CPE 490: Information Systems Engineering I: Computer Networking

Chapter 3 - The Date Link Layer

Professor Du

Department of Electrical and Computer Engineering
Stevens Institute of Technology
Email: xdu16@stevens.edu

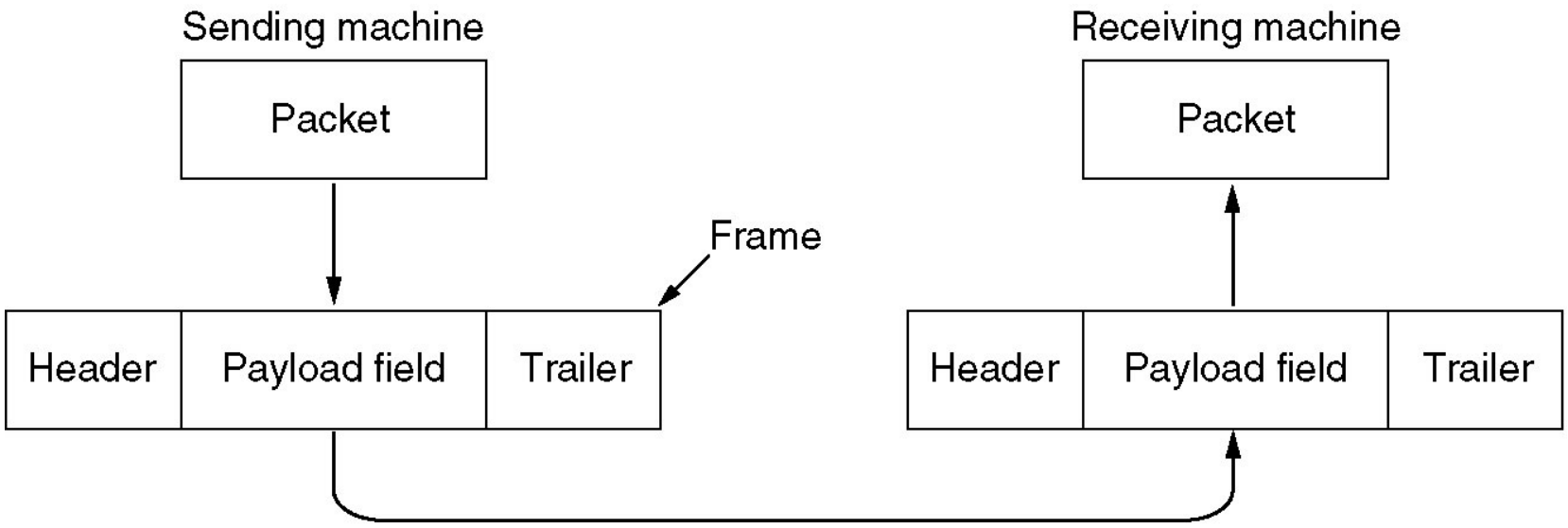
Data Link Layer Design Issues

- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control

Functions of the Data Link Layer

- Provide service interface to the network layer
- Dealing with transmission errors
- Regulating data flow
  - ✓ Slow receivers not swamped by fast senders
- To accomplish these goals,
  - ✓ the data link layer takes the packets from the network layer and
  - ✓ encapsulates them into frames for transmission.
    - Each frame contains a header, a payload field (packet), and a trailer.

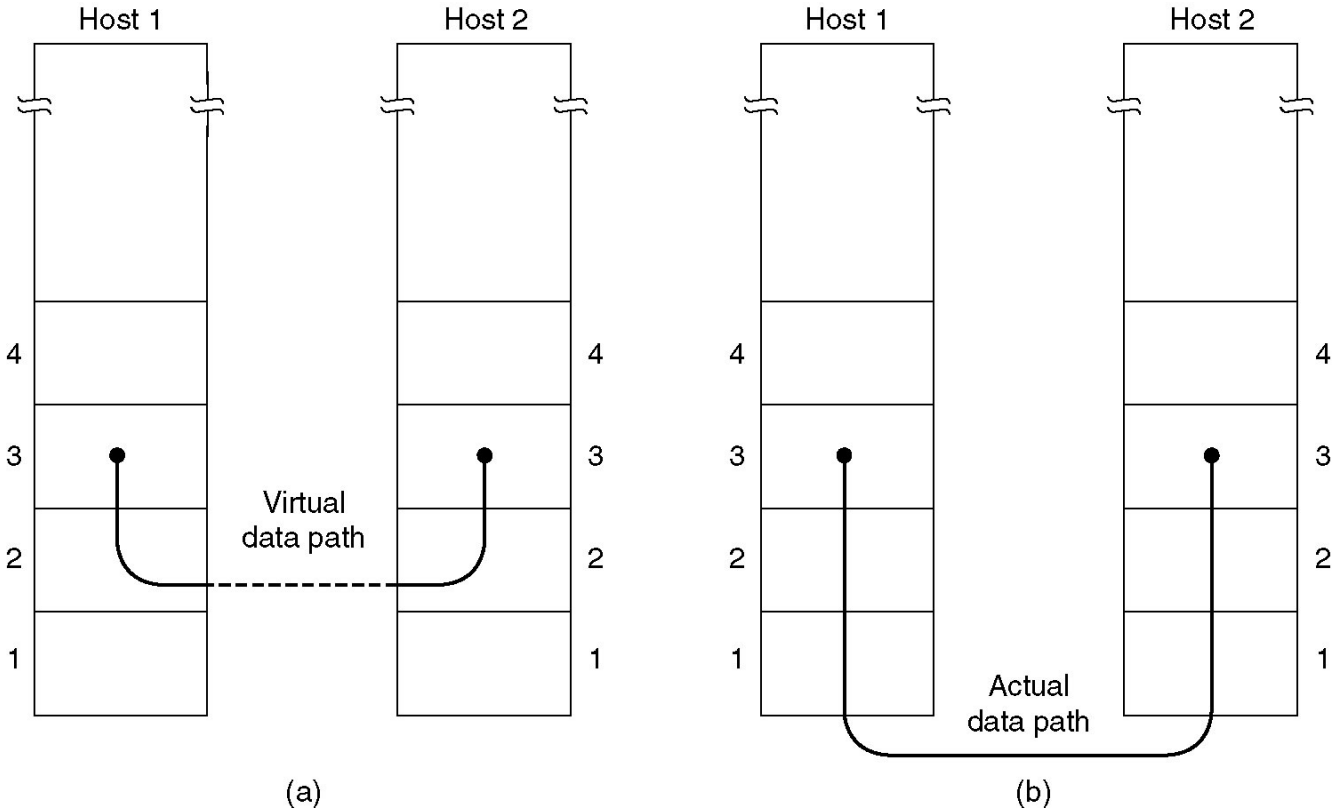
Functions of the Data Link Layer (2)



Relationship between packets and frames.

Services Provided to Network Layer

- The principle service is to
  - ✓ Transfer data from the network layer on source machine to the network layer on the destination machine.



(a) Virtual communication – Using a data link layer protocol.
(b) Actual communication.

Services Provided to Network Layer

- Three typical services
  - ✓ Unacknowledged connectionless service.
  - ✓ Acknowledged (Ack.) connectionless service.
  - ✓ Acknowledged connection-oriented service.
- Unacknowledged connectionless service
  - ✓ The source sends independent frames to the destination without Ack.
  - ✓ No logic connection is established beforehand or released afterward.
  - ✓ If a frame is lost due to noise on the line, no attempt is made to detect the loss or recover it in the data link layer.
  - ✓ This class of service is appropriate when the error rate is very low and the recovery is left to the higher layers.
  - ✓ Most LANs use this service.

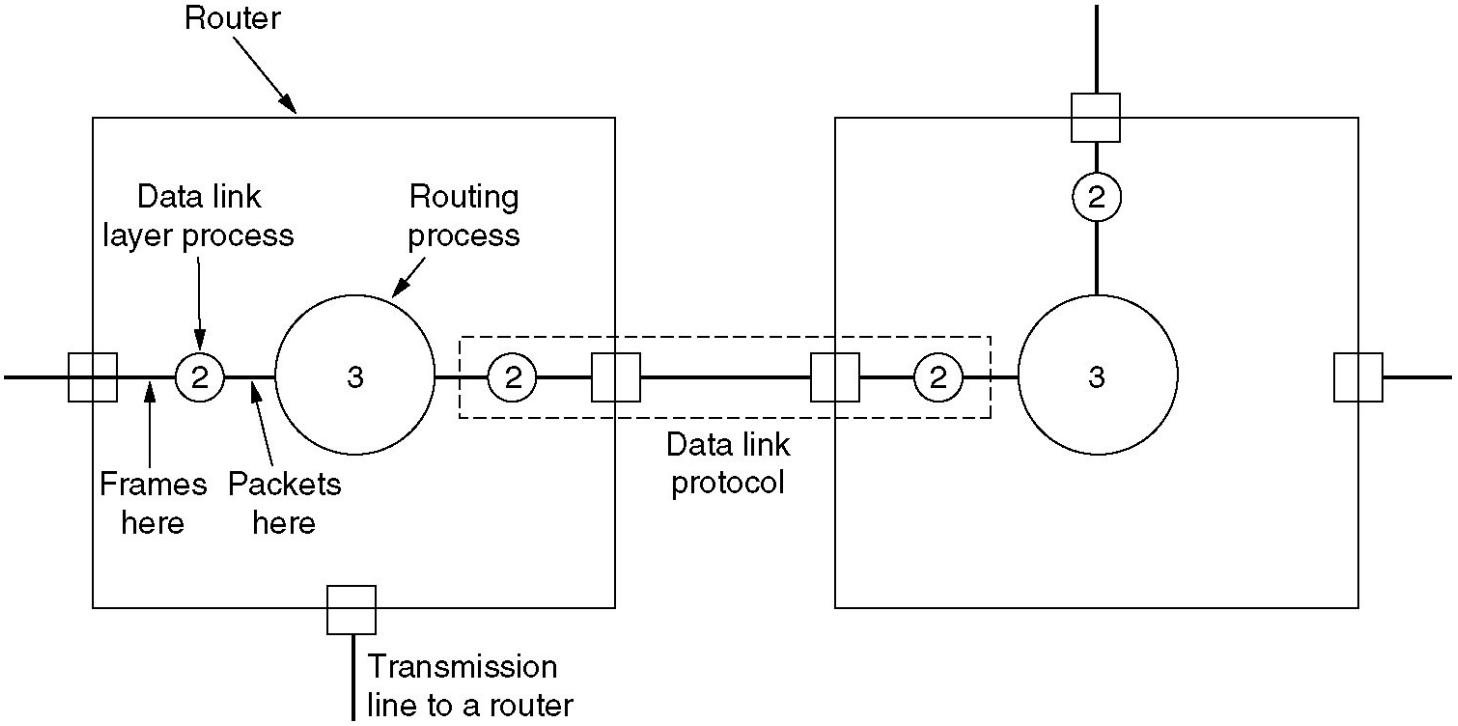
Services Provided to Network Layer

- Acknowledged connectionless service
  - ✓ No logic connection, but each frame is acknowledged.
  - ✓ Useful for unreliable channels, such as wireless systems.
  - ✓ Tradeoff between Ack. in data link layer and network layer.
    - Frame has a max. length imposed by the hardware.
    - E.g., a large message (in the network layer) is broken up into 10 frames.
    - Ack in the data link layer is more efficient, if 2 frames are lost.
- Acknowledged connection-oriented service
  - ✓ A connection is established before data transmission.
  - ✓ The data link layer guarantees
    - that each frame is received
    - each frame is received exactly once
    - and all frames are received in the right order.
  - ✓ With connectionless (+ Ack) service,
    - A lost Ack can cause a packet to be sent several times.

Services Provided to Network Layer

- An example - A WAN subnet consisting of routers.
  - ✓ When a frame arrives at a router, the hardware checks it for errors (using error detection and correction codes),
  - ✓ Then passes the frame to the data link layer software
    - which might be embedded in a chip on the network interface board.
  - ✓ The data link layer software checks to see if this is the frame expected, and if so, forwards the payload field (packet) to the routing software (network layer).
  - ✓ the routing software then chooses the outgoing line and passes the packet back down to the data link layer software,
    - which then transmit it.
  - ✓ One copy of the data link layer software handles all the transmission lines.

Services Provided to Network Layer



Placement of the data link protocol.

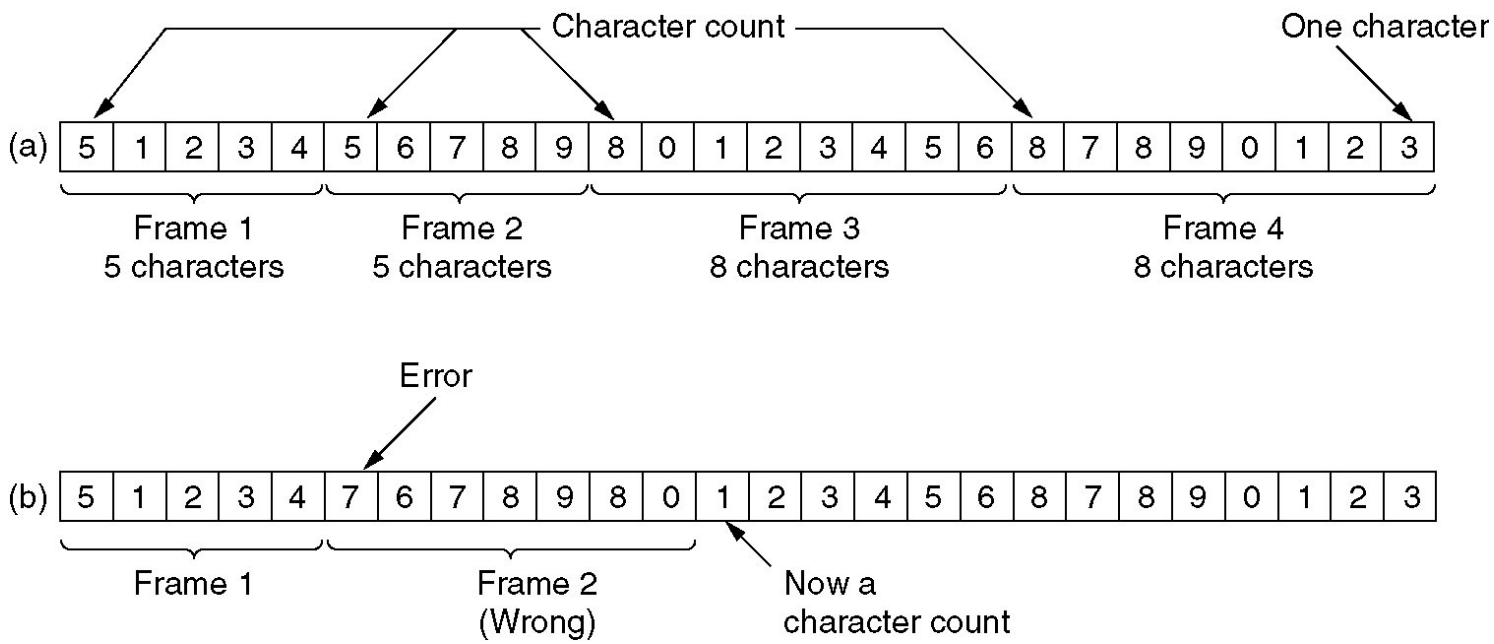
Framing

- The physical layer accepts raw bit streams (0/1).
- The data link layer needs to detect possible transmission errors.
- The usual approach is to break the bit streams into frames and compute checksum for each frame.
  - ✓ When a frame arrives at the receiver, the checksum is recomputed.
  - ✓ If the newly computed checksum is different from the one contained in the frame, the data link layer knows there are errors and can deal with it
    - discarding the bad frame; reporting errors and asking for re-sending.
- Breaking bit streams into frames is more difficult than it seems.
  - ✓ How to do framing?

Framing

- Breaking bit streams into frames
  - inserting time gaps between frames
    - However, timing is hard to be guaranteed in networks.
    - Gaps may be squeezed out and other gaps may be inserted during transmission.
- Three methods
  - Character count
  - Flag bytes with byte stuffing
  - Starting and ending flags, with bit stuffing.
- Character count
  - Using a field in the header to specify the number of characters (bytes) in the frame.
  - Problem – the counter can be garbled by a transmission error.
    - Hard to resynchronize.
  - Rarely used anymore.

Framing

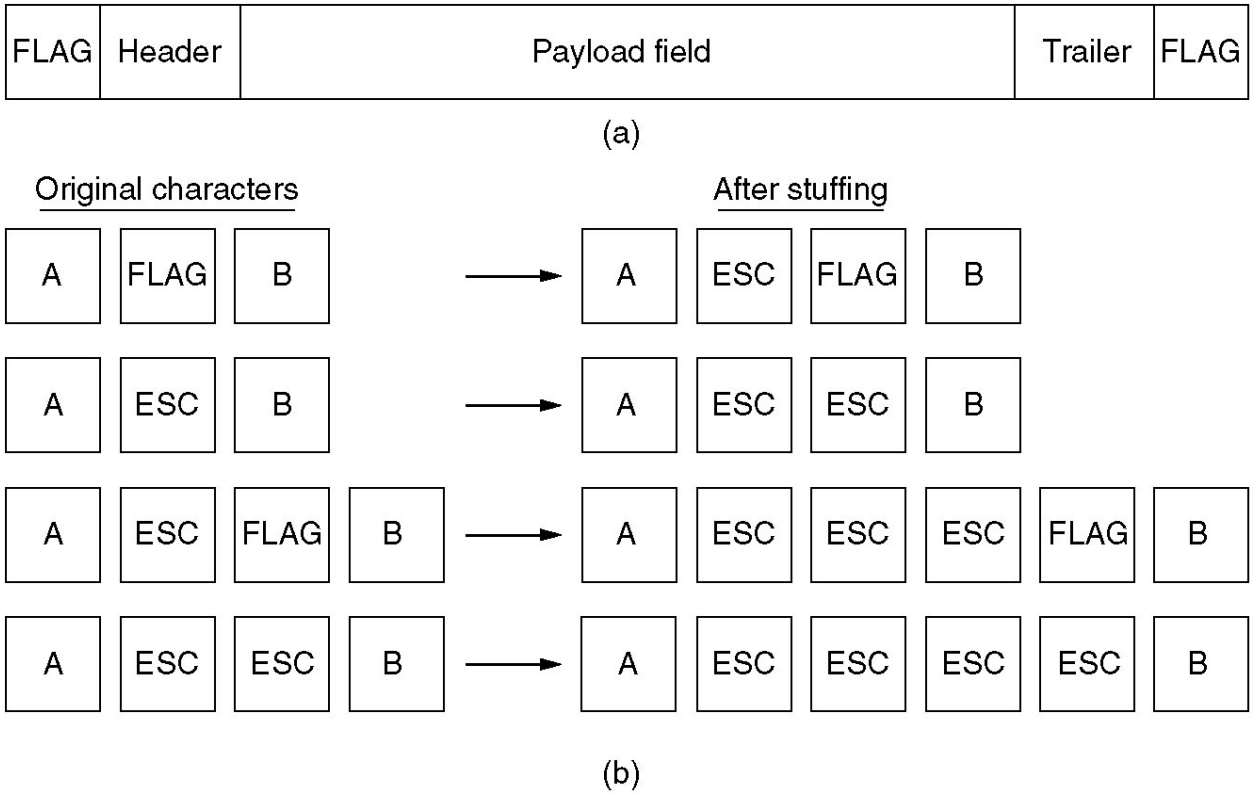


A character stream. (a) Without errors. (b) With one error.

Framing

- Flag bytes with byte stuffing
  - Each frame starts and ends with special bytes.
    - The start byte and the end byte can be different.
    - Most protocols use the same start and end bytes – flag byte.
  - If the receiver loses synchronization, it can just search for the flag byte.
  - The flag byte pattern may occur in the data.
    - Byte (character) stuffing – the sender's data link layer inserts a special escape byte (ESC) before each "accidental" flag byte in the data.
  - What if the ESC byte occurs in the data?
    - stuffing with a ESC byte.
  - A major disadvantage of byte stuffing - it is closely tied to the use of 8-bit characters.
- Starting and ending flags, with bit stuffing
  - For arbitrary sized characters - Bit stuffing.

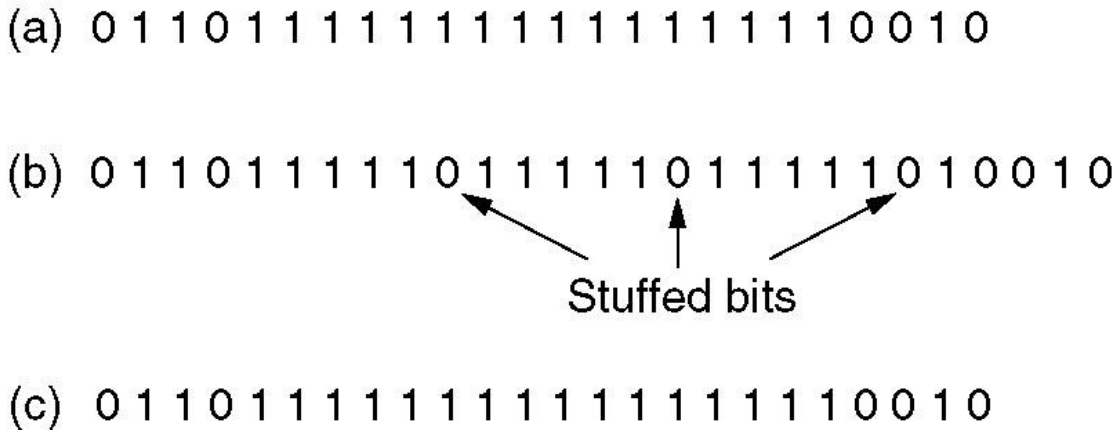
Framing



(a) A frame delimited by flag bytes.

(b) Four examples of byte sequences before and after stuffing.

Framing



Bit stuffing: Each frame begins and ends with a special bit pattern, 01111110 (in fact, a flag byte). Whenever the sender's data link layer encounters **five consecutive 1s (11111)** in the data, it automatically stuffs a 0 bit into the outgoing bit stream.

When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit.

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

Error Detection and Correction

- Two strategies for dealing with transmission errors
  - ✓ Including enough redundant information with each packet – to enable the receiver to correct errors.
  - ✓ Including only enough redundancy to allow the receiver to detect errors.
- Error-Correcting Codes
- Error-Detecting Codes
- Which strategy should be used?
  - ✓ depending on the channel reliability.
- m data bits + r check bits = n bits – **codeword**.
- The number of different corresponding bits between two codeword is called the **Hamming distance**.
  - ✓ E.g., 10001001 and 10110001
  - ✓ The Hamming distance is 3.

Error Detection and Correction

- In most data transmission applications, all 2^m possible data messages are legal.
- However, not all of the 2^n possible codewords are legal,
  - ✓ since the check bits are computed by certain algorithm.
- Given the coding algorithm, the complete list of the legal codewords can be constructed.
  - ✓ For a received packet (a codeword with possible errors), a codeword in the list with minimum Hamming distance is found.
- To detect d bits of errors, a code with minimum distance d+1 is needed.
- To correct d errors, a code with minimum distance 2d+1 is needed.
- E.g., a code with only four valid codewords:
  - ✓ 0000000000, 0000011111, 1111100000, 1111111111.
  - ✓ The minimum distance of the code is 5.
  - ✓ It can correct double errors, e.g., receiving 0000000111 → 0000011111.
  - ✓ However, a triple error changes 0000000000 to 0000000111 can not be corrected.

Error Detection and Correction

- A simple Error-Detecting Code - Parity bit
  - The parity bit is chosen so that the number of 1 bits in the codeword is even.
  - E.g., 1011010+0; 1001010+1;
- An Error-Correcting Code – A **Hamming code**.
  - The bits that are power of 2 (1, 2, 4, 8, etc) are check bits.
  - The rest bits (3, 5, 6, 7, 9, etc) are data bits.
  - Each check bit forces the parity of some collection of bits, including itself, to be even.
  - A data bit may be included in several parity computations.
  - For a data bit k, rewrite k as a sum of powers of 2,
    - E.g., 11 = 1+2+8.
    - 11 contributes to check bits 1, 2, and 8

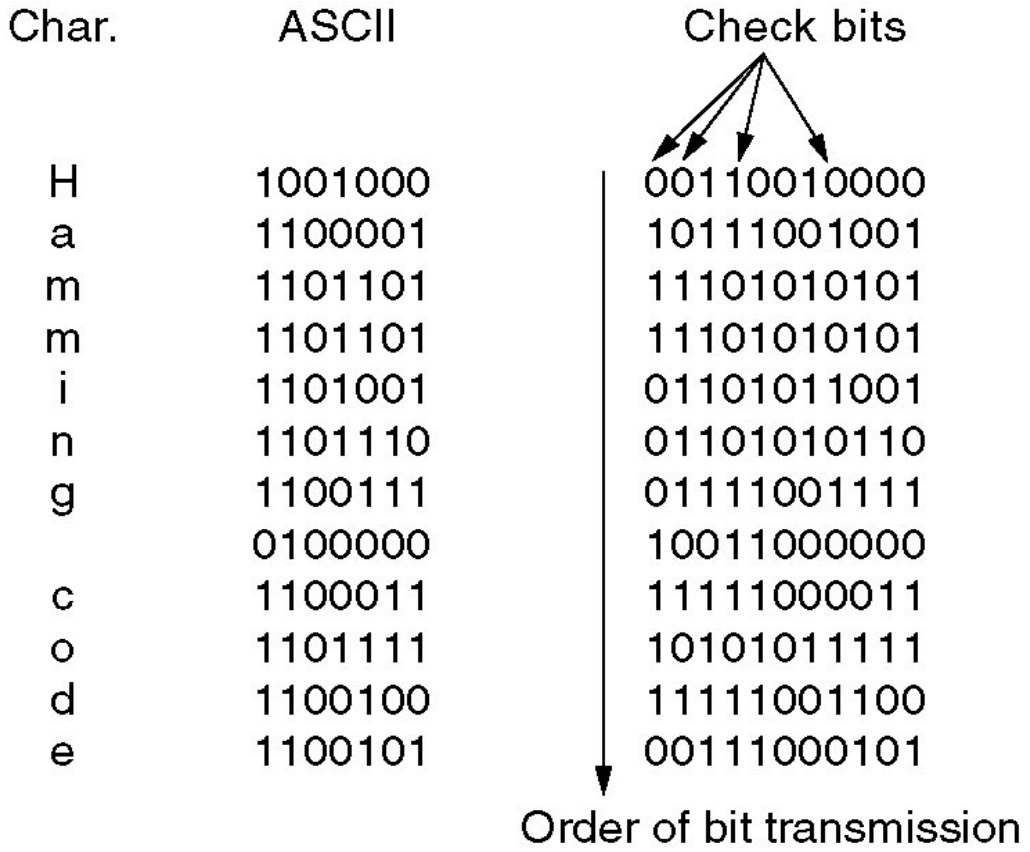
Check bit	Contributing data bits
1	3, 5, 7, 9, **11**
2	3, 6, 7, 10, **11**
4	5, 6, 7
8	9, 10, **11**

Error Detection and Correction

- The Hamming code can correct 1-bit transmission error.
- When a codeword is received, the receiver initializes a counter to zero.
- The receiver then examines each check bit k (k = 1, 2, 4, 8, …).
  - ✓ If the check bit does not have the correct parity, k is added to the counter.
  - ✓ If the counter is zero after all the check bits have been examined, the codeword is accepted as valid.
  - ✓ If the counter is non-zero, it contains the location of the incorrect bit.
  - ✓ E.g., if check bit 1, 2 and 8 are in error, then the error data bit is 11.

Error-Correcting Codes

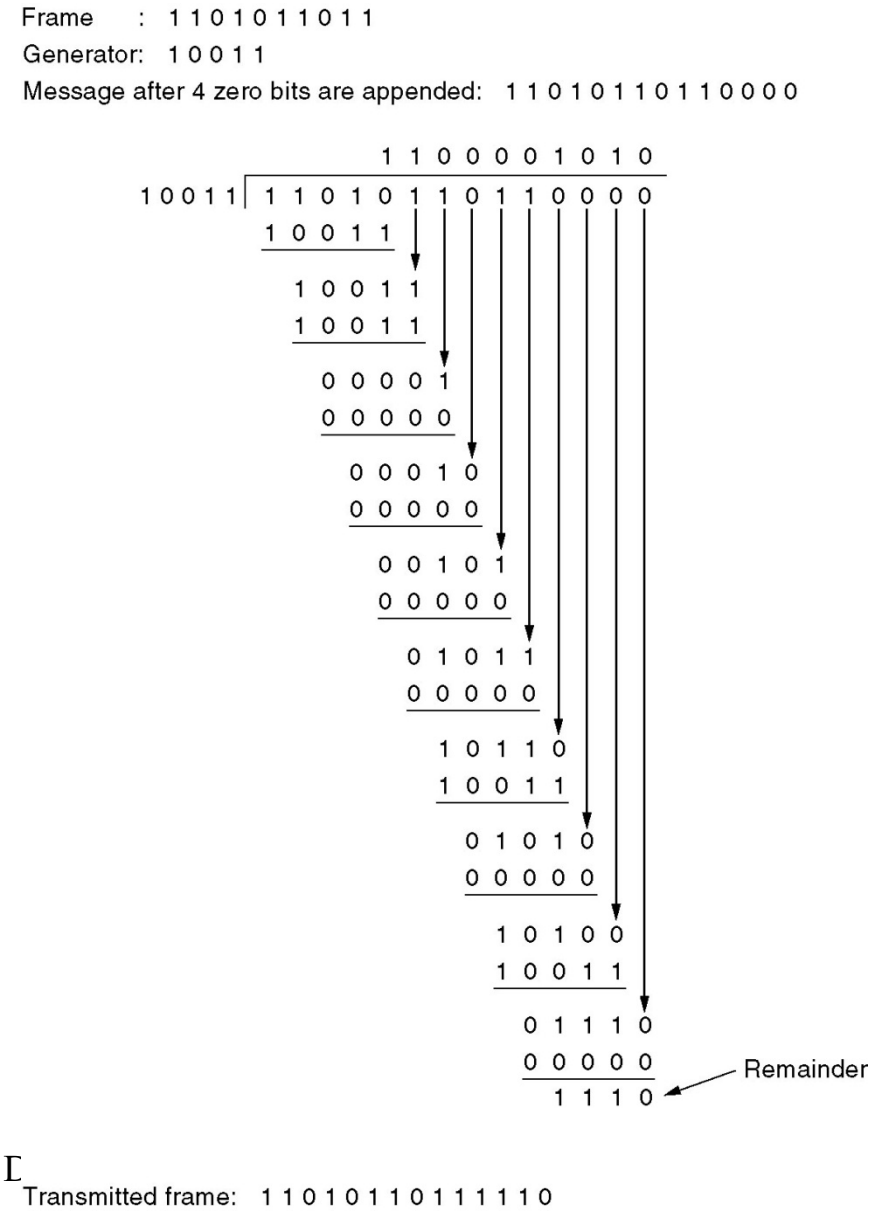
- The Hamming code can be used to correct a single **burst error** – several consecutive bit errors in one transmission.
  - ✓ A sequence of k consecutive codewords are arranged as a matrix.
  - ✓ The data is transmitted one column at a time.
  - ✓ When the frame arrives at the receiver, the matrix is re-constructed.
  - ✓ This approach can correct a single burst error of length k or less.



Error-Detecting Codes

Calculation of the CRC (Cyclic Redundancy Check) code checksum.

- CRC codes are based on treating bit strings as coefficients (0 or 1) of polynomials.
- A frame with m bits – M(x)
- Generator polynomial (degree r) – G(x) used by both the sender and the receiver.
- R(x) – Remainder of $x^r M(x)/G(x)$
- Transmission string – divisible by G(x): $T(x) = x^r M(x) - R(x)$
- Transmission errors – E(x)
- The receiving end – [T(x)+E(x)]/G(x) = E(x)/G(x)
- Errors are detected if E(x)/G(x) is not 0 (or happens to be E(x)/G(x) = 0)



CRC Code

- Some commonly used CRC codes.
- $x^{15} + x^{14} + 1$ will not divide $x^k + 1$ for any value of k below 32,768.
- No polynomial with an odd number of terms has x+1 as a factor in the modulo 2 system.
  - ✓ Odd number of bit errors can be detected by a G(x) with factor X+1.
- A polynomial code with r check bits will detect all burst errors of length =< r.
- The polynomial use by IEEE 802 is

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

  - ✓ It detects all burst errors of length 32 or less and all bursts affecting an odd number of bits.
- CRC can be computed by a simple shift register circuit – efficient computation achieved by hardware.
- Virtually all LANs use CRC.

Elementary Data Link Protocols

- An Unrestricted Simplex Protocol
  - Data is transmitted in one direction only.
  - Both the transmitting and receiving network layers are always ready – no flow control issue.
  - Processing time can be ignored.
  - Infinite buffer space is available.
  - The communication channel between the data link layers never damages or loses frames.
- A Simplex Stop-and-Wait Protocol
  - The receiving network layer needs time to process incoming data.
  - Finite buffer space.
  - The main problem is flow control.
  - Stop-and-wait: the sender sends one frame and then waits for an Ack before proceeding.
- A Simplex Protocol for a Noisy Channel
  - Frames may be either damaged or lost completely.
  - Avoiding duplicate packets in the data link layer.

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.)

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
```

Unrestricted Simplex Protocol

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

Simplex Stop-and-Wait Protocol

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

A Simplex Protocol for a Noisy Channel

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.)

```
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 */
       }
  }
}
```

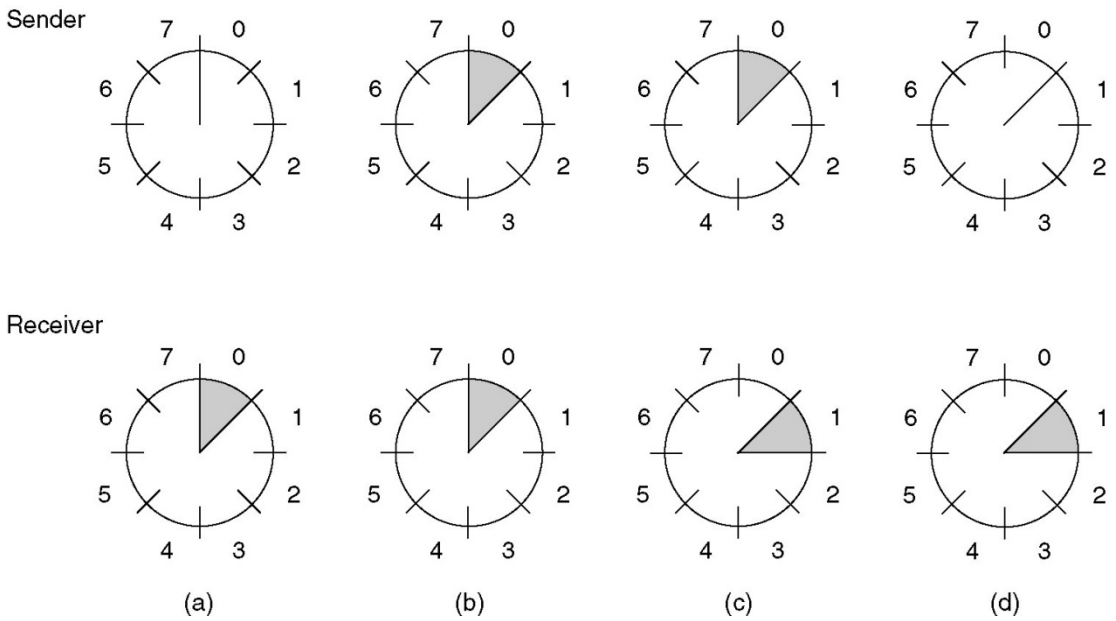
A positive acknowledgement with retransmission protocol.

Sliding Window Protocols

- A One-Bit Sliding Window Protocol
- A Protocol Using Go Back N
- A Protocol Using Selective Repeat

Sliding Window Protocols (2)

- Sender's sequence #: frames that have been sent or can be sent buy not yet Ack. When an Ack comes in, the lower edge increases 1.
- Receiver's window corresponding to the frames it may accept.
- When a frame (seq. # equals to the lower edge), the receiver passes it to network layer and generates an Ack.
- The receiver's window always remains the initial size.



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 ack. has been received.

A One-Bit Sliding Window Protocol

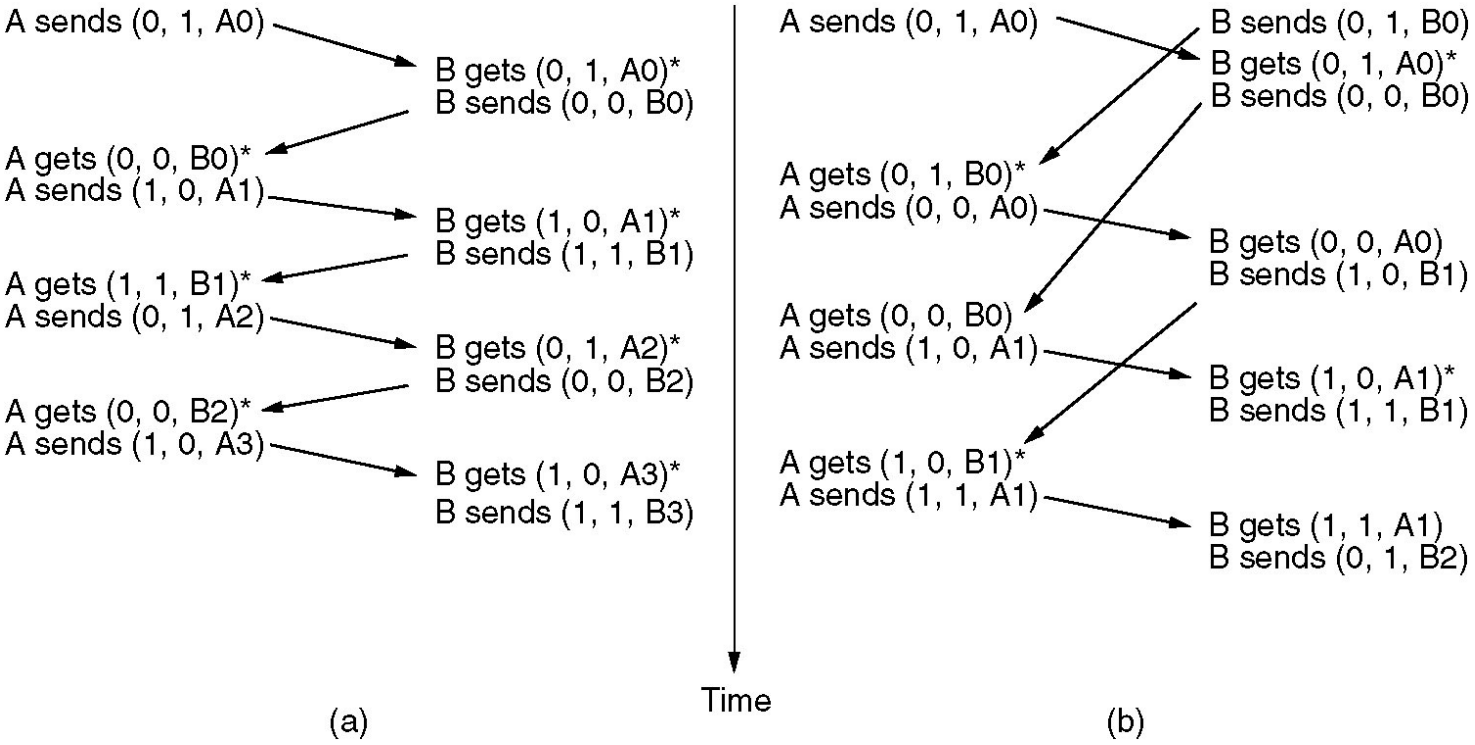
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /*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

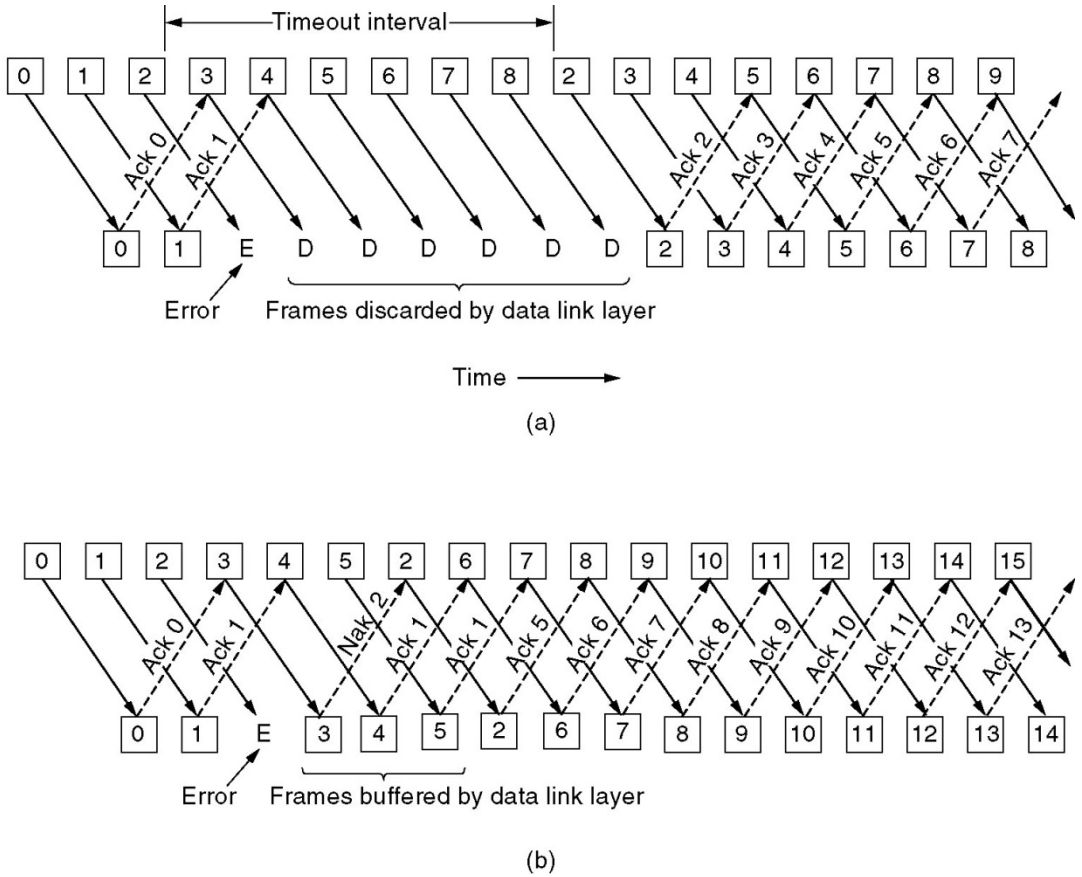
```
while (true) {
    wait_for_event(&event);                 /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {           /* a frame has arrived undamaged. */
        from_physical_layer(&r);            /* go get it */

        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 (2)



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.

A Protocol Using Go Back N



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

```
/* 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

```
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

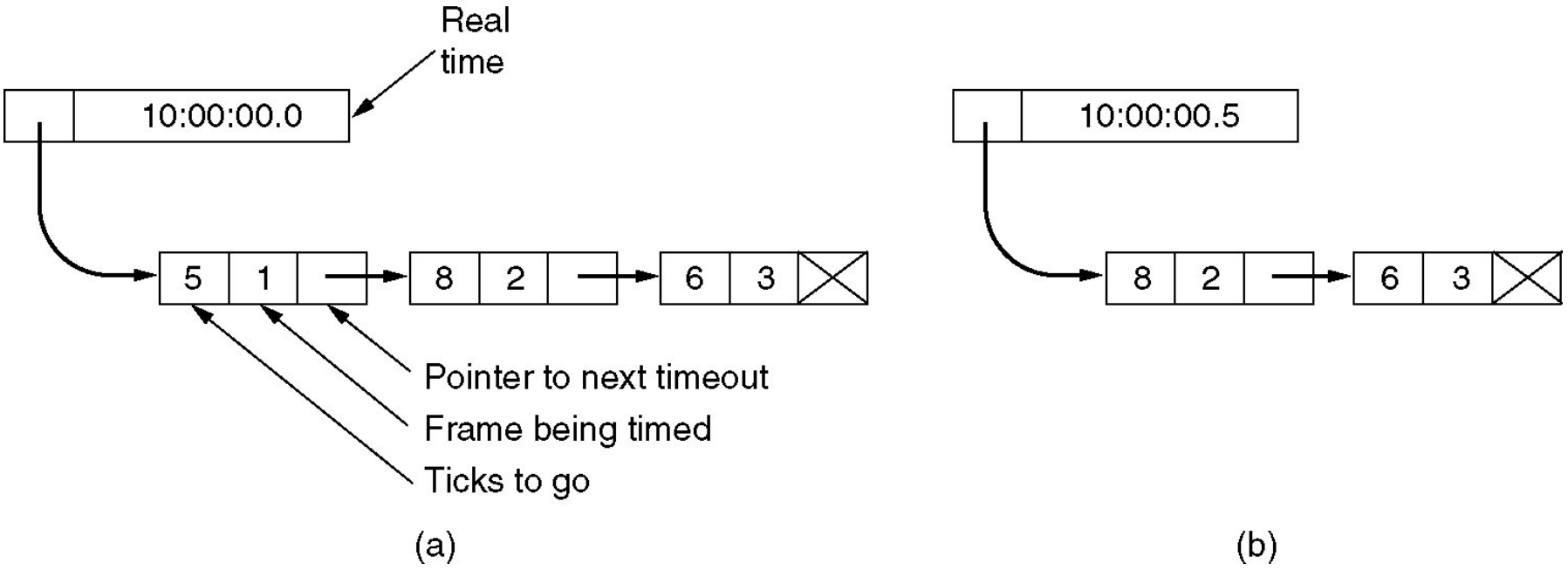
```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  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

```
            /* 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 (2)



Simulation of multiple timers in software.

A Sliding Window Protocol Using Selective Repeat

```
/* 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)) II ((c < a) && (a <= b)) II ((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)

```
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)

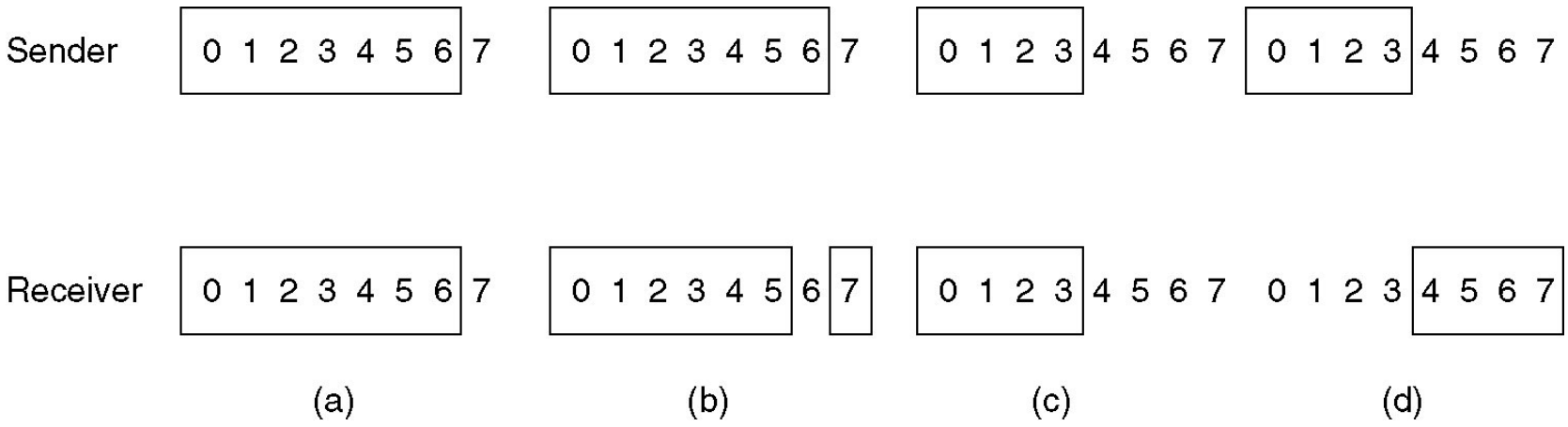
```
while (true) {
  wait_for_event(&event);                          /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                      /* accept, save, and transmit a new frame */
         nbuffered = nbuffered + 1;                /* expand the window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

```
            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)



(a) Initial situation with a window size seven.

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

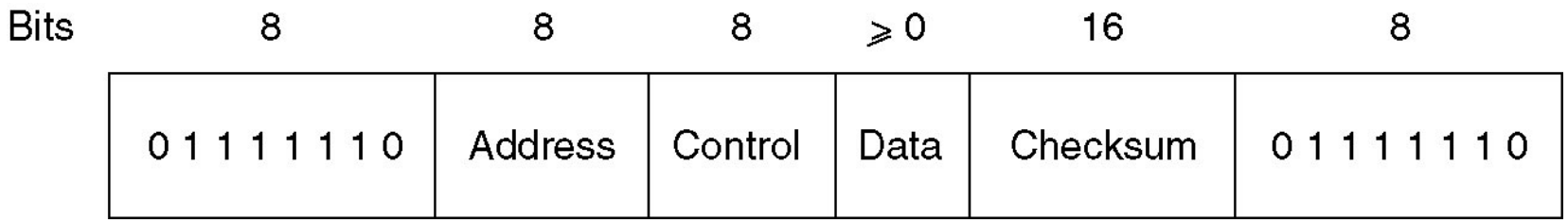
(c) Initial situation with a window size of four.

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

Example Data Link Protocols

- HDLC – High-Level Data Link Control
- The Data Link Layer in the Internet

High-Level Data Link Control



Frame format for bit-oriented protocols.

High-Level Data Link Control (2)

Control field of

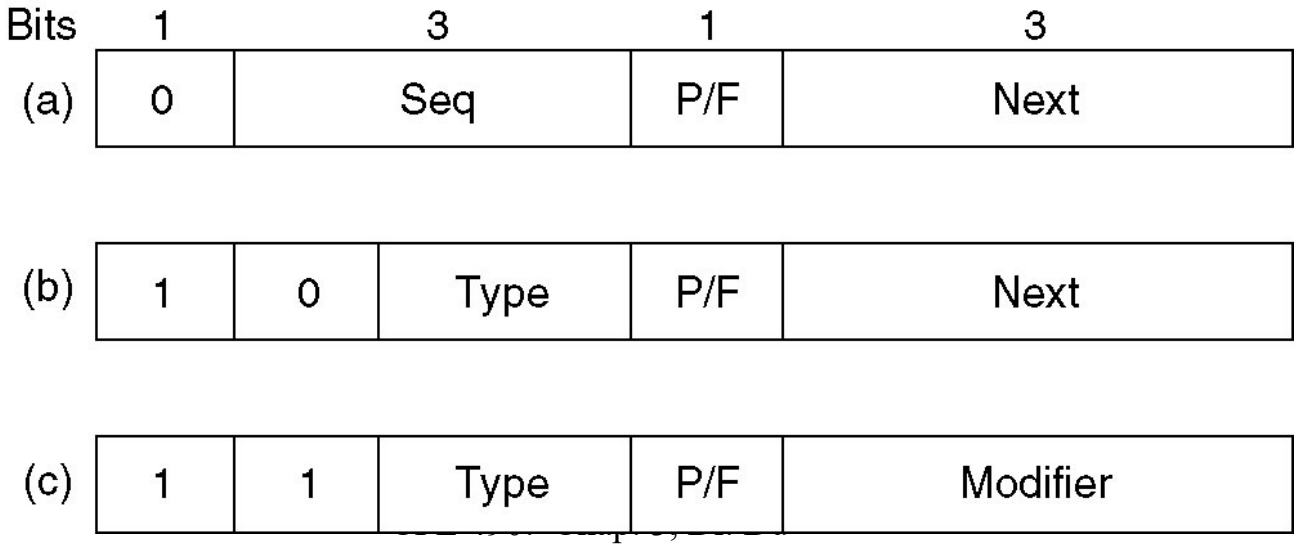
(a) An information frame.

(b) A supervisory frame.

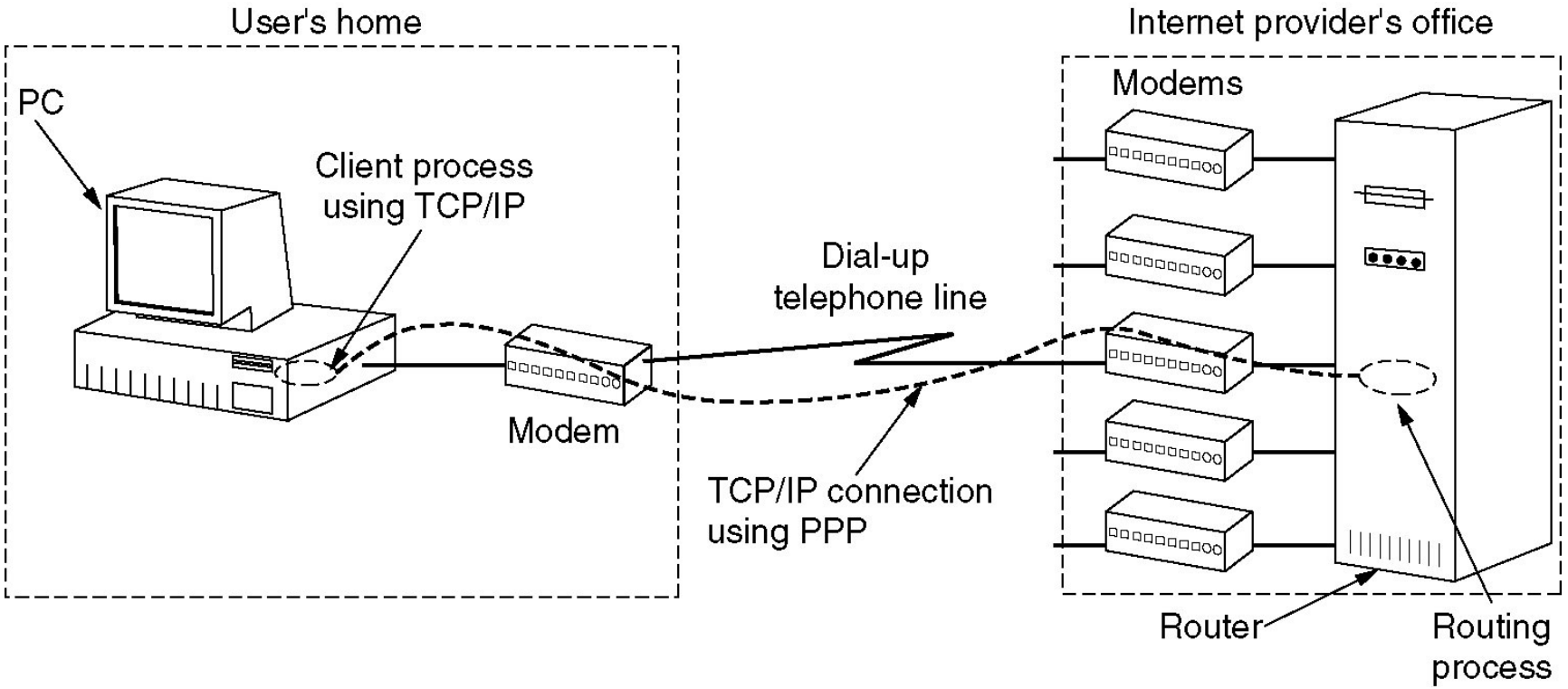
(c) An unnumbered frame.

P/F – Poll/Final: used when a computer polls a group of terminals.

- P - the computer is inviting the terminal to send data.
- The final one is set to F.

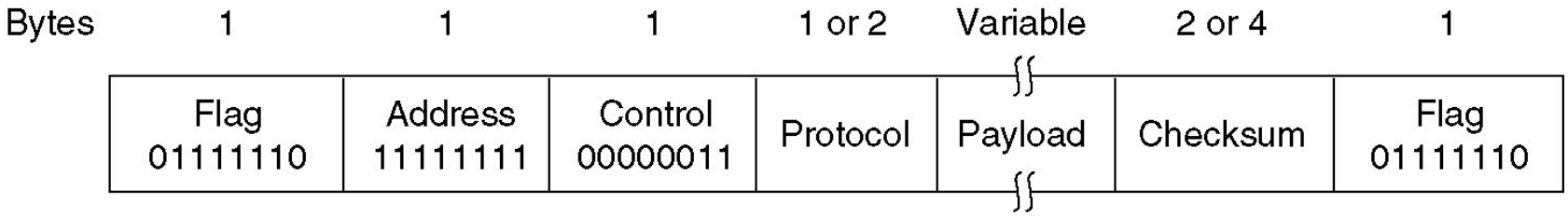


The Data Link Layer in the Internet



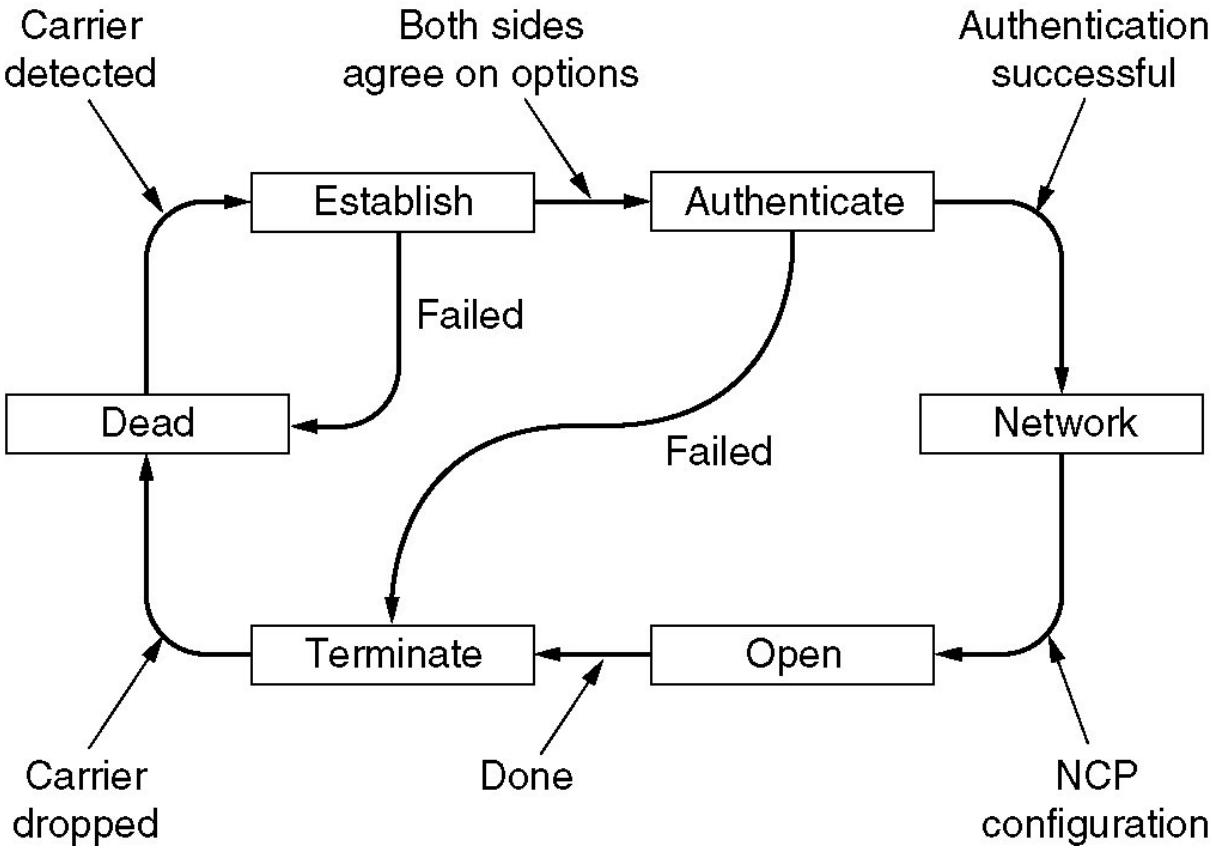
A home personal computer acting as an internet host.

PPP – Point to Point Protocol



The PPP full frame format for unnumbered mode operation.

PPP – Point to Point Protocol (2)



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

NCP - Network Control Protocol

PPP – Point to Point Protocol (3)

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.