Chapter 3
(Week 6)

The Data Link Layer (CONTINUATION)

ANDREW S. TANENBAUM
COMPUTER NETWORKS
FOURTH EDITION
PP. 211-246

Elementary Data Link Protocols (1/14)

- Three protocols of increasing complexity:
- A simulator for protocols is available via Web http://www.prenhall.com/tanenbaum
- ASSUMPTION 1:
- In the physical layer (PhL), data link layer (DLL), and network layer (NL) are independent processes that comunicate by passing messages back and forth.
- In many cases, PhL and DLL processes will be running on a processor inside a special network I/O chip and NL code will be running on the main CPU.

Elementary Data Link Protocols (2/14)

- However, other implementations are also possible:
- three processes inside a single I/O chip;
- or PhL and DLL as procedures called by NL process.
- In any event, treating the three layers as separate processes makes the discussion conceptually cleaner and also serves to emphasize the independence of the layers.

Elementary Data Link Protocols (3/14)

ASSUMPTION 2:

- Machine A wants to send a long stream of data to machine B, using reliable, connection-oriented service.
- Later, we will consider the case where B also wants to send data to A simultaneously.
- A is assumed to have an infinite supply of data ready to send and never has to wait for data to be produced.
- Instead, when A's DLL asks for data, NL is always able to comply immediately. This restriction will be dropped later.

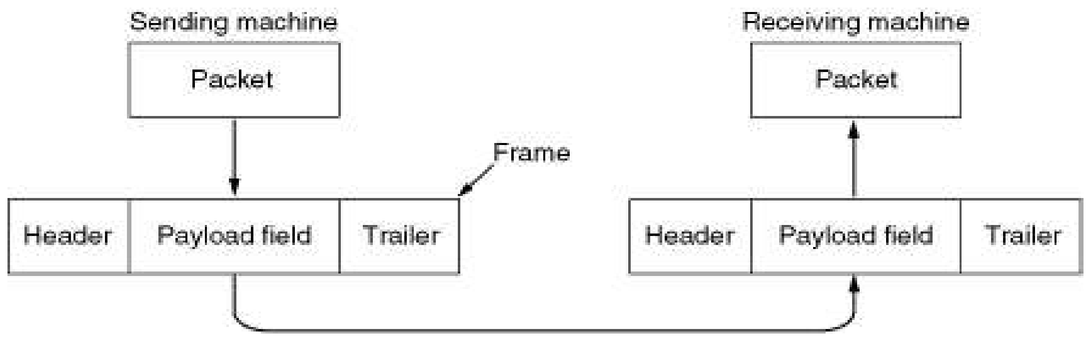
Elementary Data Link Protocols (4/14)

ASSUMPTION 3:

- Machines do not crash. That is these protocols deal with communication errors, but not the problems caused by computers crashing and rebooting.
- The packet passed across the interface to DLL from NL is pure data, whose every bit is to be delivered to the destination's NL.
- The fact that the destination's NL may interpret part of the packet as a header is of no concern to DLL.

Elementary Data Link Protocols (5/14)

- When DLL accepts a packet, it encapsulates the packet in a frame by adding a data link header and trailer to it.



- Thus, a frame consists of an embedded packet, some control information (in the header), and a checksum (in the trailer).
- The frame is then transmitted to DLL on the other machine.

Elementary Data Link Protocols (6/14)

- Library procedures:
- *to_physical_layer* is for sending a frame
- *from_physical_layer* is for receiving a frame
- The transmitting hardware computes and appends the checksum (thus creating the trailer), so that DLL software need not worry about it.
- The polynomial algorithm discussed earlier in this chapter might be used, for example.

Elementary Data Link Protocols (7/14)

- Initially, the receiver just sits around waiting for something to happen (e.g., a frame has arrived).
- *wait_for_event(&event)* is for receiver to act
- Variable *event* tells what happened.
- For example, *event=cksum_err* means that the checksum is incorrect, there was a transmission error.
- *event=frame_arrival* means the inbound frame arrived undamaged.
- The set of possible events differs for the various protocols.

Elementary Data Link Protocols (8/14)

- Following slide shows some declarations (in C) common to many of protocols to be discussed later.
- Five data structures are defined there:

a) *boolean* – is an enumerated type and can take on the values true and false.

b) *seq_nr* is a small integer (0 - MAX_SEQ) used to number the frames so that we can tell them apart.

c) *packet* is the unit of information exchanged between NL and DLL on the same machine, or between NL peers. In our model *packet* contains MAX_PKT bytes, but may be of variable length.

Elementary Data Link Protocols (9/14)

d) frame_kind is wether there are any data in frame, because some of protocols distinguish frames containing only control information from those containing data as well.

c) frame is composed of four fields: *kind*, *seq*, *ack*, and *info*. The first three contain control information. These control fields are collectively called the frame header. A last one may contain actual data to be transferred.

- *kind* – whether there are any data in the frame.
- *seq* – for sequence numbers
- *ack* – for acknowledgements
- *info* – in data frame it contains a single packet

A number of procedures are also listed in figure.

Elementary Data Link Protocols (10/14)
Protocol Definitions

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

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

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

Continued →

Some definitions needed in the protocols to follow.
These are located in the file protocol.h.

Protocol Definitions (ctd.) (11/14)

Some definitions needed in the protocols to follow. These are located in the file protocol.h.

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

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

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

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

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

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

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

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

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

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

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

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

Elementary Data Link Protocols (12/14)

- In most of the protocols, we assume that the channel is unreliable and loses entire frames upon occasion.
- To be able to recover from such calamities, the sending DLL must start an internal or clock whenever it sends a frame.
- If no reply has been received within a certain predetermined time interval, the clock times out and DLL receives an interrupt signal.

Elementary Data Link Protocols (13/14)

- In our protocols this is handled by allowing the procedure *wait_for_event* to return *event=timeout*.
- The procedures *start_timer* and *stop_timer* turn the timer on and off, respectively.
- The procedures *start_ack_timer* and *stop_ack_timer* control an auxiliary timer used to generate acknowledgements under certain conditions.

Elementary Data Link Protocols (14/14)

- PROTOCOL 1: An Unrestricted Simplex Protocol
- PROTOCOL 2: A Simplex Stop-and-Wait Protocol
- PROTOCOL 3: A Simplex Protocol for a Noisy Channel

PROTOCOL 1: Unrestricted Simplex Protocol (1/2)

- Data are transmitted in one direction only.
- Both the transmitting and receiving network layers are always ready.
- Processing time can be ignored.
- Infinite buffer space is available.
- The communication channel between the data link layers never damages or loses frames.
- This is thoroughly unrealistic protocol and we nickname it as “utopia”.

Unrestricted Simplex Protocol (2/2)

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

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

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

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

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

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

PROTOCOL 2: Simplex Stop-and-Wait Protocol (1/3)

- Now we will drop the most unrealistic restriction used in Protocol 1: the ability of the receiving NL to process incoming data infinitely quickly.
- The communication channel is still assumed to be error free however, and the data traffic is still simplex.
- The main problem we have to deal with here is how to prevent the sender from flooding the receiver with data faster than the latter is able to process them.

Simplex Stop-and-Wait Protocol 2/2

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

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

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

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

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

Simplex Stop-and-Wait Protocol (3/3)

- If the recever requires a time Δt to execute *from_physical_layer* plus *to_network_layer*, the sender must transmit at an average rate less than one frame per time Δt.
- If we assume that no automatic buffering and queuing are done within the receiver's hardware, the sender must never transmit a new frame until the old one has been fetched by *from_physical_layer*, let the new one overwrite the old one.
- Protocols in which the sender sends one frame and then waits for an acknowledgement before proceeding are called stop-and-wait.

PROTOCOL 3: A Simplex Protocol for a Noisy Channel (1/8)

- Frames may be either damaged or lost completely.
- However, we assume that if a frame is damaged in transit, the receiver hardware will detect this when it computes the checksum.
- If the frame is damaged in such a way that the checksum is nevertheless correct, an unlikely occurrence, this protocol (and all other protocols) can fail (i.e., deliver an incorrect packet to the network layer).

A Simplex Protocol for a Noisy Channel (2/8)

- A simple solution is to change Protocol 1 by adding timer such a way:
- The sender could send a frame, but the receiver would only send an acknowledgement frame if the data were correctly received.
- If a damaged frame arrived at the receiver, it would be discarded.
- After a while the sender would time out and send the frame again.
- This process would be repeated until the frame finally arrived intact.

A Simplex Protocol for a Noisy Channel (3/8)

- The above scheme has a fatal flaw in it.
- The task of DLL processes is to provide error-free, transparent communication between NL processes.
- NL on machine A gives a series of packets to its DLL, which must ensure that an identical series of packets are delivered to NL on machine B by its DLL.
- In particular, NL on B has no way of knowing that a packet has been lost or duplicated, so DLL must guarantee that no combination of transmission errors, however unlikely, can cause a duplicate packet to be delivered to NL.

A Simplex Protocol for a Noisy Channel (4/8)

Consider the following scenario:

1) NL on A gives packet 1 to its DLL. The packet is correctly received at B and passed to NL on B. B sends an acknowledgement frame back to A.
2) The acknowledgement frame gets lost completely. It just never arrives at all. Not only data frames but also control frames may be lost.
3) DLL on A eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frames was lost or damaged and sends the frame containing packet 1 again.

A Simplex Protocol for a Noisy Channel (5/8)

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

One solution: By sequence number in the header of each frame the receiver can check if it is a new frame or a duplicate to be discarded.

A Simplex Protocol for a Noisy Channel (6/8)

A positive acknowledgement with retransmission protocol.

```
/* 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 */
          }
     }
 }
}
```

Continued →

A Simplex Protocol for a Noisy Channel (ctd.) (7/8)

```
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 */
       }
  }
}
```

A positive acknowledgement with retransmission protocol.

A Simplex Protocol for a Noisy Channel (8/8)

- Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called PAR (Positive Acknowledgement with Retransmission) or ARQ (Automatic Repeat reQuest).
- Like protocol 2, this one also transmits data only in one direction.
- Protocol 3 differs from Protocol 2 in that both sender and receiver have a variable whose value is remembered while DLL is in the wait state.

Sliding Window Protocols (1/17)

- In the previous protocols, data frames were transmitted in one direction only.
- In most practical situations, there is a need for transmitting data in both directions.

Sliding Window Protocols (2/17)

- One way of achieving full-duplex data transmission is to have two separate communication channels and use each one for simplex data traffic (in different directions).
- If this is done, we have two separate physical circuits, each with a "forward" channel (for data) and a "reverse" channel (for acknowledgements).

Sliding Window Protocols (3/17)

- In both cases the bandwidth of the reverse channel is almost entirely wasted.
- In effect, the user is paying for two circuits but using only the capacity of one.

Sliding Window Protocols (4/17)

- Another way of achieving full-duplex data transmission is to use the same circuit for data in both directions.
- In protocols 2 and 3 “forward” and “reverse” channels were used and both of them have the same capacity.

Sliding Window Protocols (5/17)

- Now we will discuss the model in which the data frames from A to B are intermixed with the achnowledgement frames from A to B.
- By looking at the kind field in the header of an incoming frame, the receiver can tell whether the frame is data or acknowledgement.

Sliding Window Protocols (6/17) Piggybacking (1)

- Although interleaving data and control frames on the same circuit is an improvement over having two separate physical circuits, yet another improvement is possible.
- When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the network layer passes it the next packet.

Sliding Window Protocols (7/17)
Piggybacking (2)

- The ACK is attached to the outgoing data frame (using the ACK field in the frame header).
- In effect, ACK gets free ride on the next outgoing data frame.
- The technique of temporarily delaying outgoing ACK so that they can be hooked onto the next outgoing data frame is known as Piggybacking.

Sliding Window Protocols (8/17) Piggybacking (3)

- The principal advantage of using piggybacking over having distinct ACK frames is a better use of the available channel bandwidth.
- The ACK field in the frame header costs only a few bits, whereas a separate frame would need a header, the ACK, and a checksum.
- In addition, fewer frames sent means fewer “frame arrival” interrupts, and perhaps fewer buffers in the receiver, depending on how the receiver’s software is organized.

Sliding Window Protocols (9/17) Piggybacking (4)

- In the next protocol to be examined, the piggyback field costs only 1 bit in the frame header.
- It rarely costs more than a few bits.
- However, piggybacking introduces a complication not present with separate ACSs.
- How long should the data link layer wait for a packet onto which to piggyback the acknowledgement?

Sliding Window Protocols (10/17) Piggybacking (5)

- If the data link layer waits longer than the sender's timeout period, the frame will be retransmitted, defeating the whole purpose of having ACKs.
- 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 ACK immediately, depending on how long the projected wait was going to be.

Sliding Window Protocols (11/17) Piggybacking (6)

- 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 ACK is piggybacked onto it; otherwise, if no packet has arrived by the end of this time period, the data link layer just sends a separate ACK frame.

Sliding Window Protocols (12/17)

- The next three protocols are bidirectional protocols that belong to a class called sliding window protocols.

Sliding Window Protocols (13/17)

- PROTOCOL 4: A One-Bit Sliding Window Protocol
- PROTOCOL 5: A Protocol Using Go Back N
- PROTOCOL 6: A Protocol Using Selective Repeat

Sliding Window Protocols (14/17)

- The three differ among themselves in term of efficiency, complexity, and buffer requirements.
- 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 number to 0 and 1.

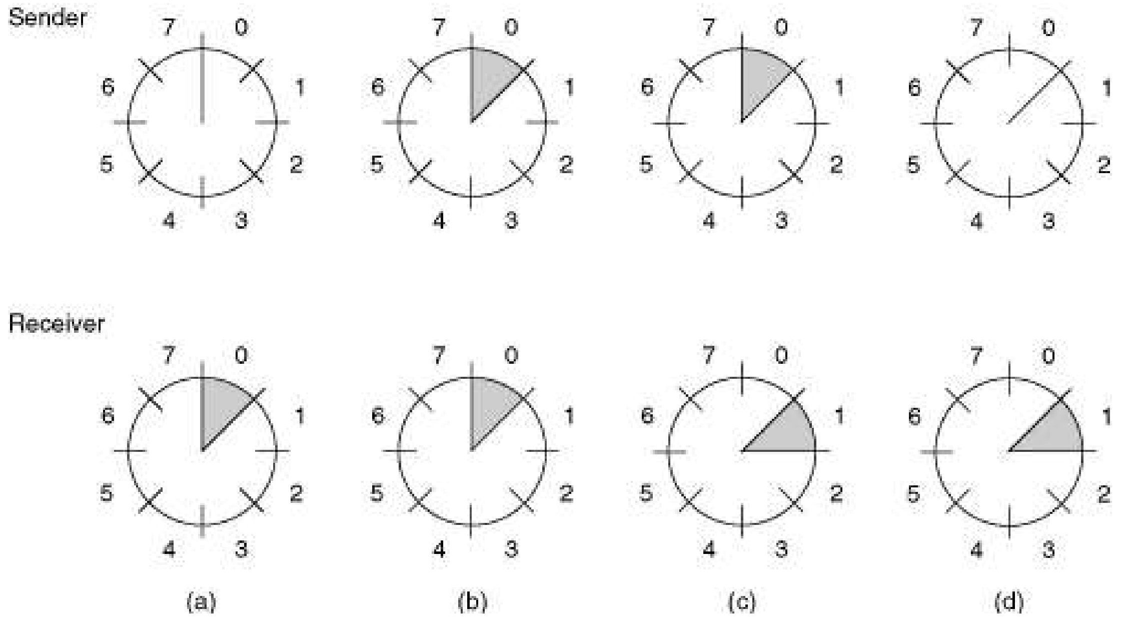
Sliding Window Protocols (15/17)

- 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.

Sliding Window Protocols (16/17)

- 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.

Sliding Window Protocols (17/17)



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.

PROTOCOL 4: A One-Bit Sliding Window Protocol (1/3)

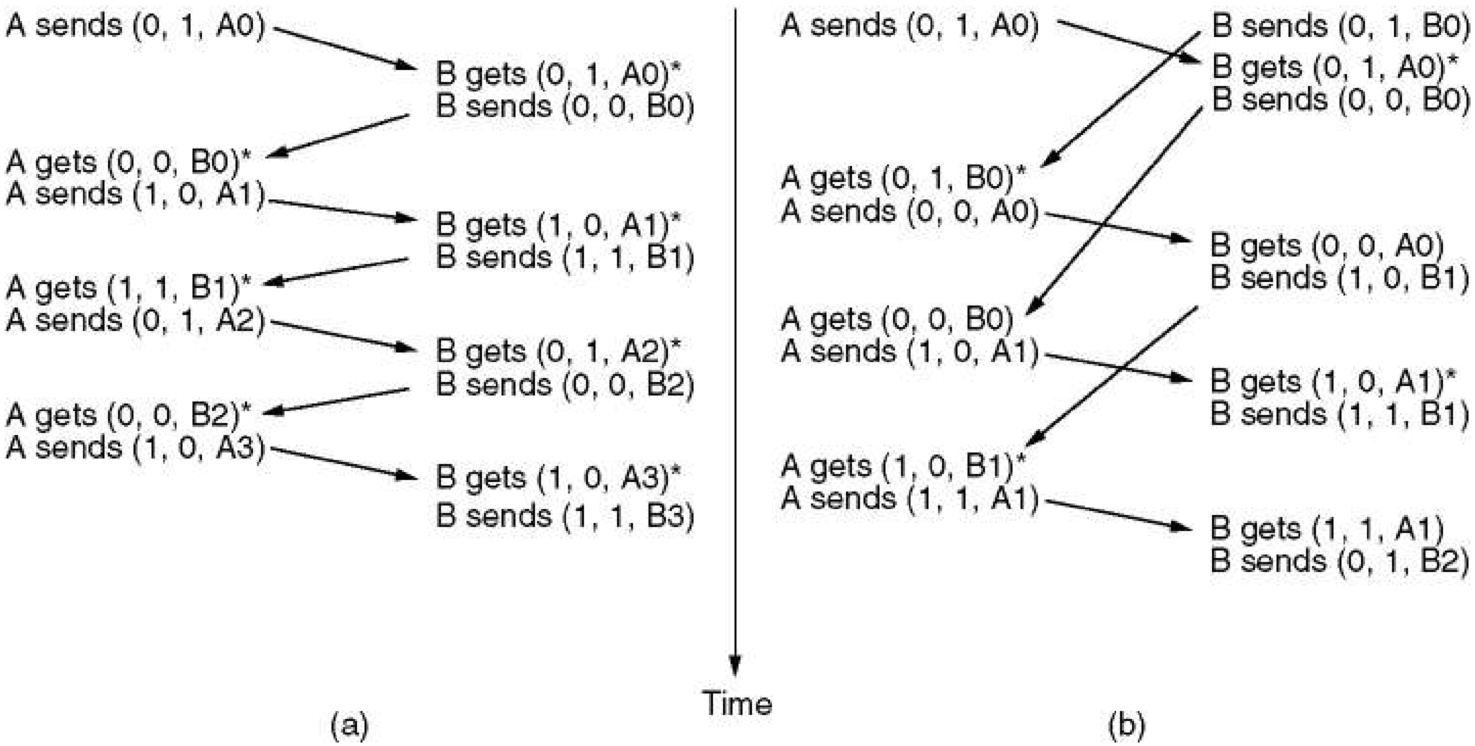
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                         /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;              /* 0 or 1 only */
  seq_nr 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 */
```


Continued →

A One-Bit Sliding Window Protocol (ctd.) (2/3)

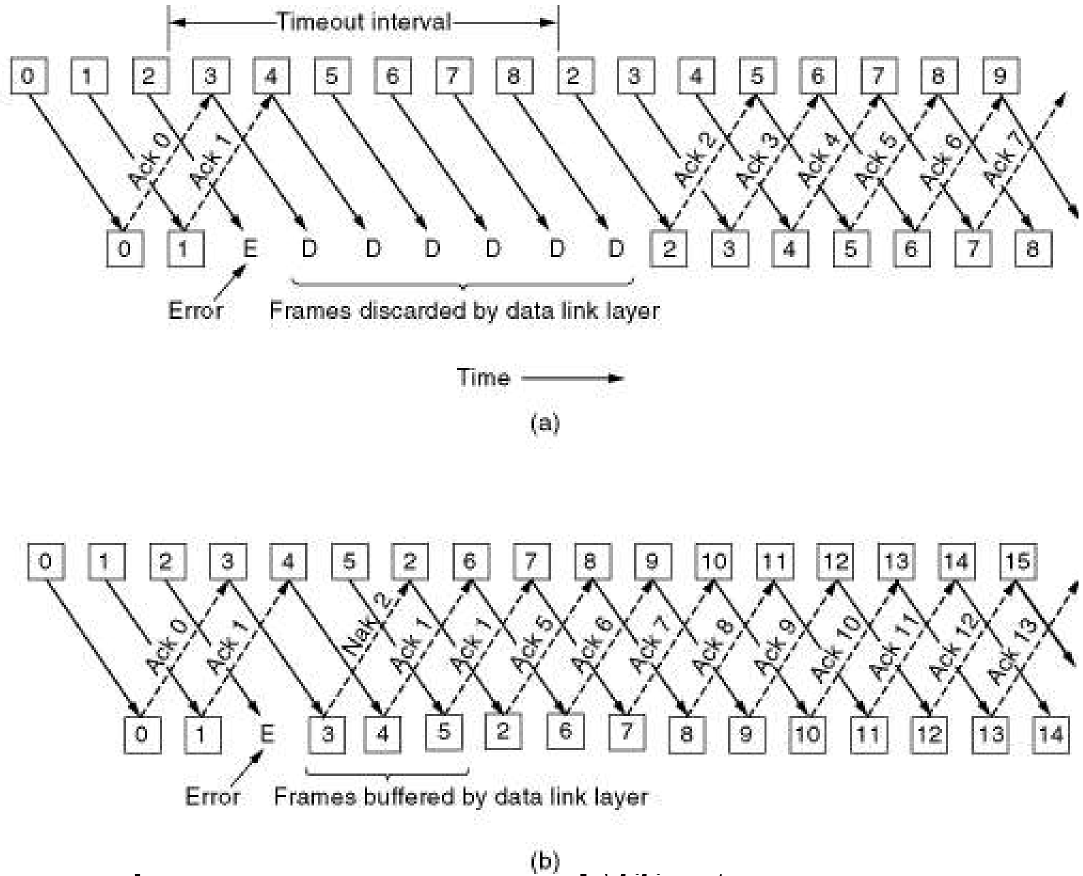
```
   while (true) {
     wait_for_event(&event);              /* frame_arrival, cksum_err, or timeout */
     if (event == frame_arrival) {        /* a frame has arrived undamaged. */
          from_physical_layer(&r);        /* go get it */

          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 senderís 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 */
   }
}
```

A One-Bit Sliding Window Protocol (3/3)



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.

PROTOCOL 5: A Protocol Using Go Back N (1/6)



Pipelining and error recovery. Effect on an error when

(a) Receiver's window size is 1.

(b) Receiver's window size is large.

Sliding Window Protocol Using Go Back N (2/6)

```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike 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                        /* should be 2^n – 1 */
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 */
}
```


Continued →

Sliding Window Protocol Using Go Back N (3/6)

```
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;                     /* # 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 */
```


Continued →

Sliding Window Protocol Using Go Back N (4/6)

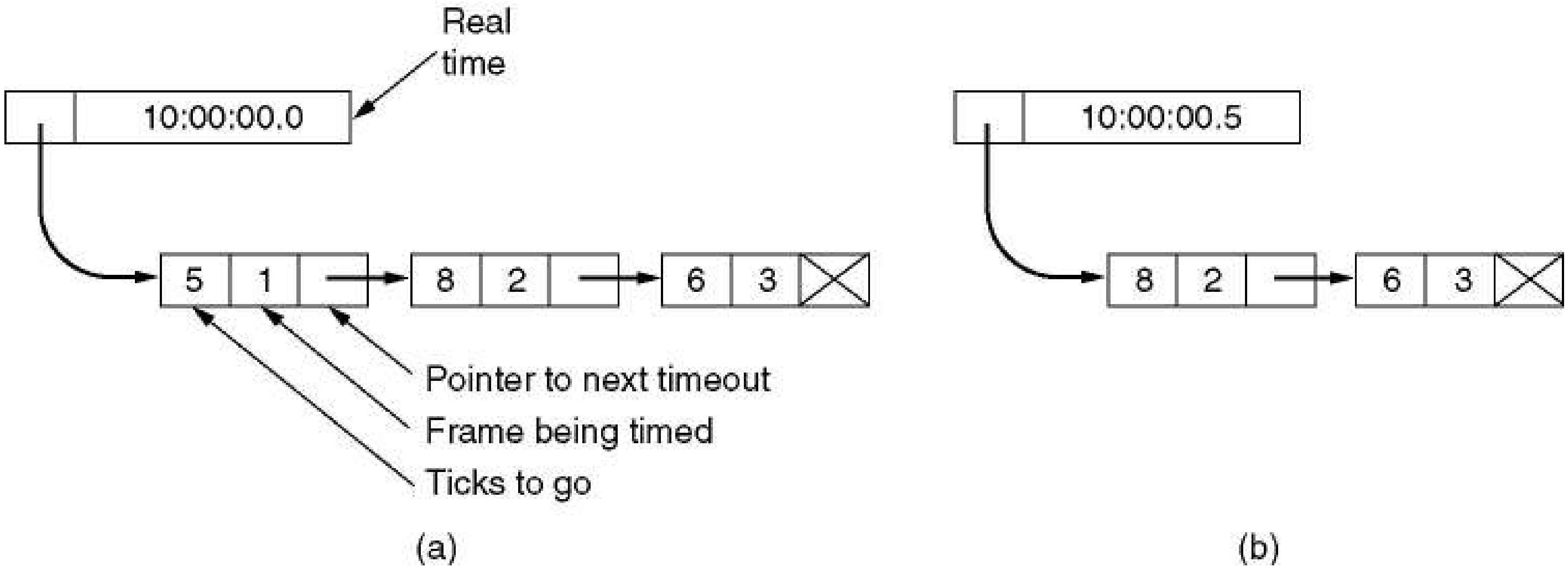
```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  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 */
          }
```


Continued →

Sliding Window Protocol Using Go Back N (5/6)

```
            /* 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 1 frame */
                  inc(next_frame_to_send);   /* prepare to send the next one */
            }

    }

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

Sliding Window Protocol Using Go Back N (6/6)



Simulation of multiple timers in software.

PROTOCOL 6: A Sliding Window Protocol Using Selective Repeat (1/5)

```
/* Protocol 6 (nonsequential receive) 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 protocol5, 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 */
}
```


Continued →

A Sliding Window Protocol Using Selective Repeat (2/5)

```
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;
```


Continued →

A Sliding Window Protocol Using Selective Repeat (3/5)

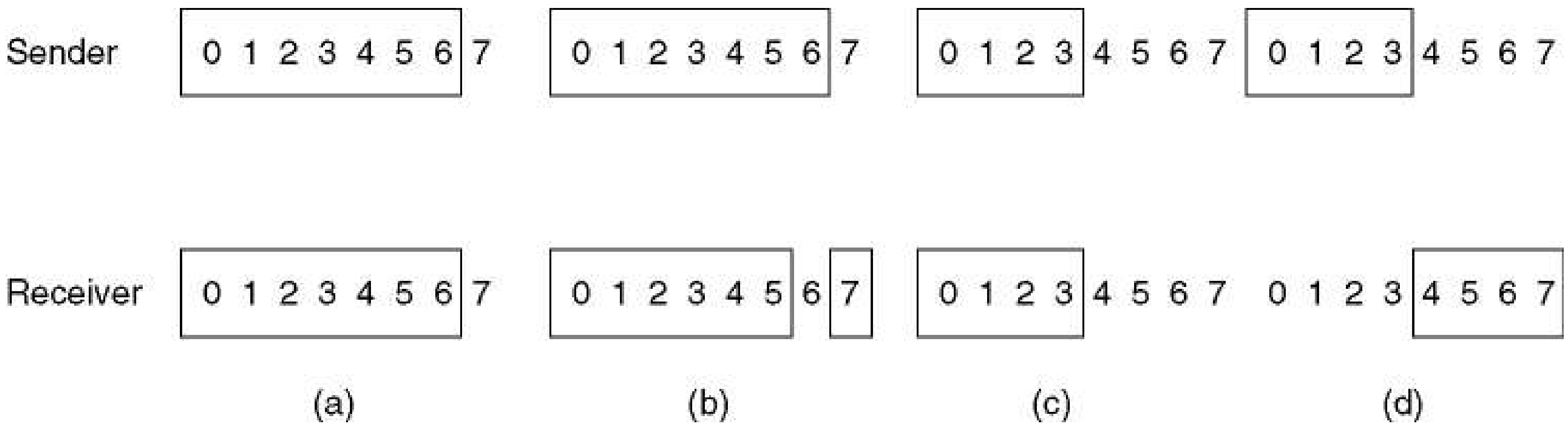
```
while (true) {
  wait_for_event(&event);                    /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                /* accept, save, and transmit a new frame */
          nbuffered = nbuffered + 1;         /* expand the window */
          from_network_layer(&out_buf[next_frame_to_send % 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 */
                      }
                }
          }
```


Continued →

A Sliding Window Protocol Using Selective Repeat (4/5)

```
        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();
 }
}
```

A Sliding Window Protocol Using Selective Repeat (5/5)



(a) Initial situation with a window size seven.

(b) After seven frames sent and received, but not acknowledged.

(c) Initial situation with a window size of four.

(d) After four frames sent and received, but not acknowledged.

Protocol Verification

- Much research has been done trying to find formal, mathematical techniques for specifying and verifying protocols.
- The most widely used techniques are following:

1) Finite State Machined Models

2) Petri Net Models

Finite State Machined Models (1/5)

- Protocol machine (sender or receiver) is always in a specific state at every instant of time.
- The state of protocol machine consists of all the values of its variables, including the program counter.
- The number of states of protocol machine is 2^n, where n is the number of bits needed to represent all the variables combined.

Finite State Machined Models (2/5)

- The states of the complete system is the combination of all the states of the two protocol machines and the channel. In Protocol 3:

-There are two states of sender: (1) sending the frame 0 and (2) sending the frame 1;

-There are two states of receiver: (1) waiting for frame 0 and (2) waiting for frame 1;

-There are four states of channel: 1) a 0 frame moving from sender to receiver; 2) a 1 frame moving from sender to receiver; (3) an ACK frame moving from receiver to sender; (4) an empty channel.

The complete system has 16 distinct states.

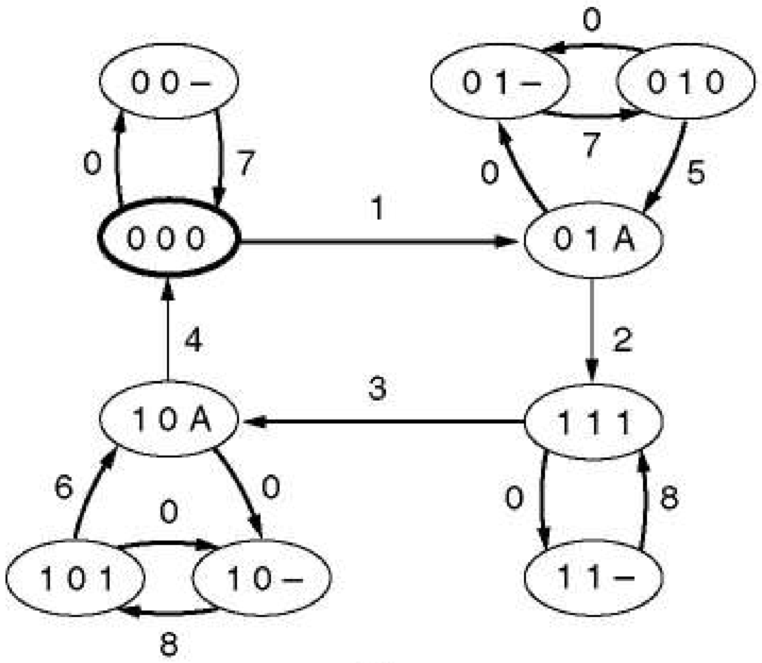
Finite State Machined Models (3/5)

- From each state, there are zero or more possible transitions to other states.
- For a protocol machine, a transition might occur when a frame is sent, when a frame arrives, when a timer expires, when an interrupt occurs, etc.
- For the channel, typical events are insertion of a new frame onto the channel by a protocol machine, delivery of a frame to a protocol machine, or loss of a frame due to noise.

Finite State Machined Models (4/5)

- The initial state corresponds to the description of the system when it starts running, or at some convenient starting place shortly thereafter.
- From the initial state, some, perhaps all, of the other states can be reached by a sequence of transitions.
- The reachablity analysis is helpful in determining whether a protocol is correct.

Finite State Machined Models (5/5)



(a)

Transition	Who runs?	Frame accepted	Frame emitted	To network layer
0	–	(frame lost)		–
1	R	0	A	Yes
2	S	A	1	–
3	R	1	A	Yes
4	S	A	0	–
5	R	0	A	No
6	R	1	A	No
7	S	(timeout)	0	–
8	S	(timeout)	1	–

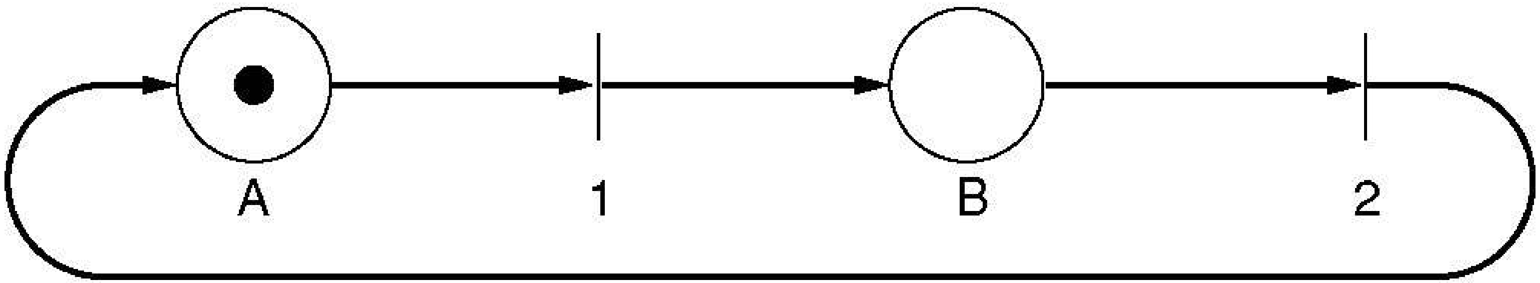
(b)

(a) State diagram for protocol 3. (b) Transmissions.

Each state is labeled by three characters: SRC (Sender Receiver Channel). S={0,1}; R={0,1}; C={0,1,ACK, empty(-)}

{000} is the initial state.

Petri Net Models (1/4)



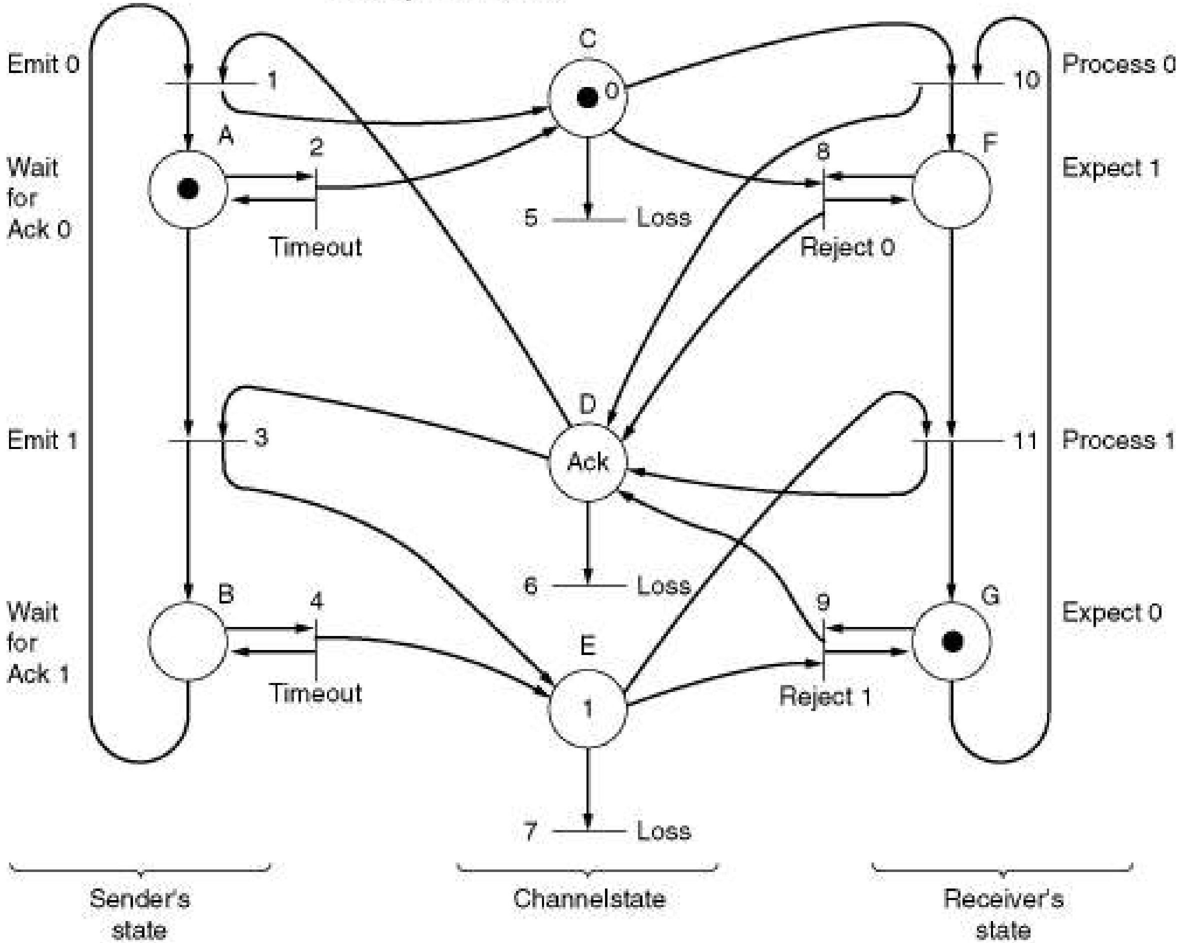
A Petri net with two places and two transitions.

- A place represents a state which (part of) the system may be in.
- A token (heavy dot) indicate the current state.
- A transmission is indicated by a horizontal or vertical bar.
- Input arcs are coming from input places.
- Output arcs are going to output places.

Petri Net Models (2/4)

- A transition is enabled if there is at least one input token in each of its input places.
- Any enabled transition may fire at will, removing one token from each input place and depositing a token in each output place.
- If the number of input arcs and output arcs differs, tokens will not be conserved.
- If two or more transitions are enabled, any one of them may fire.

Petri Net Models (3/4)



A Petri net model for protocol 3.

Petri Net Models (4/4)

- Petri nets can be represented in convenient algebraic form resembling a grammar.
- Each transition contributes one rule to the grammar.
- Each rule specifies the input and output places of the transition.

1:BD→AC	5: C →	9: EG →DG
2:A →A	6: D →	10:CG →DF
3:AD →BE	7: E →	11:EF →DG
4:B →B	8: CF →DF	

Example Data Link Protocols (1/2)

- Many networks use one of the bit-oriented protocols – SDLC (Synchronous Data Link Control), HDLC (High-level Data Link Control), ADCCP (Advanced Data Communication Control Procedure), or LAPB (Link Access Procedure) – at data link level.
- All of these protocols use flag bytes to delimit frames, and bit stuffing to prevent flag bytes from occurring in the data.

Example Data Link Protocols (2/2)

- HDLC – High-Level Data Link Control

This is a classical bit-oriented protocol whose variants have been in use for decades in many applications.

- The Data Link Layer in the Internet

PPP is the data link protocol used to connect home computers to the Internet.

High-Level Data Link Control (1/2)

Bits	8	8	8	≥ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

Frame format for bit-oriented protocols.

- Uses bit stuffing for data transparency.
- Address field is to identify one of the terminals;
- Control field is used for sequence numbers, acknowledgements, and other purpose.
- Data field may contain any information.
- Checksum field is a cyclic redundancy code.

High-Level Data Link Control (2/2)

Bits	1	3		1	3
(a)	0	Seq		P/F	Next
(b)	1	0	Type	P/F	Next
(c)	1	1	Type	P/F	Modifier

Control field of

(a) An information frame.

(b) A supervisory frame.

(c) An unnumbered frame.

Seq is the frame sequence number.

Next is piggybacked acknowledgement.

Type is for distinguishing various kinds of Supervisory frames.

P/F is Poll/Final.

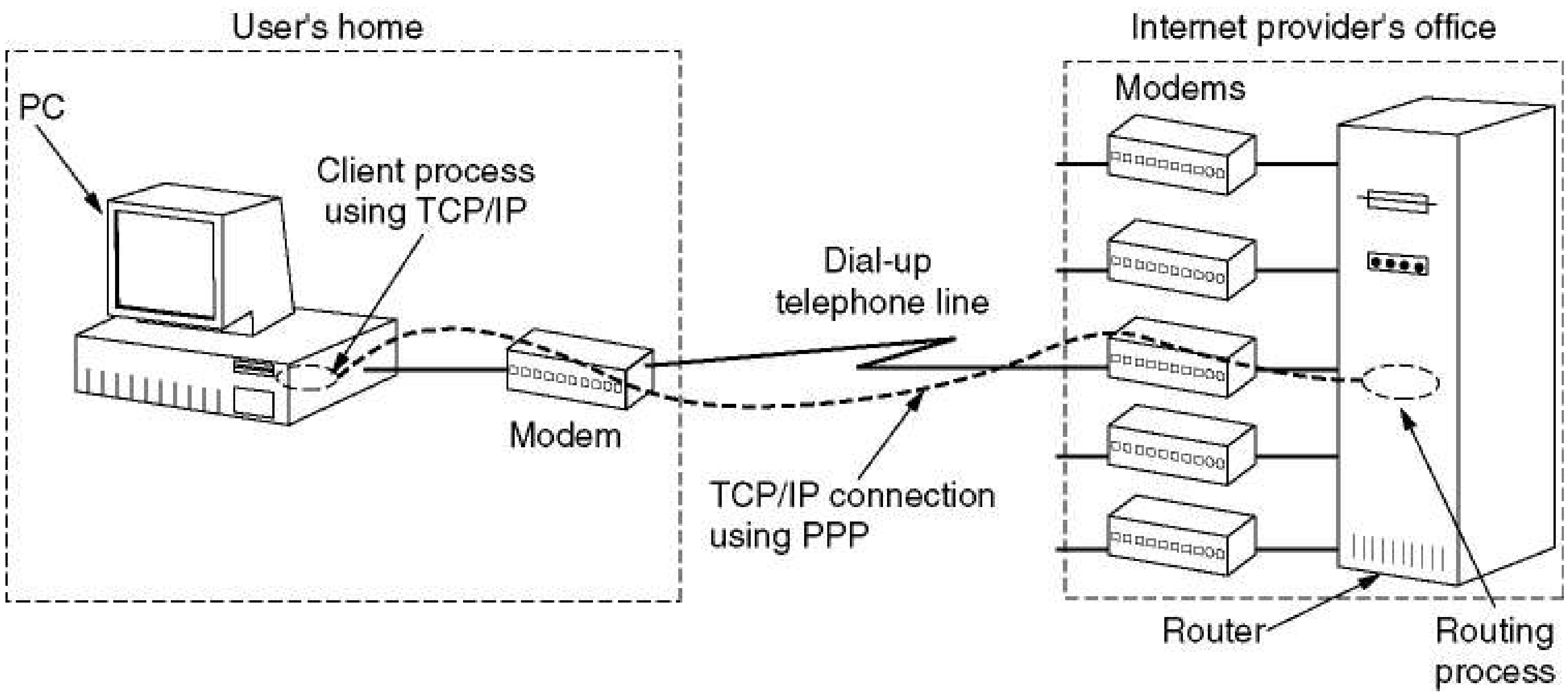
The Data Link Layer in the Internet (1/3)

- Internet consists of individual machines (hosts and routers) and the communication infrastructure that connects them.
- LANs are widely used for interconnection, but most of the wide area infrastructure is built up from point-to-point leased lines.
- In Chap.4 we will look at LANs.
- In this section we will examine the data link protocols used on point-to-point lines in the Internet.

The Data Link Layer in the Internet (2/3)

- In practice, point-to-point communication is primarily used in two situations:
- All routers in subnet are communicated by point-to-point leased lines (or router-router leased line connection).
- Many users have home connections to the Internet using modems and dial-up telephone lines (dial-up host-router connection).
- For both router-router leased line connection and dial-up host-router connection, some point-to-point data link protocol is required on line for framing, error control, etc.
- The one used in Internet is called PPP.

The Data Link Layer in the Internet (3/3)



A home personal computer acting as an internet host.

PPP – Point to Point Protocol (1/5)

- PPP handles error detection, supports multiple protocols, allows IP addresses to be negotiated at connection time, permits authentication, and has many other features.
- PPP provides three 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.

PPP – Point to Point Protocol (2/5)

- 2) LCP – Link Control Protocol for bringing lines up, testing them, negotiating options, and bringing them down again gracefully when they are no longer needed.
- 3) NCP – Network Control Protocol .

  A way to negotiate network-layer options in a way that is independent of the network layer protocol to be used.

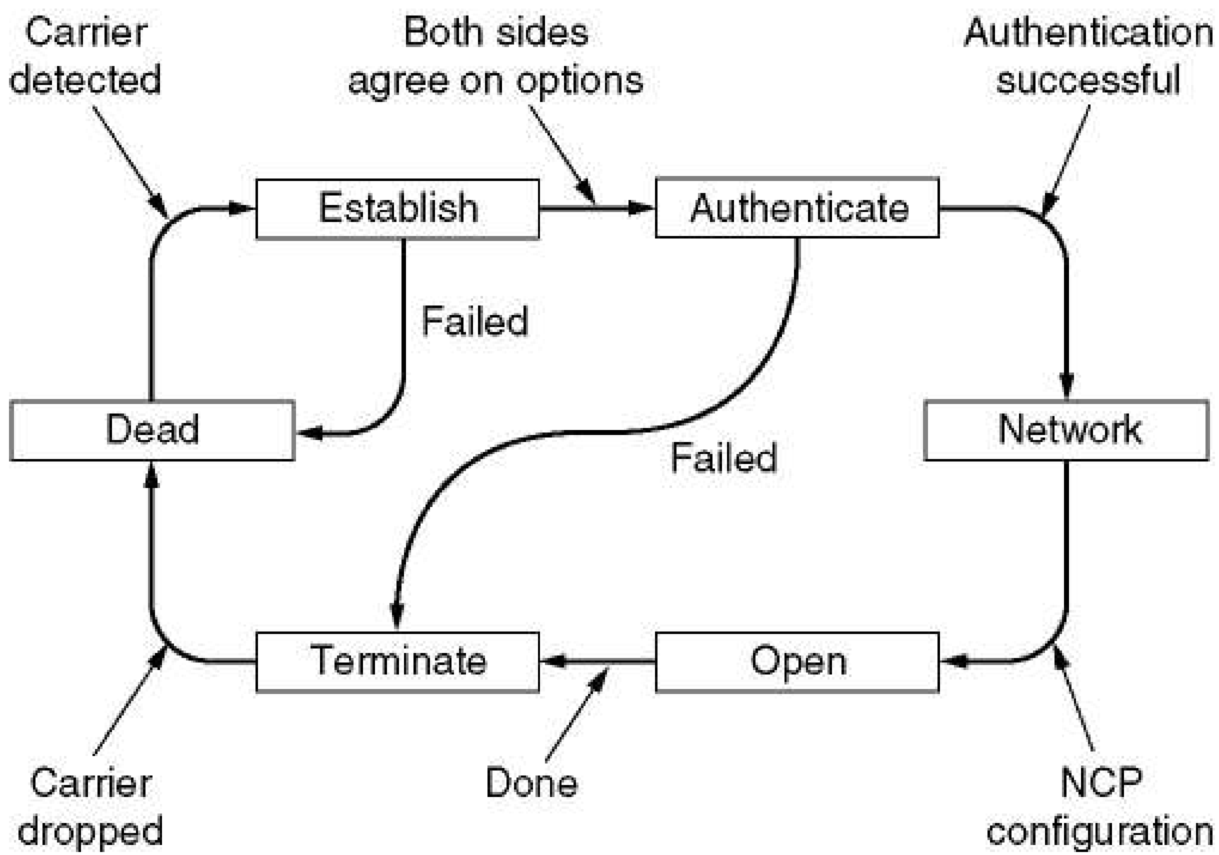
PPP – Point to Point Protocol (3/5)

Bytes	1	1	1	1 or 2	Variable	2 or 4	1
	Flag 01111110	Address 11111111	Control 00000011	Protocol	Payload	Checksum	Flag 01111110

The PPP full frame format for unnumbered mode operation.

- Address is always set to the binary value 1111111 to indicate that all stations are toaccept the frame.
- Conrol is always set to the 00000011 which indicates an unnumbered frame.
- Protocol tells what kind of packet is in the Payload field.

PPP – Point to Point Protocol (4/5)



A simplified phase diagram for bring a line up and down.

PPP – Point to Point Protocol (5/5)

Name	Direction	Description
Configure-request	I → R	List of proposed options and values
Configure-ack	I ← R	All options are accepted
Configure-nak	I ← R	Some options are not accepted
Configure-reject	I ← R	Some options are not negotiable
Terminate-request	I → R	Request to shut the line down
Terminate-ack	I ← R	OK, line shut down
Code-reject	I ← R	Unknown request received
Protocol-reject	I ← R	Unknown protocol requested
Echo-request	I → R	Please send this frame back
Echo-reply	I ← R	Here is the frame back
Discard-request	I → R	Just discard this frame (for testing)

The LCP (Link Control Protocol) frame types.

I –INITIATOR R - RESPONDER