Pearson New International Edition

Computer Networks

Andrew S. Tanenbaum David J. Wetherall

Fifth Edition

Pearson New International Edition

Computer Networks
Andrew S. Tanenbaum David J. Wetherall
Fifth Edition

PEARSON

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.

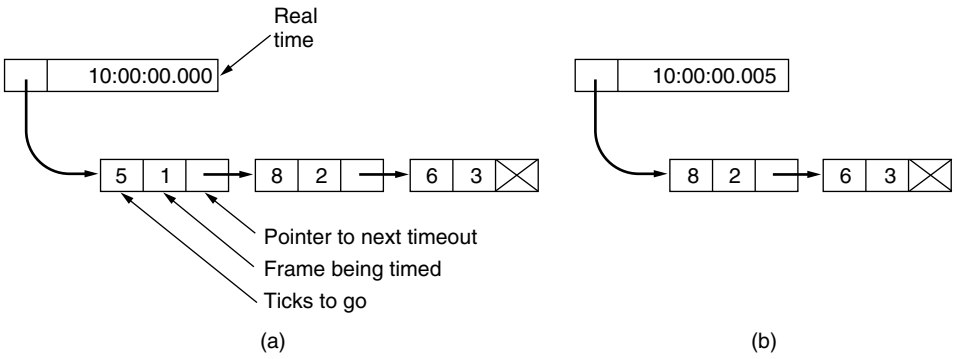

Figure 20. Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

As an illustration of how the timers could be implemented, consider the example of Fig. 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. 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.

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. 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. 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. 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. 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 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. 22(c) and Fig. 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 mod 4. Notice that although i and $(i + 4)$ mod 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

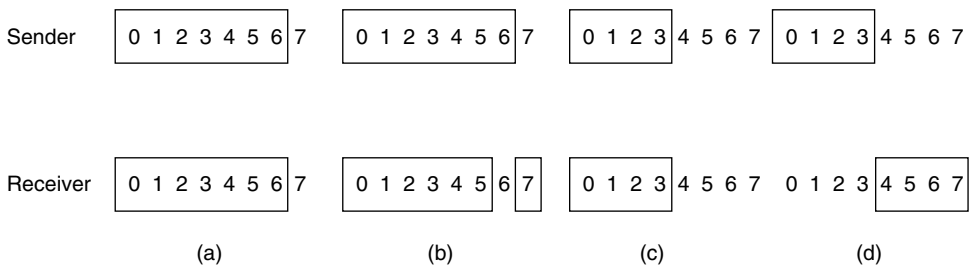

Figure 22. (a) Initial situation with a window of size7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

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.