally applicable.

In this section, we will examine some protocols that resolve the contention for the channel without any collisions at all, not even during the contention period. 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.

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. The basic question remains: which station gets the channel after a successful transmission? We continue using the model of Fig. 4-5 with its discrete contention slots.

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. In general, station j may announce that it has a frame to send by inserting a 1 bit into slot j . After all N slots have passed by, each station has complete knowledge of which stations wish to transmit. At that point, they begin transmitting frames in numerical order (see Fig. 4-6).

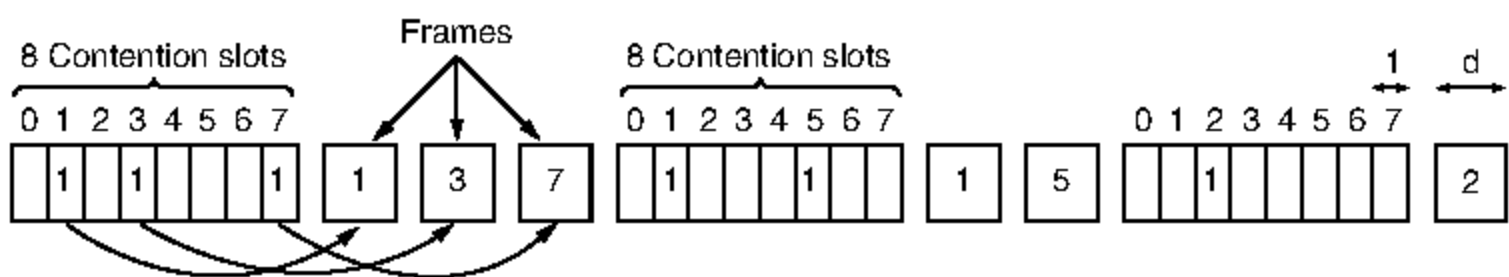


Figure 4-6. The basic bit-map protocol.

Since everyone agrees on who goes next, there will never be any collisions. After the last ready station has transmitted its frame, an event all stations can easily monitor, another N -bit contention period is begun. If a station becomes ready just after its bit slot has passed by, it is out of luck and must remain silent until every station has had a chance and the bit map has come around again.

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.

Under conditions of low load, the bit map will simply be repeated over and over, for lack of data frames. Consider the situation from the point of view of a low-numbered station, such as 0 or 1. Typically, when it becomes ready to send, the “current” slot will be somewhere in the middle of the bit map. On average, the station will have to wait $N/2$ slots for the current scan to finish and another full N slots for the following scan to run to completion before it may begin transmitting.

The prospects for high-numbered stations are brighter. Generally, these will only have to wait half a scan ($N/2$ bit slots) before starting to transmit. High-numbered stations rarely have to wait for the next scan. Since low-numbered stations must wait on average $1.5N$ slots and high-numbered stations must wait on average $0.5N$ slots, the mean for all stations is N slots.

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 $d/(d + N)$.

At high load, when all the stations have something to send all the time, the N -bit contention period is prorated over N frames, yielding an overhead of only 1 bit per frame, or an efficiency of $d/(d + 1)$. The mean delay for a frame is equal to the sum of the time it queues inside its station, plus an additional $(N - 1)d + N$ once it gets to the head of its internal queue. This interval is how long it takes to wait for all other stations to have their turn sending a frame and another bitmap.

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. Passing the token to the next station then simply consists of receiving the token in from one direction and transmitting it out in the other direction, as seen in Fig. 4-7. Frames are also transmitted in the direction of the token. This way they will circulate around the ring and reach whichever station is the destination. However, to stop the frame circulating indefinitely (like the token), some station needs

to remove it from the ring. This station may be either the one that originally sent the frame, after it has gone through a complete cycle, or the station that was the intended recipient of the frame.

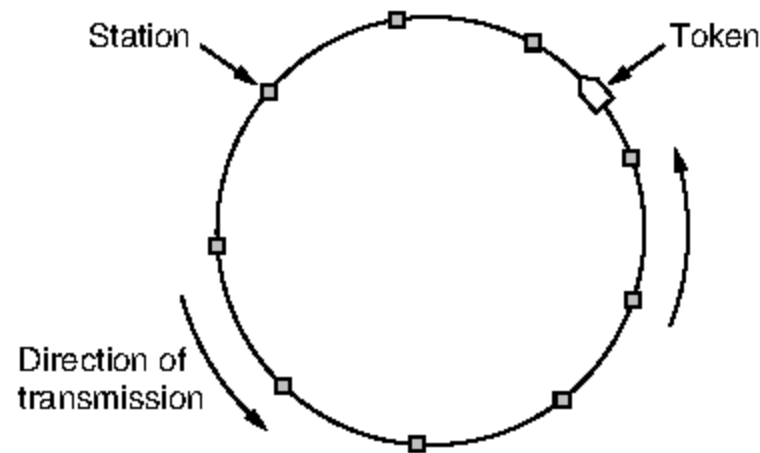


Figure 4-7. Token ring.

Note that we do not need a physical ring to implement token passing. 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**.

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. After sending a frame, each station must wait for all N stations (including itself) to send the token to their neighbors and the other $N - 1$ stations to send a frame, if they have one. A subtle difference is that, since all positions in the cycle are equivalent, there is no bias for low- or high-numbered stations. For token ring, each station is also sending the token only as far as its neighboring station before the protocol takes the next step. Each token does not need to propagate to all stations before the protocol advances to the next step.

Token rings have cropped up as MAC protocols with some consistency. An early token ring protocol (called “Token Ring” and standardized as IEEE 802.5) was popular in the 1980s as an alternative to classic Ethernet. In the 1990s, a much faster token ring called **FDDI (Fiber Distributed Data Interface)** was beaten out by switched Ethernet. In the 2000s, a token ring called **RPR (Resilient Packet Ring)** was defined as IEEE 802.17 to standardize the mix of metropolitan area rings in use by ISPs. We wonder what the 2010s will have to offer.

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. A station wanting to use the

channel now broadcasts its address as a binary bit string, starting with the high-order bit. All addresses are assumed to be the same length. The bits in each address position from different stations are **BOOLEAN ORed** together by the channel when they are sent at the same time. We will call this protocol **binary countdown**. It was used in Datakit (Fraser, 1987). 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. For example, if stations 0010, 0100, 1001, and 1010 are all trying to get the channel, in the first bit time the stations transmit 0, 0, 1, and 1, respectively. These are ORed together to form a 1. Stations 0010 and 0100 see the 1 and know that a higher-numbered station is competing for the channel, so they give up for the current round. Stations 1001 and 1010 continue.

The next bit is 0, and both stations continue. The next bit is 1, so station 1001 gives up. The winner is station 1010 because it has the highest address. After winning the bidding, it may now transmit a frame, after which another bidding cycle starts. 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.

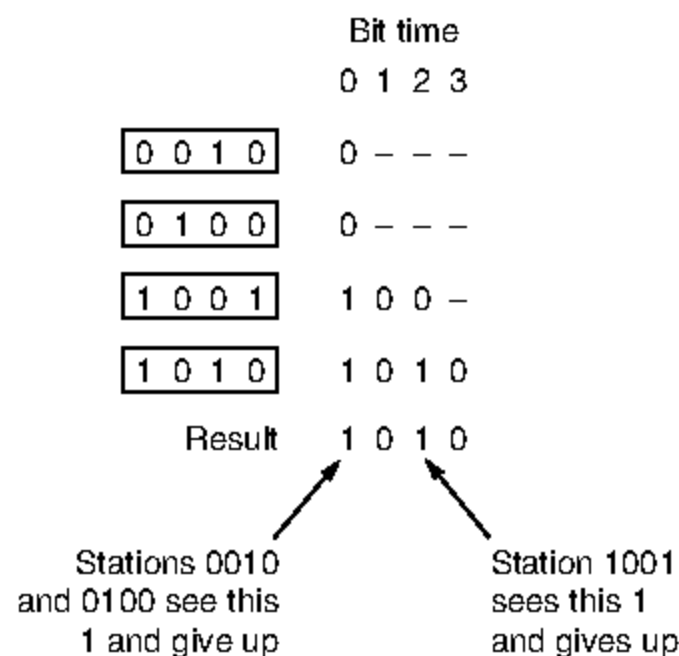


Figure 4-8. The binary countdown protocol. A dash indicates silence.

The channel efficiency of this method is $d/(d + \log_2 N)$. If, however, the frame format has been cleverly chosen so that the sender's address is the first field in the frame, even these $\log_2 N$ bits are not wasted, and the efficiency is 100%.

Binary countdown is an example of a simple, elegant, and efficient protocol that is waiting to be rediscovered. Hopefully, it will find a new home some day.

4.2.4 Limited-Contention Protocols

We have now considered two basic strategies for channel acquisition in a broadcast network: contention, as in CSMA, and collision-free 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. Just the reverse is true for the collision-free protocols. At low load, they have relatively high delay but as the load increases, the channel efficiency improves (since the overheads are fixed).

Obviously, it would be nice if we could combine the best properties of the contention and collision-free protocols, arriving at a new protocol that used contention at low load to provide low delay, but used a collision-free technique at high load to provide good channel efficiency. Such protocols, which we will call **limited-contention protocols**, do in fact exist, and will conclude our study of carrier sense networks.

Up to now, the only contention protocols we have studied have been symmetric. That is, each station attempts to acquire the channel with some probability, p , with all stations using the same p . Interestingly enough, the overall system performance can sometimes be improved by using a protocol that assigns different probabilities to different stations.

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

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

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 $1/e$.

From Fig. 4-9, it is fairly obvious that the probability of some station acquiring the channel can be increased only by decreasing the amount of competition. The limited-contention protocols do precisely that. They first divide the stations into (not necessarily disjoint) groups. Only the members of group 0 are permitted

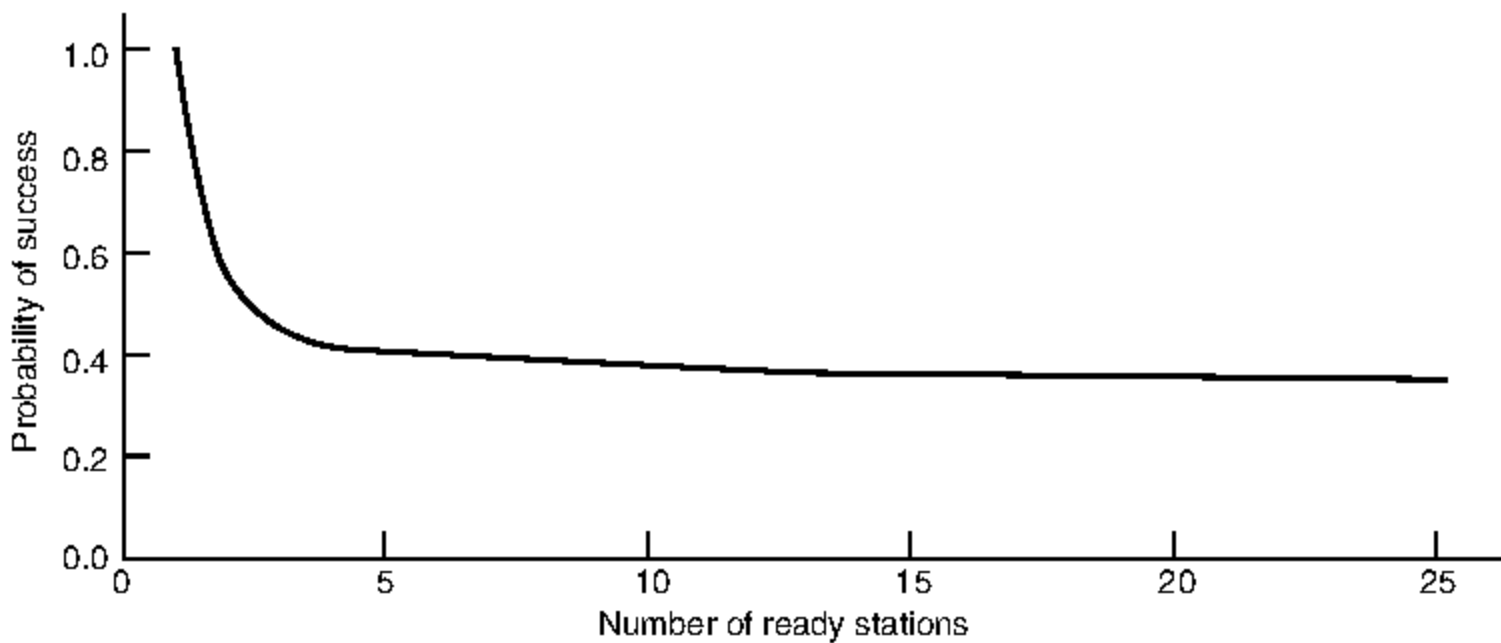


Figure 4-9. Acquisition probability for a symmetric contention channel.

to compete for slot 0. If one of them succeeds, it acquires the channel and transmits its frame. If the slot lies fallow or if there is a collision, the members of group 1 contend for slot 1, etc. By making an appropriate division of stations into groups, the amount of contention for each slot can be reduced, thus operating each slot near the left end of Fig. 4-9.

The trick is how to assign stations to slots. Before looking at the general case, let us consider some special cases. At one extreme, each group has but one member. Such an assignment guarantees that there will never be collisions because at most one station is contending for any given slot. We have seen such protocols before (e.g., binary countdown). The next special case is to assign two stations per group. The probability that both will try to transmit during a slot is p^2 , which for a small p is negligible. As more and more stations are assigned to the same slot, the probability of a collision grows, but the length of the bit-map scan needed to give everyone a chance shrinks. The limiting case is a single group containing all stations (i.e., slotted ALOHA). What we need is a way to assign stations to slots dynamically, with many stations per slot when the load is low and few (or even just one) station per slot when the load is high.

The Adaptive Tree Walk Protocol

One particularly simple way of performing the necessary assignment is to use the algorithm devised by the U.S. Army for testing soldiers for syphilis during World War II (Dorfman, 1943). In short, the Army took a blood sample from N soldiers. A portion of each sample was poured into a single test tube. 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 process was repeated recursively until the infected soldiers were determined.

For the computerized version of this algorithm (Capetanakis, 1979), it is convenient to think of the stations as the leaves of a binary tree, as illustrated in Fig. 4-10. In the first contention slot following a successful frame transmission, slot 0, all stations are permitted to try to acquire the channel. If one of them does so, fine. If there is a collision, then during slot 1 only those stations falling under node 2 in the tree may compete. If one of them acquires the channel, the slot following the frame is reserved for those stations under node 3. 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.

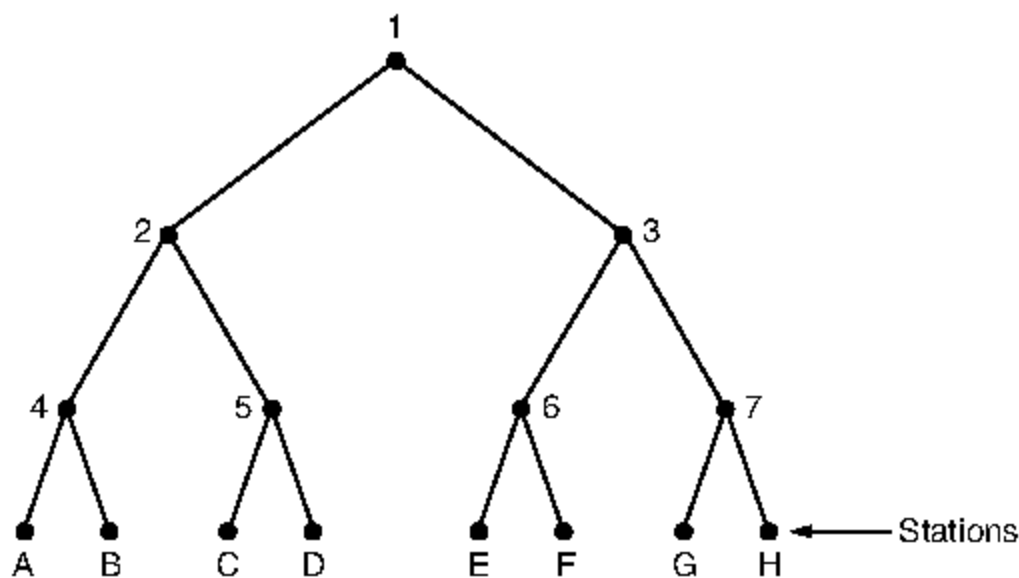


Figure 4-10. The tree for eight stations.

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. Similarly, one could argue that nodes 2 and 3 should be skipped as well for the same reason. Put in more general terms, at what level in the tree should the search begin? Clearly, the heavier the load, the farther down the tree the search should begin. We will assume that each station has a good estimate of the number of ready stations, q , for example, from monitoring recent traffic.

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