CSMC 417

Computer Networks

Prof. Ashok K Agrawala

© 2003 Ashok Agrawala

Set 4

Spring 2003

Chapter 3

The Data Link Layer

Spring 2003

Data Link Layer Design Issues

Services Provided to the Network Layer

Framing

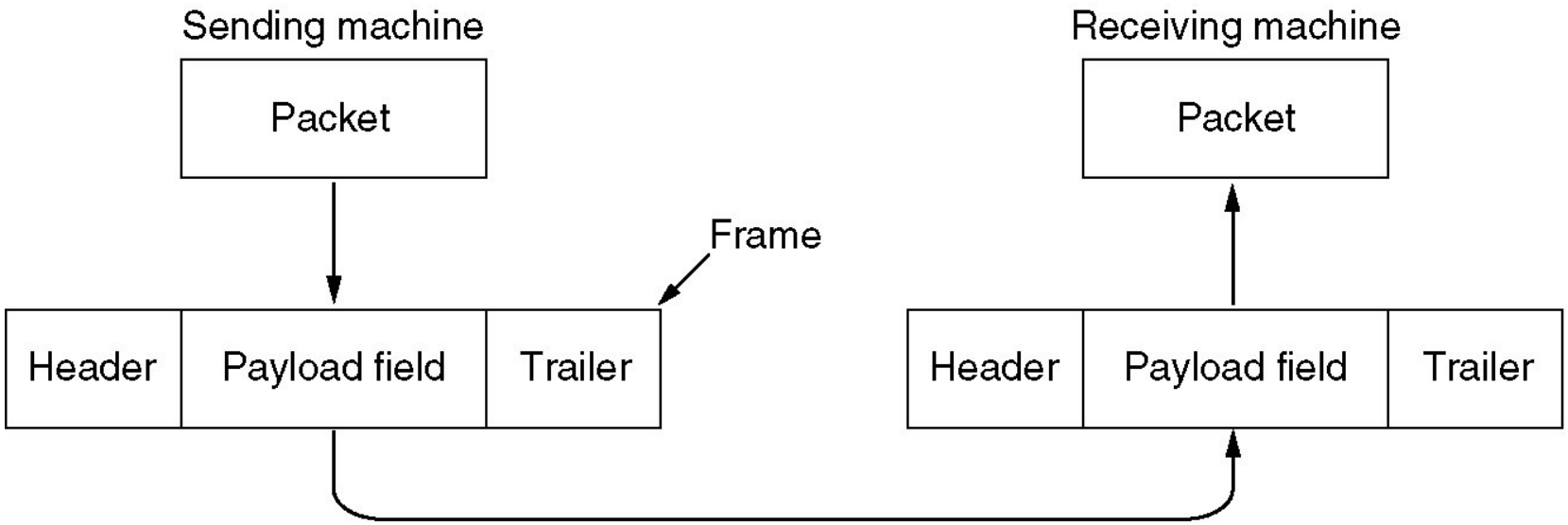
Error Control

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

Functions of the Data Link Layer (2)

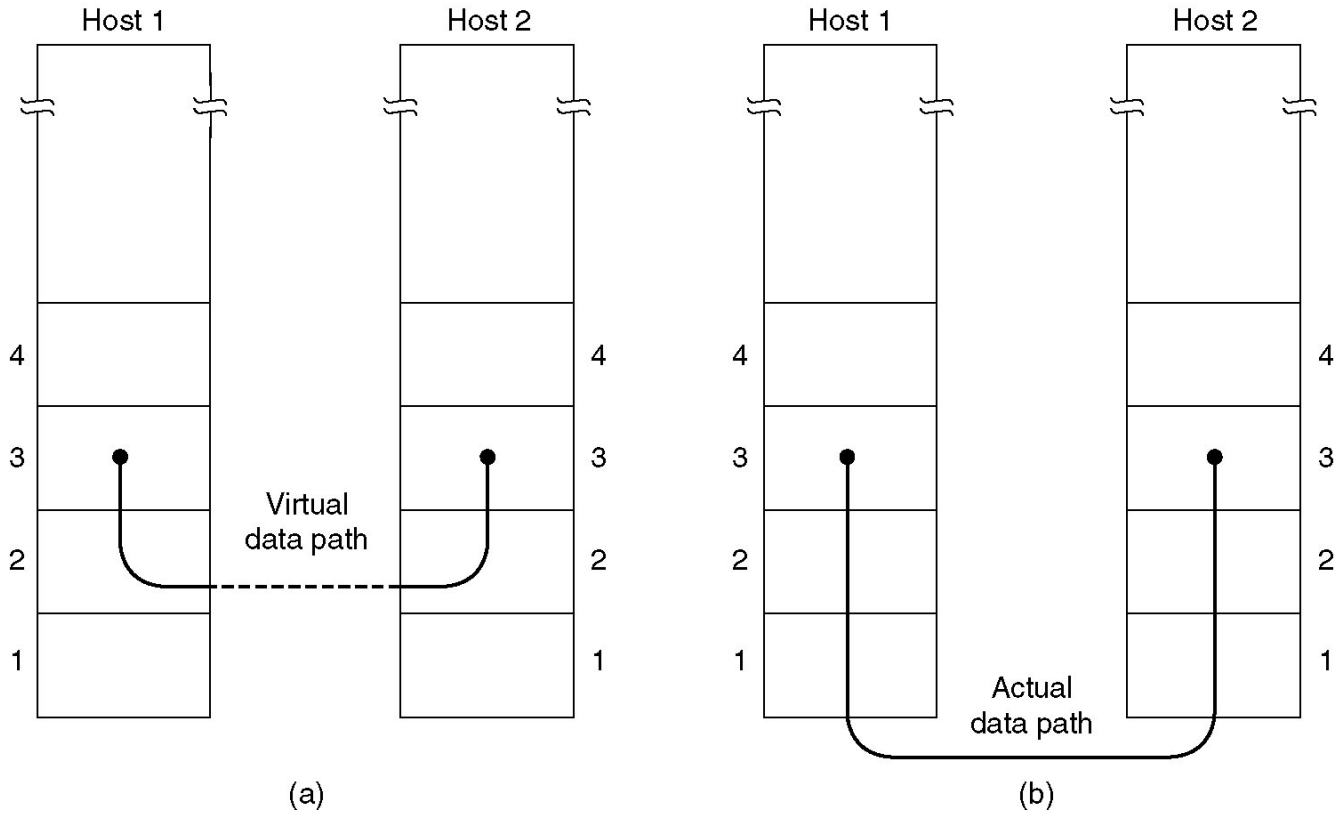


Relationship between packets and frames.

Services Provided

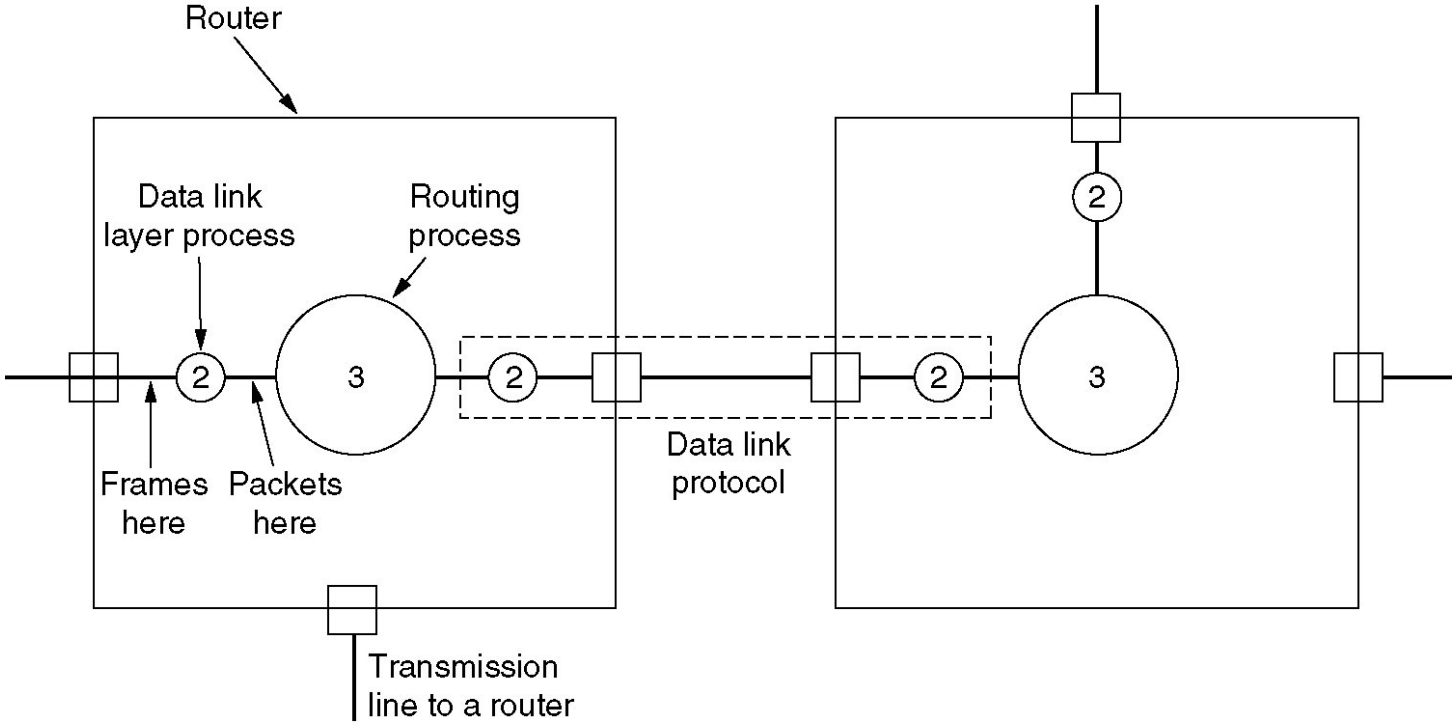
- Unacknowledged connectionless service
- Acknowledged connectionless service
- Acknowledged connection-oriented service

Services Provided to Network Layer



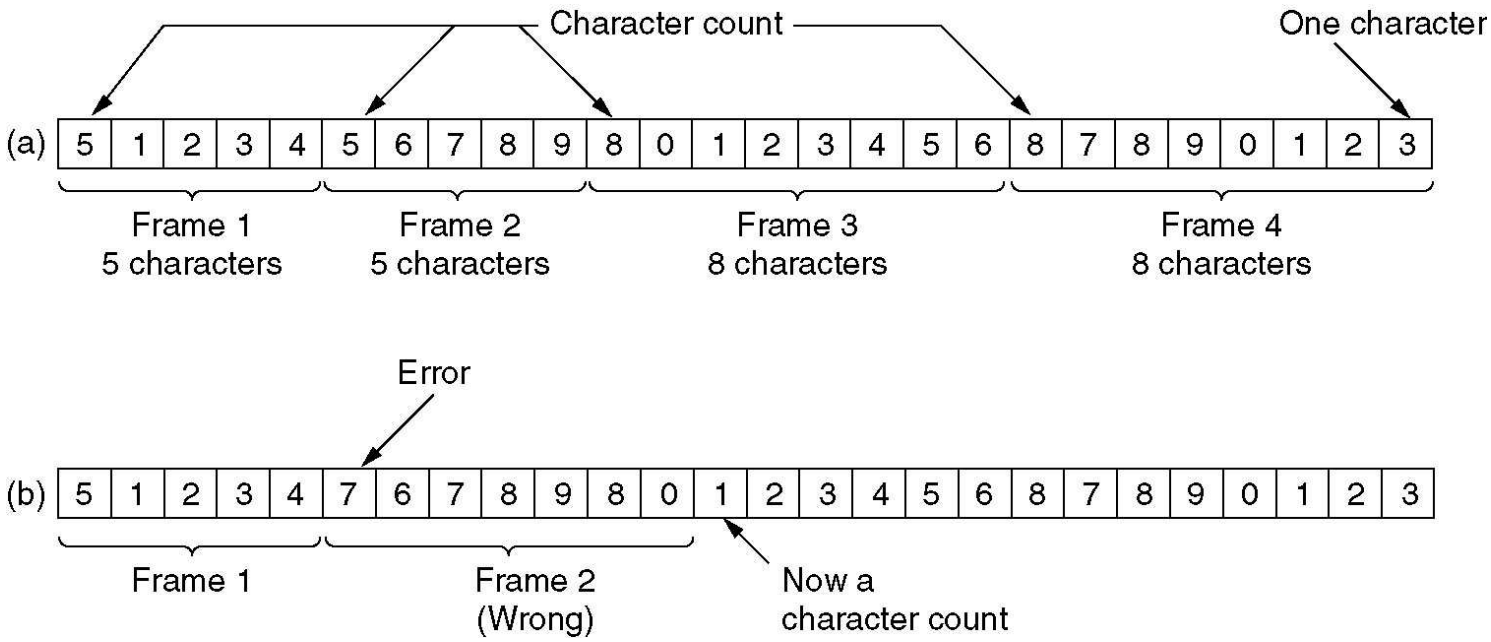
(a) Virtual communication.
(b) Actual communication.

Services Provided to Network Layer (2)



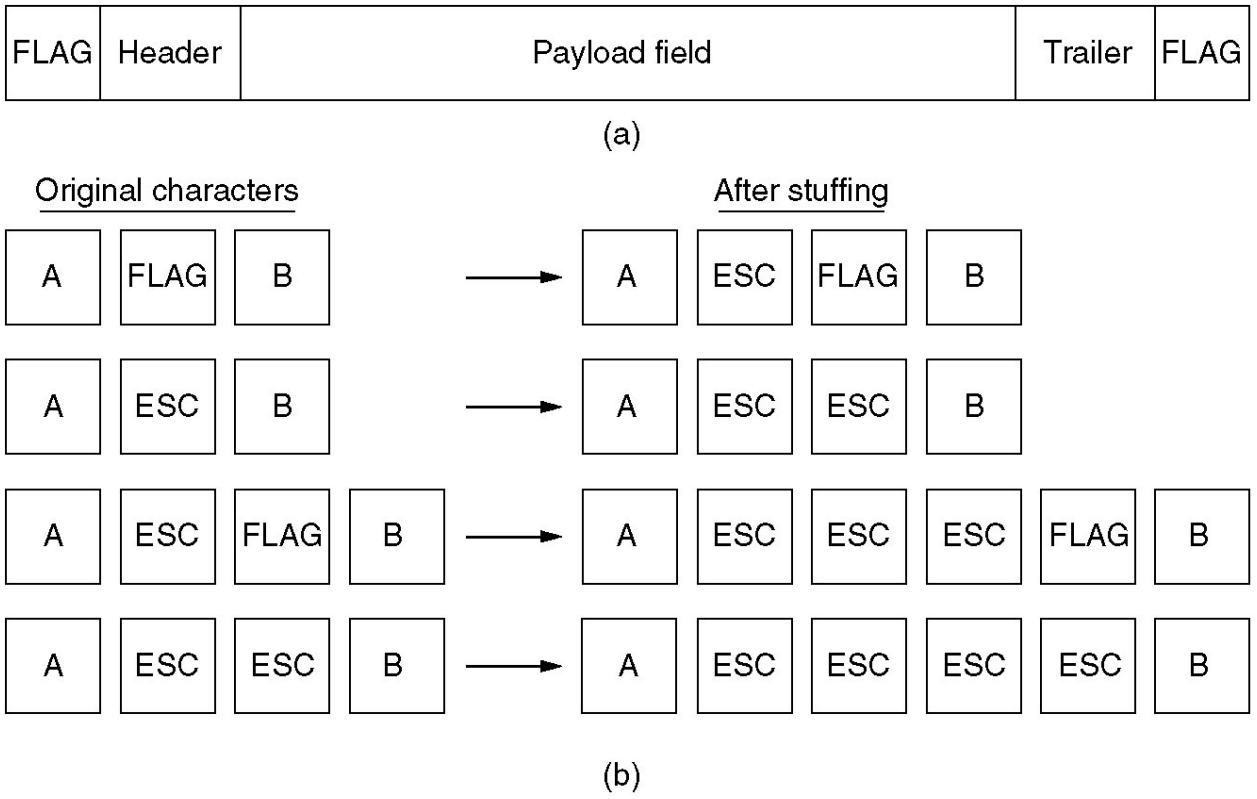
Placement of the data link protocol.

Framing



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

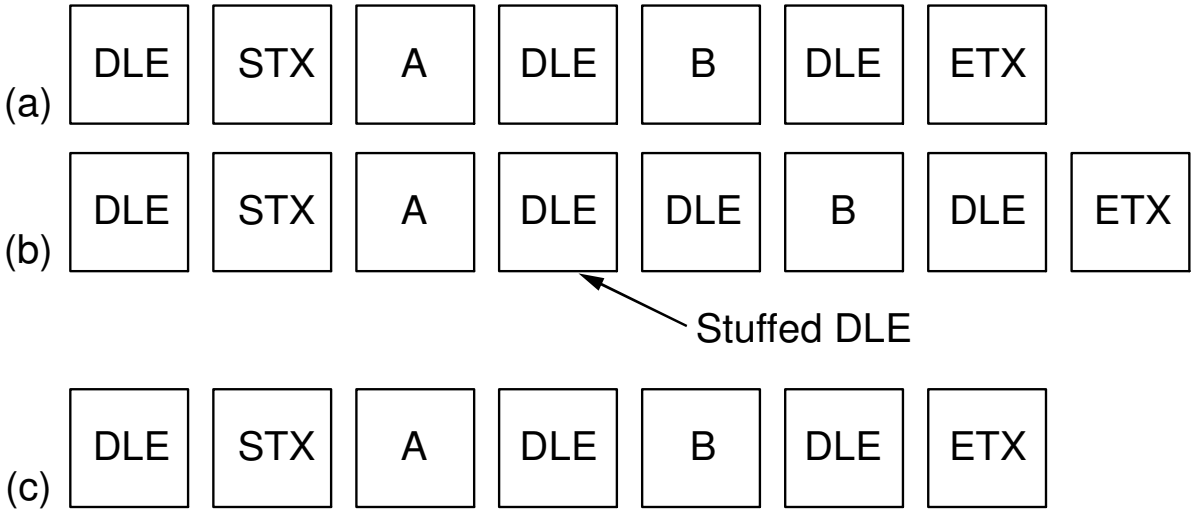
Framing (2)



(a) A frame delimited by flag bytes.

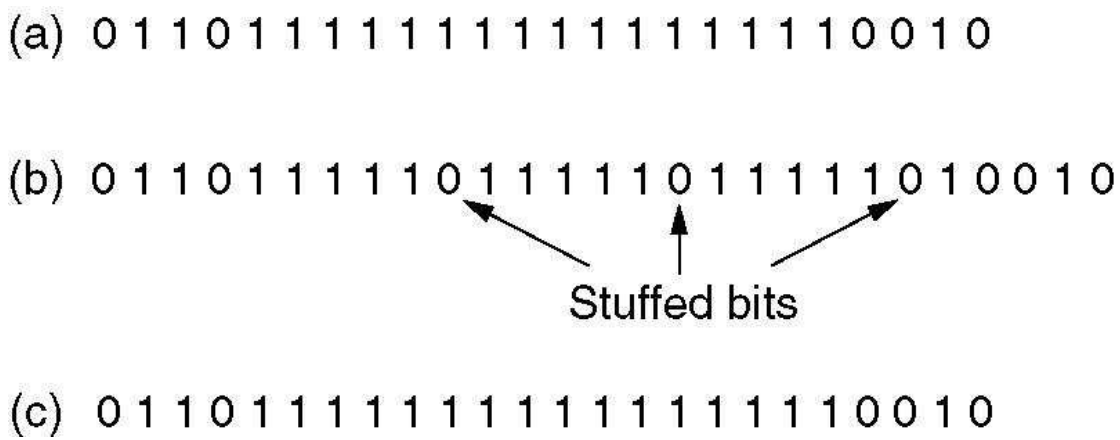
(b) Four examples of byte sequences before and after stuffing

Character Stuffing



- Special Characters at the beginning and the end
  - Stuff with additional character when special characters appear in the text
  - closely tied to ASCII character set
  - Have to embed ASCII character DLE

Framing (3)



Bit stuffing

(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.

Physical Layer Coding

Only applicable when physical medium contains some redundancy

- Some LANS encode 1 bit of data using two physical bits
  - 1 is high-low
  - 0 is low high
    - Easy to locate bit boundaries- transition in the middle
  - high-high and low-low are not allowed
  - IEEE 802 uses this scheme

Error Detection and Correction

Error-Correcting Codes

Error-Detecting Codes

Hamming Distance

- N bits in a codeword (m data bits + r redundant bits)
- Hamming distance –
  - Two codewords
    - Number of bits to be changed to go from one to the other
    - ExOR the words and count the bits
  - Two Codes
    - The minimum hamming distance between two codewords in this code
- Detect d errors if codes d+1 apart
- Correct d errors if codes 2d+1 apart
- Example – Parity
  - Distance = 2

Error Correcting

- Code with m message bits and r check bits (n=m+r)
- Correct single bit errors
- Legal messages = 2^m
- There are n illegal codewords at distance 1 from any legal codeword – These must be reserved – (n+1) 2^m
- Thus $(n+1)2^m < 2n$ or $(m+r+1)<2^r$
- Gives lower limit on r

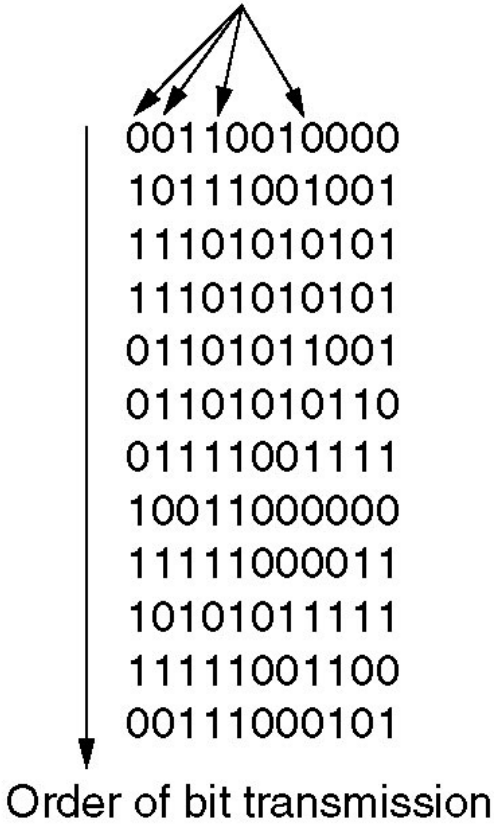
Hamming Code

- Correct one bit errors

- Number bits from left 1,2,3…
- Bits that are power to 2 (1,2,4,8,16,…) are check bits
- Rest are data bits

Error-Correcting Codes

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101



Use of a Hamming code to correct burst errors.

Error Detection

- Polynomial Codes – CRC (cyclic Redundancy Check)
- Generator Polynomial G(x)
  - Both High and low order bits must be 1
- Frame with m bits may be expressed as polynomial M(x)
- Divide $x^r M(x)$ by g(x) using modulo 2 division
- Subtract the remainder (r bits) from $x^r M(x)$ using mod 2 subtraction, resulting in T(x).
- Send T(x)

- On receiving divide T(x) by G(x). If remainder is 0 then no error

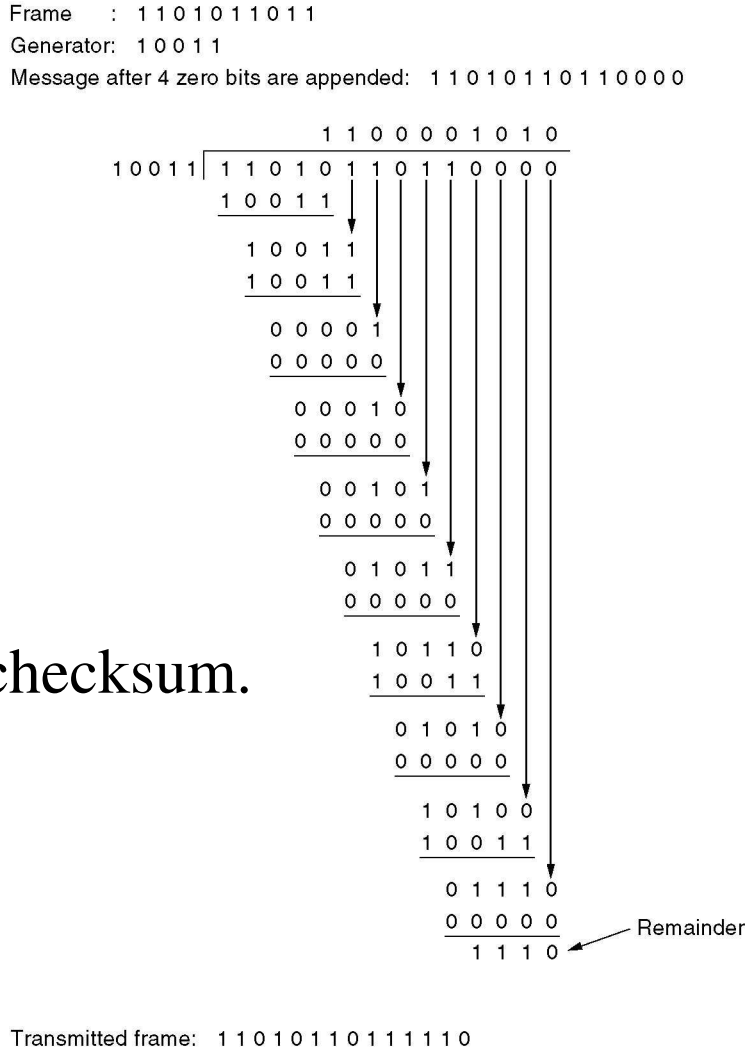
Polynomial Code

Treat bit strings as representation of polynomials with coefficients 0 or 1

- k bit frame - polynomial with terms x^0 to x^{k-1}
- Generator Polynomial G(x) - High and low order bits are 1
  - Let r be the degree of G(x). Append r zeros to the low-order end. So Frame now becomes $x^rM(x)$
  - Divide G(x) into $x^rM(x)$ using Mod 2 arithmetic
  - Subtract the remainder from $x^rM(x)$. Send the result as T(x)
- On receiving divide T(x) by G(x). If result is zero there are no errors

Error-Detecting Codes

Calculation of the polynomial code checksum.



CRC Codes

CRC-12

- $x^{12}+x^{11}+x^3+x^2+x^1+1$
- Good for 6 bit characters

CRC-16

- $x^{16}+x^{15}+x^2+1$
- Good for 8 bit characters
- Catch all single and double bit errors, errors with odd number of bits

CRC-CCITT

- $X^{16}+x^{12}+x^5+1$

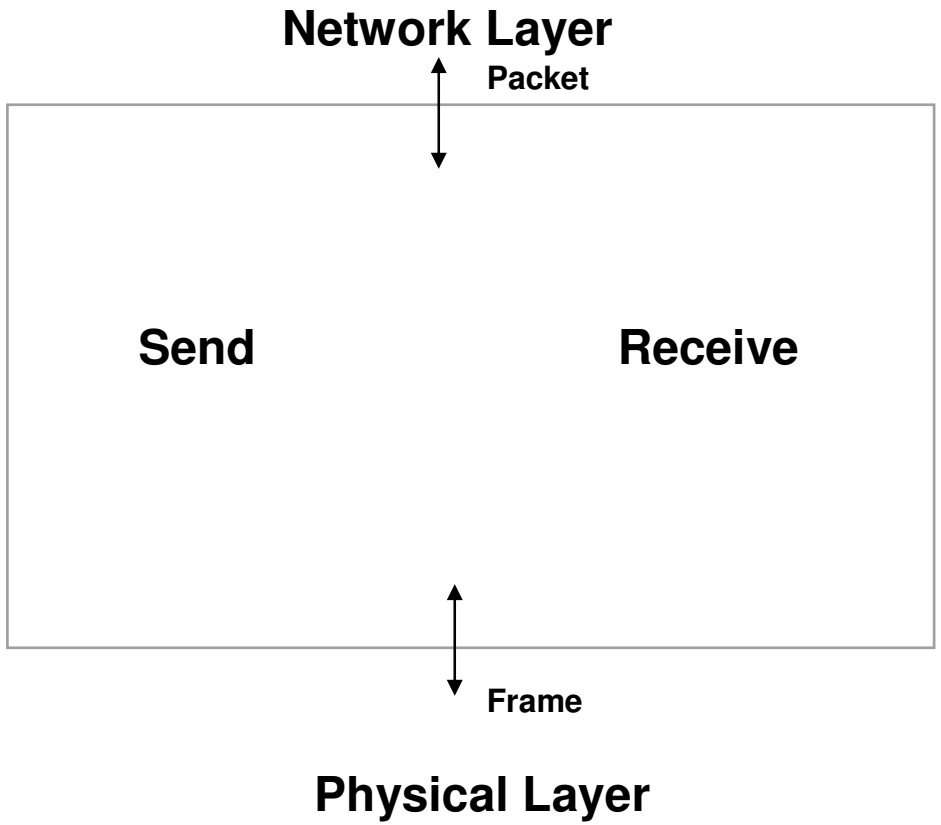
Generating Polynomials

- Many Standard Polynomials have been defined.
- Usually the polynomial to be used is defined with the protocol definition
- For 802
  - $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

Elementary Data Link Protocols

- An Unrestricted Simplex Protocol
- A Simplex Stop-and-Wait Protocol
- A Simplex Protocol for a Noisy Channel

Structure of Data Link Layer



Packet

Frame

DLL Functions

Interface with Network Layer

- `To_Network_Layer`
- `From_Network_Layer`
- Other Functions

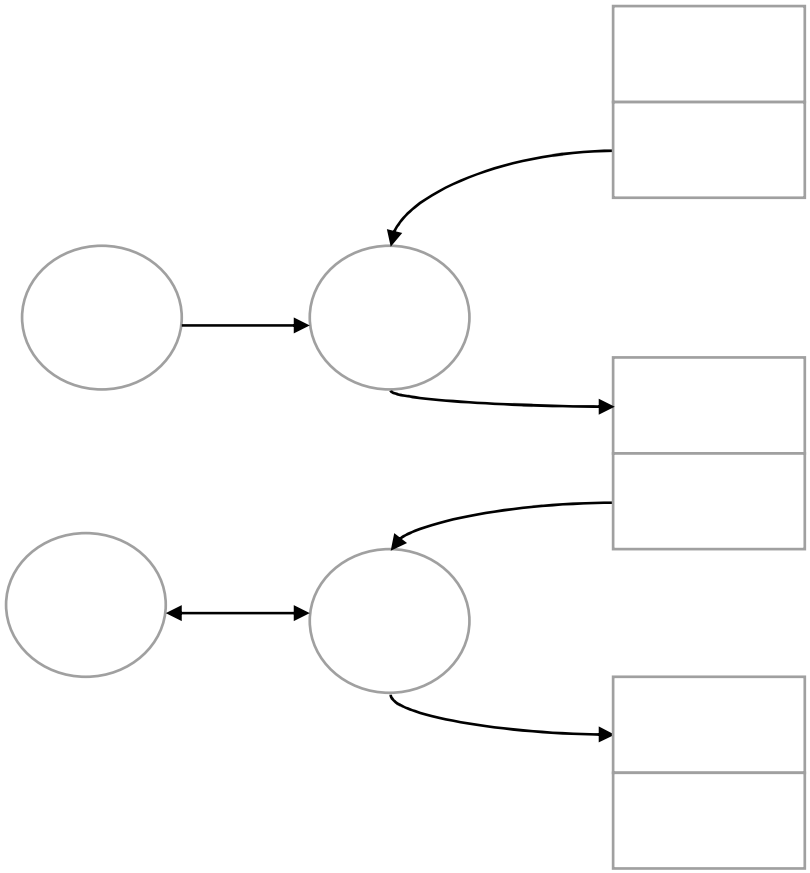
Interface with Physical Layer

- `To_Physical_Layer`
- `From_Physical_Layer`
- Other Functions

Local Operations

- Buffer management
- Error Checking
- Timers
- Sequence Numbering
- ...

Simple Operations



Unrestricted Simplex Protocol

- One direction data transfer
- Transmitting and receiving network layers always ready
- Zero processing time
- Infinite buffer space
- Perfect channel - does not damage or loose frames
- Use no sequence numbers or acknowledgements
- Only use info field in the frame

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 */
  }
}
```

Simple Stop and wait protocol

- Relax the assumption
  - Ability of the receiving network layer to process incoming data infinitely fast -- OR
  - Having infinite buffer space in the receiving data link layer
- How to prevent sender from flooding the receiver
  - If receiver requires t sec to execute From_Physical_layer and To_Network_layer, the sender must transmit at an average rate less than one frame per t sec.
  - Sender may slow sown -- time synchronize - may be too slow
- Receiver Feedback
  - Acknowledgement

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 */
  }
}
```

Simplex Protocol for Noisy Channel

- Relax the assumption
  - Channel is perfect
- Frames received may be
  - damaged ---- Checksum
  - lost ---- Sequence number

- Automatic Repeat reQuest (ARQ)
  - Cumulative Ack
  - Selective Ack

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

Spring 2003

Sliding Window Protocols

- When transit time is large Stop and Wait gives very low line utilization
- Should be able to have more frames in transit

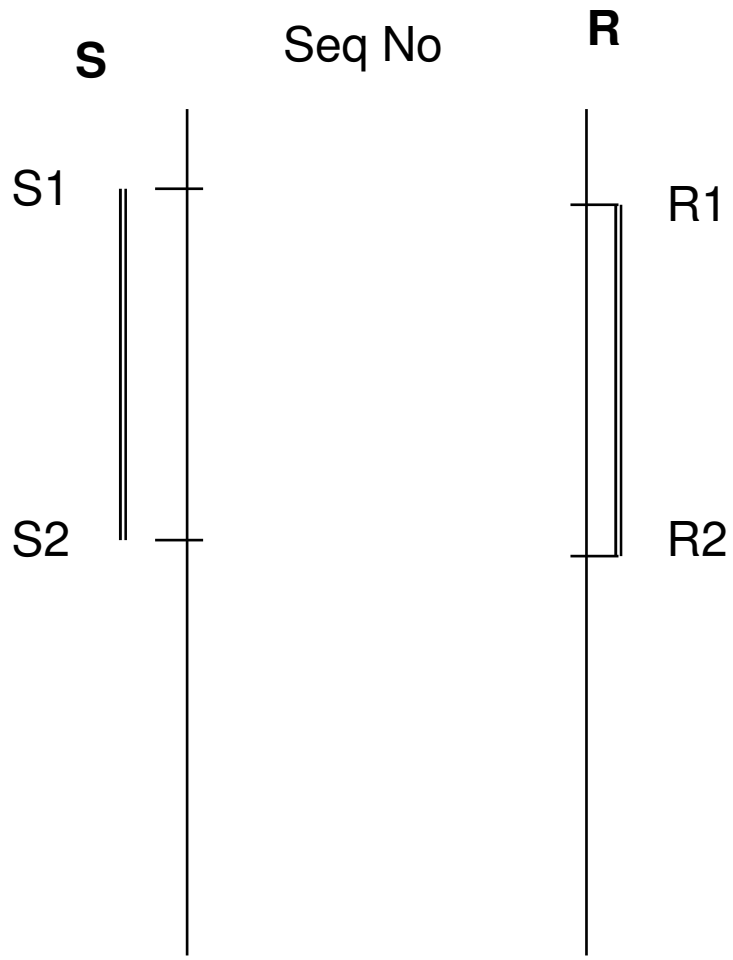
Sliding Windows

S1 – Sender Base No

S2 – Sender High No

R1 – Receiver Base No

R2 – Receiver High No



Sliding Window

- Sender is authorized to send any packet with sequence number between S1 and S2
- Receiver receives any packet in the range R1 – R2.
- If a packet outside this range is received – discard

- Acknowledge any correct packet received in this range.
  - What information to send as acknowledgement

Sliding Windows

- Acknowledgement – Seq No. N
  - Packet N received correctly
  - All packets numbered N or lower received correctly

  - Receive sequence

  - 1 2 3 5 6 7 4 8 9

  - 1 2 3 3 3 3 7 8 9
  - 1 2 3 5 6 7 4 8 9
- Packet received in error
  - Discard packet – treat as if never received
  - Header received correctly
    - Send NAK with seq no of packet

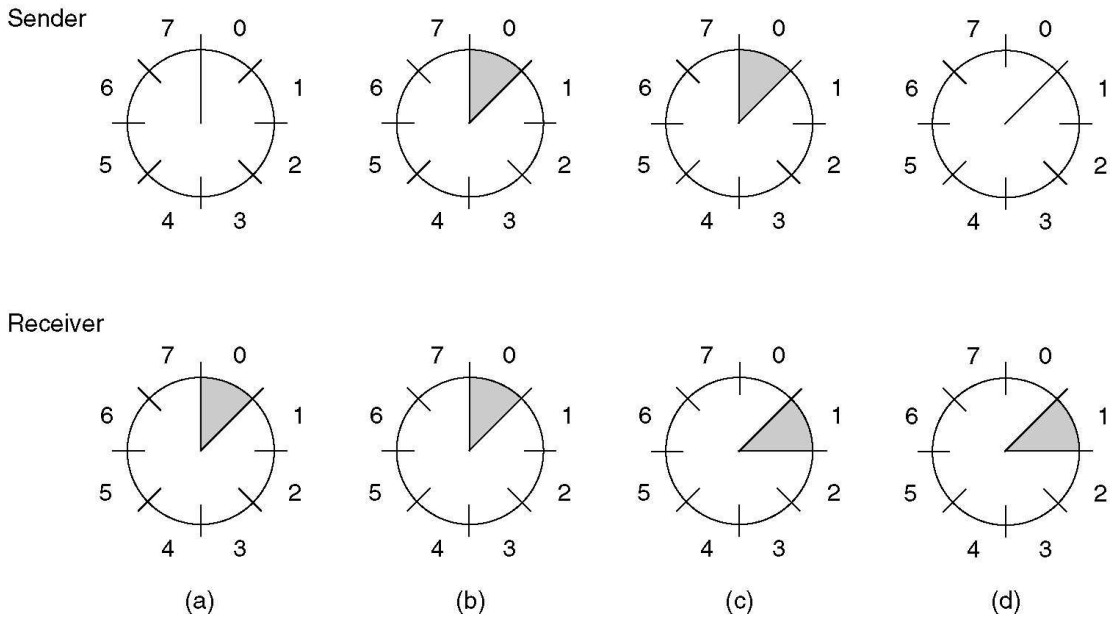
Sliding Window

- Sender
  - Advance S1 and S2
    - Reflecting packets received correctly
  - Lost packets
    - Time out
      - By each packet sent

- What should be the window size?

- Should sender adjust window size? (S2-S1)
- Effect of Packet
  - lost
  - received in error
  - Out of sequence
  - With long delay

Sliding Window Protocols (2)



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.

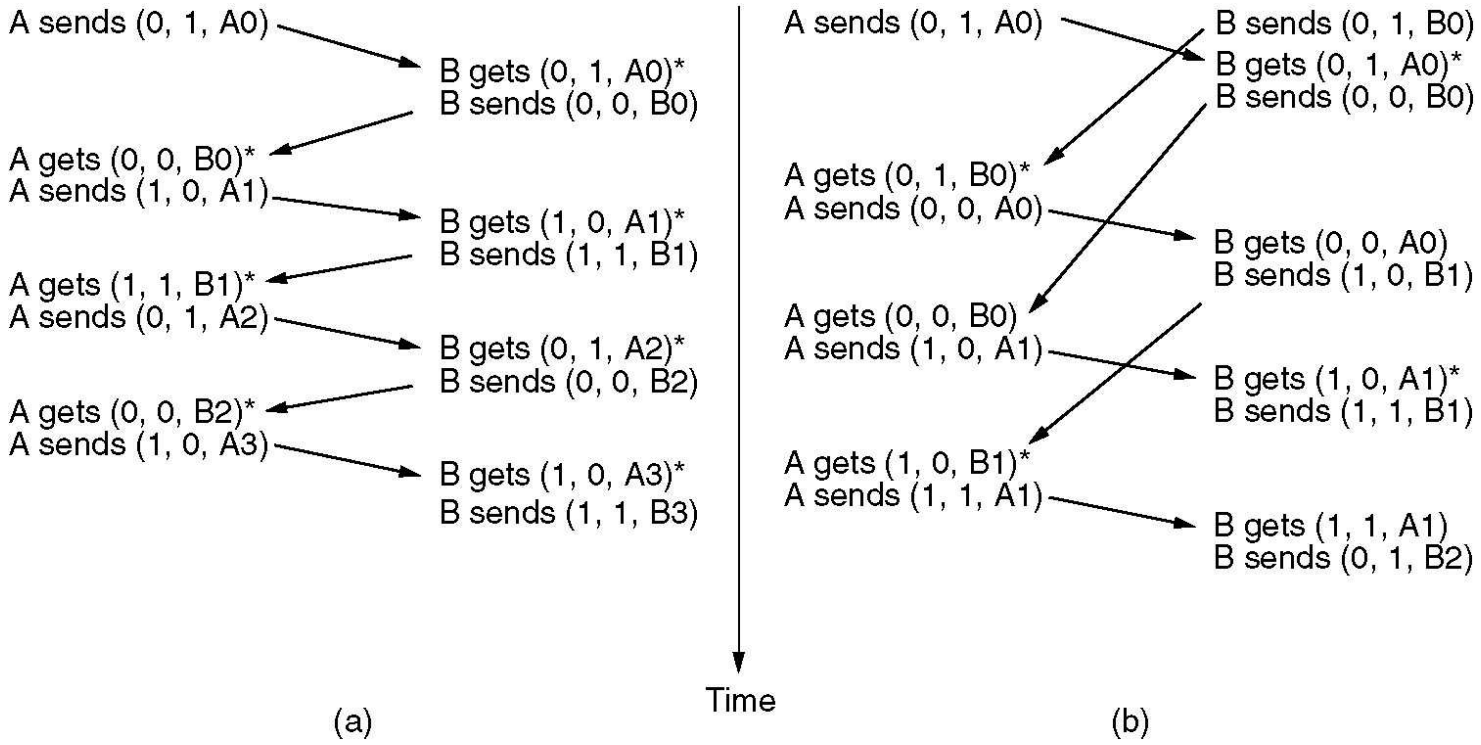
A One-Bit Sliding Window Protocol

```
/* 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 */
```

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)

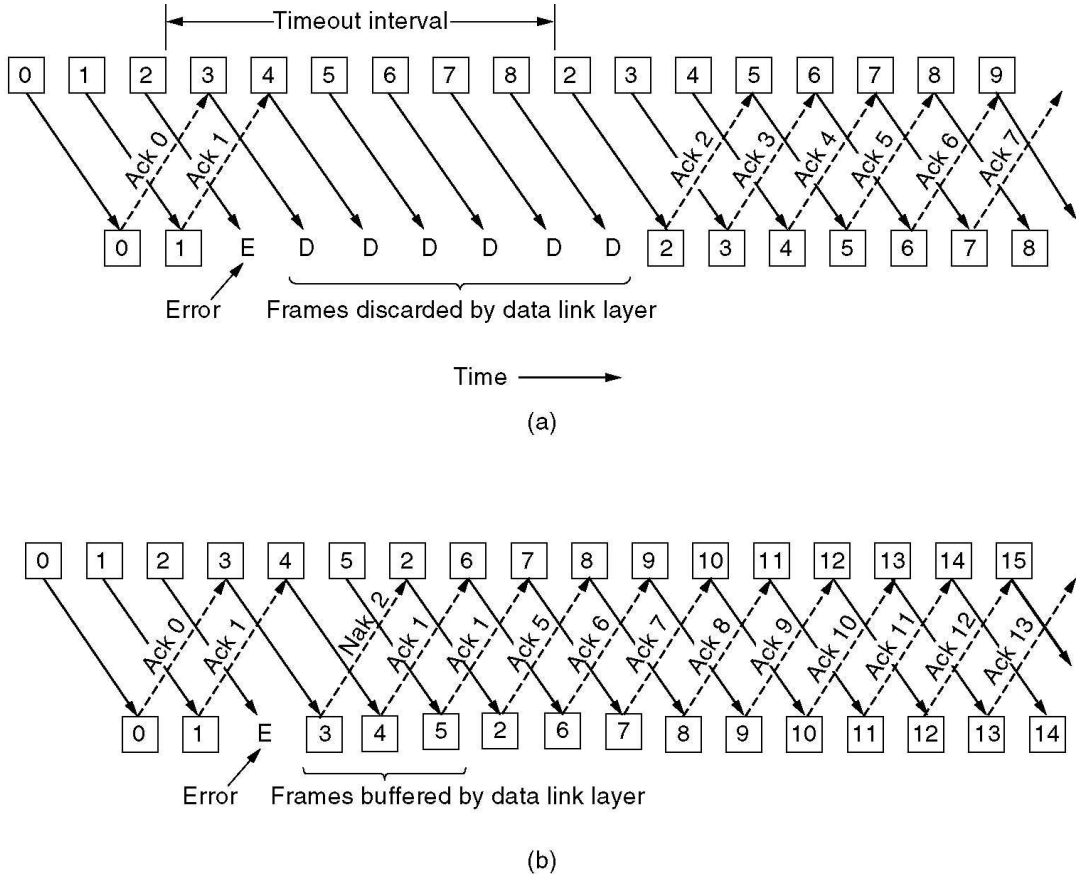


Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

Error Recovery

- Go back N
- Selective Repeat

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 */
}
```

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

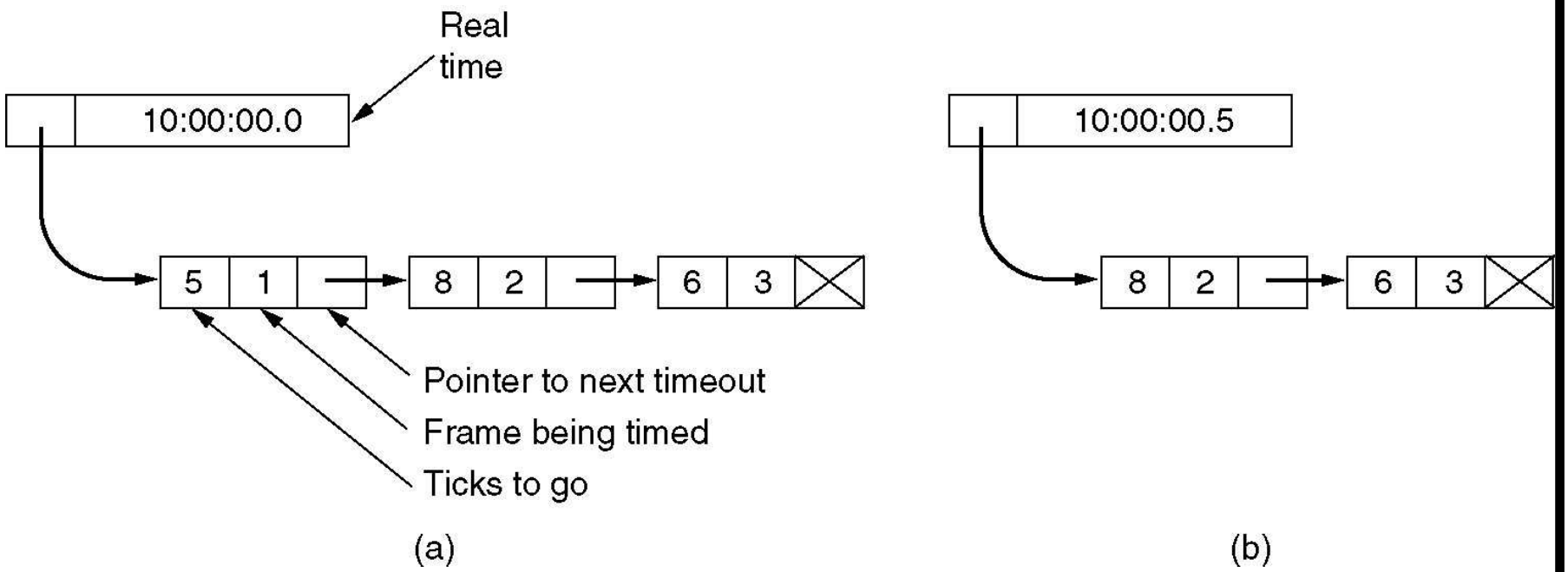
```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  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)) || ((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 */
}
```

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 */
                }
            }
        }
```

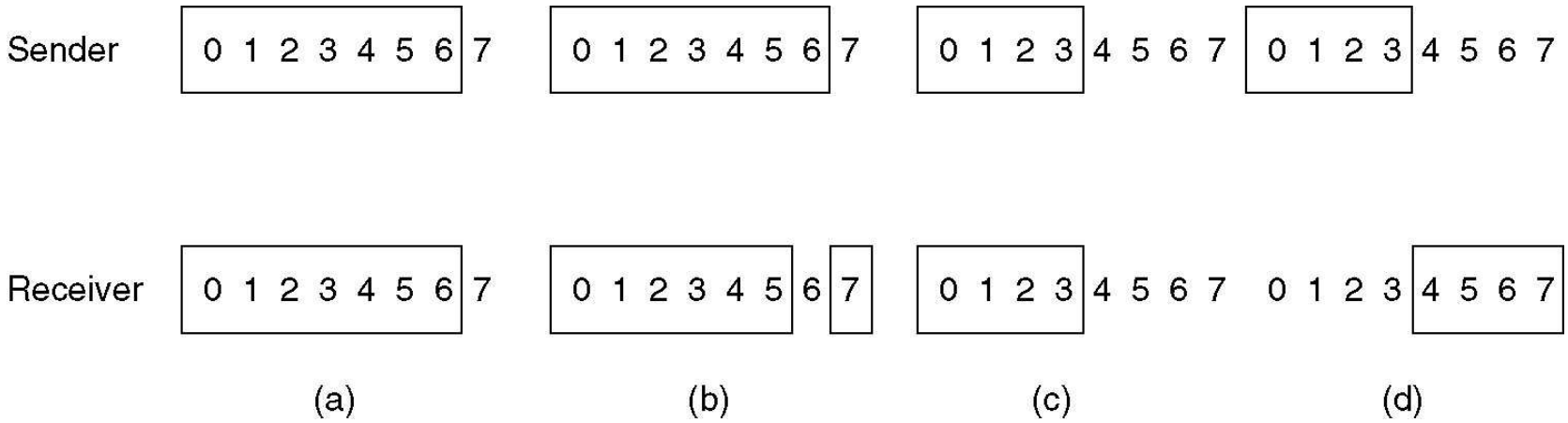
Continued →

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();
 }
}
```

Spring 2003

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.

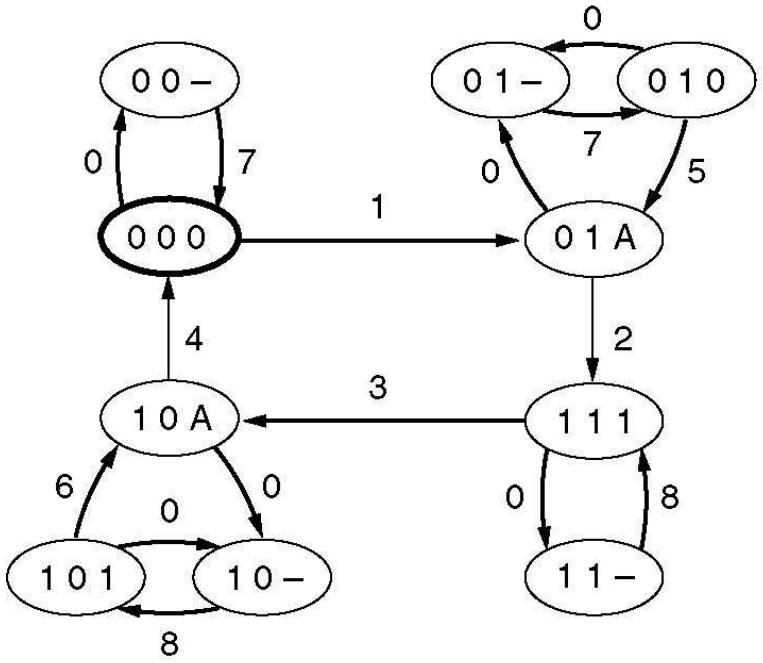
Protocol Verification

Finite State Machined Models

Petri Net Models

Spring 2003

Finite State Machined Models



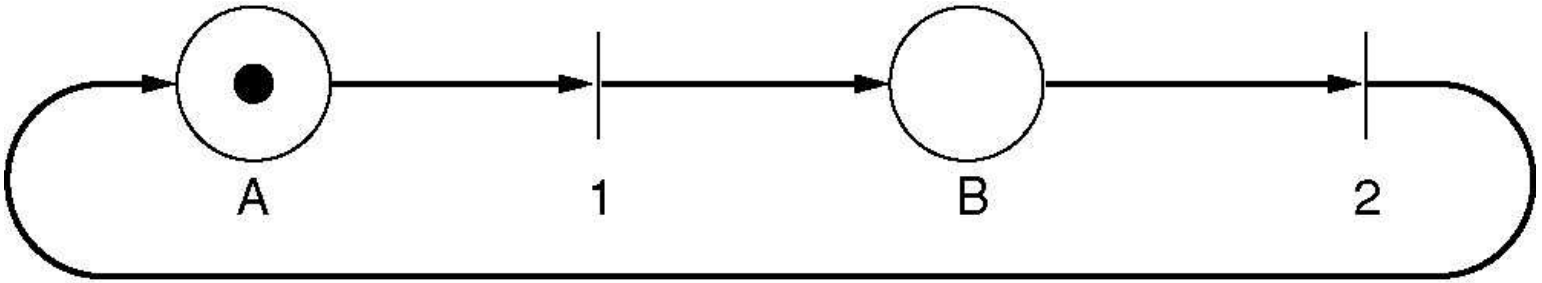
(a)

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)

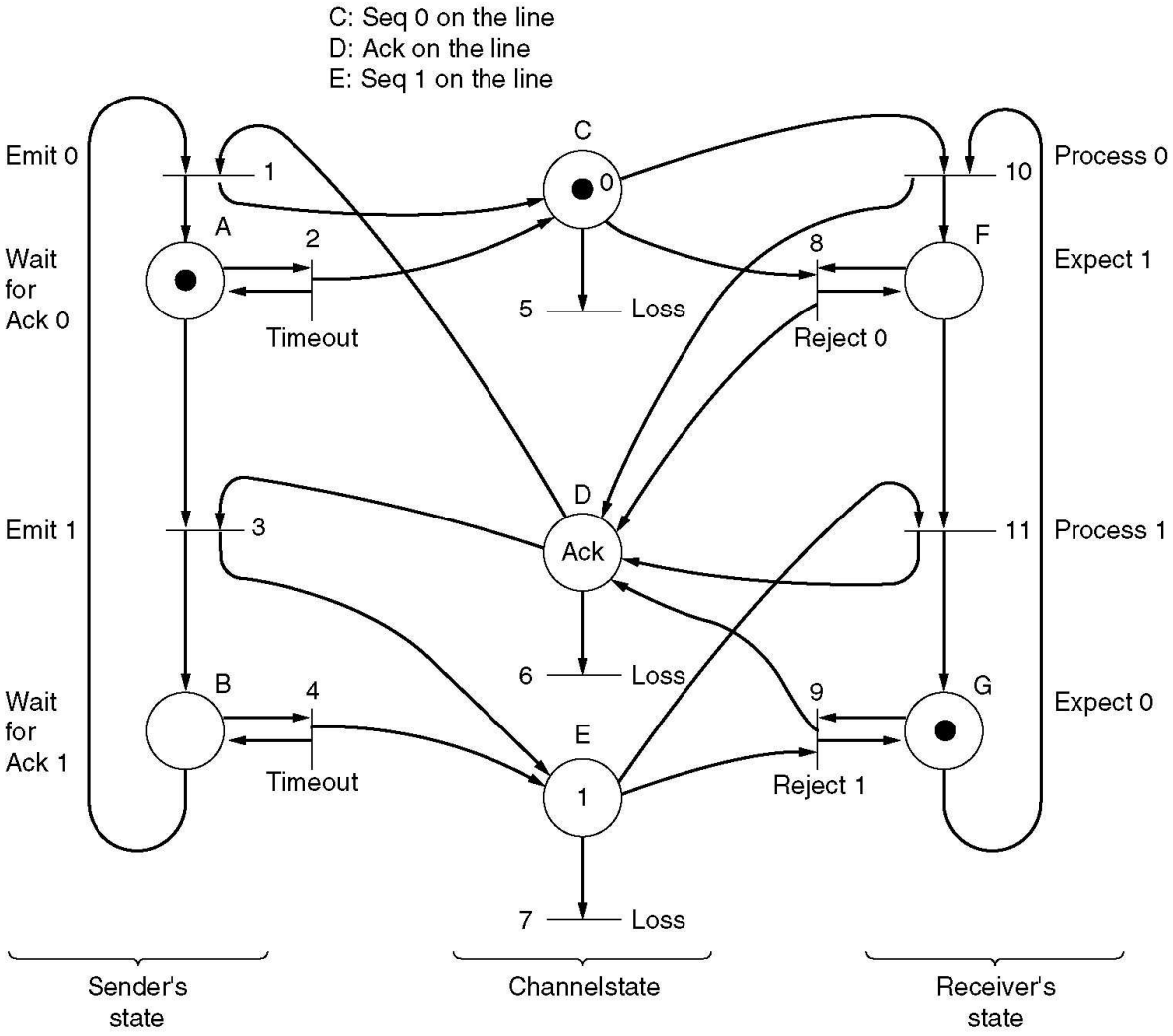
(a) State diagram for protocol 3. (b) Transmissions.

Petri Net Models



A Petri net with two places and two transitions.

Petri Net Models (2)



A Petri net model for protocol 3.

Spring 2003

Data Link Layer Protocols

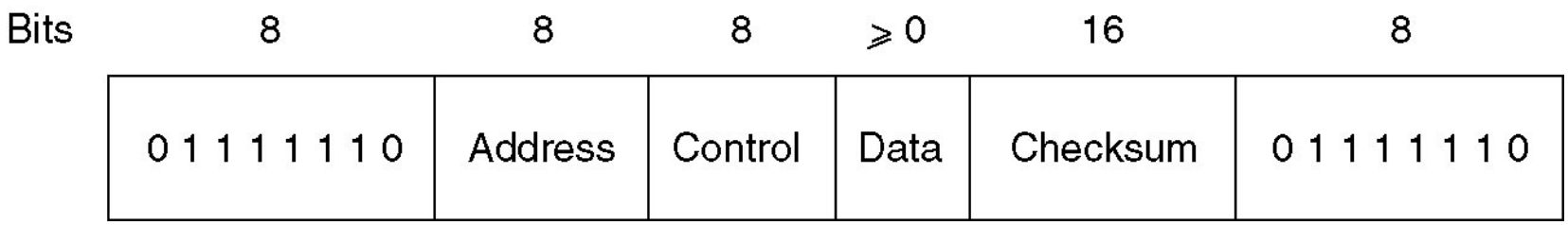
- Bit Oriented
  - SDLC
  - ADDCP
  - HDLC
  - LAP and LAPB
- Byte Oriented
  - SLIP
  - PPP

Example Data Link Protocols

HDLC – High-Level Data Link Control

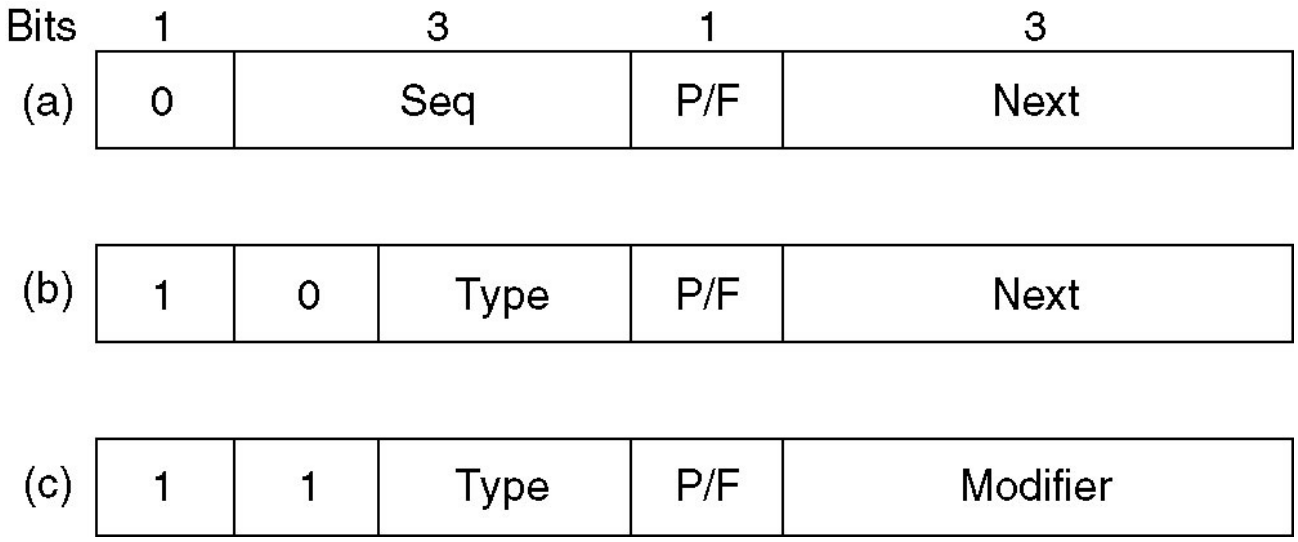
The Data Link Layer in the Internet

High-Level Data Link Control



Frame format for bit-oriented protocols.

High-Level Data Link Control (2)



Control field of

(a) An information frame.

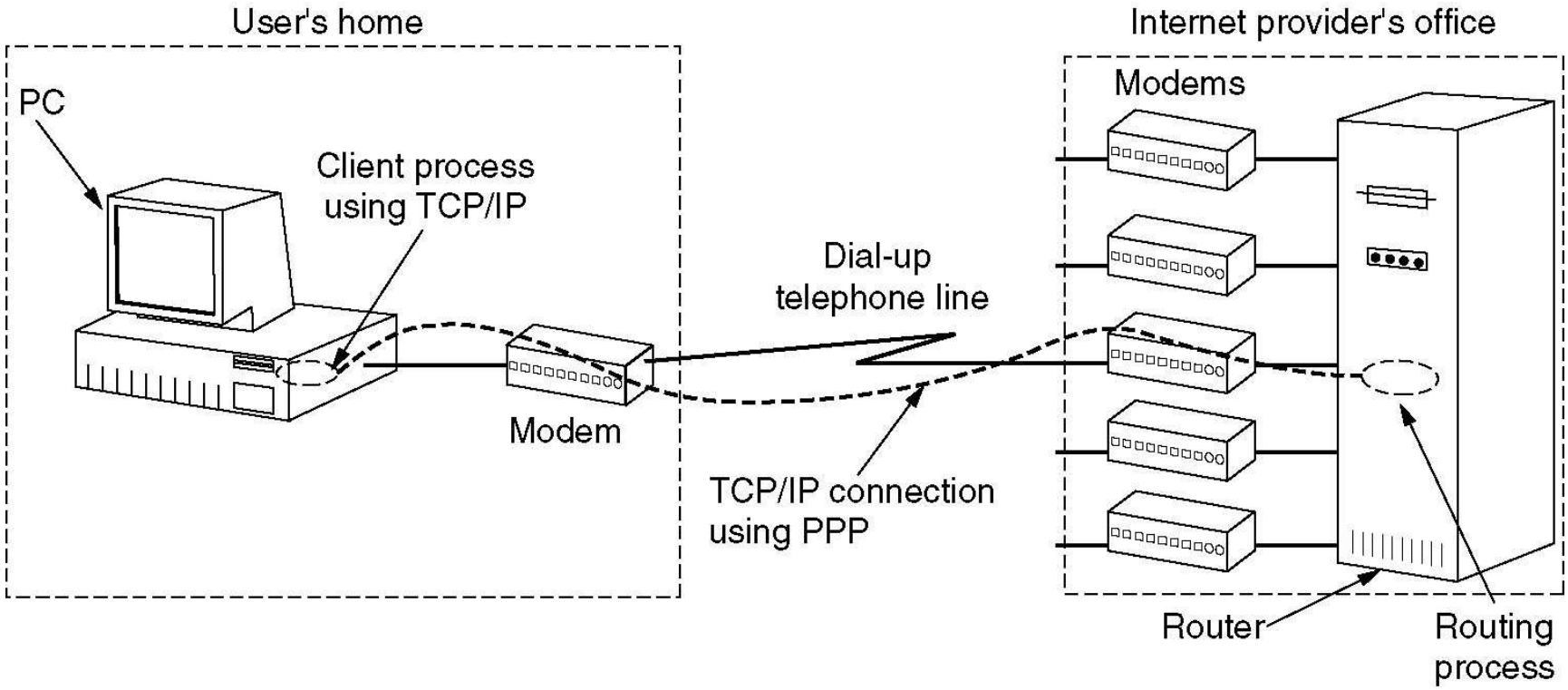
(b) A supervisory frame.

(c) An unnumbered frame.

HDLC Frame Types

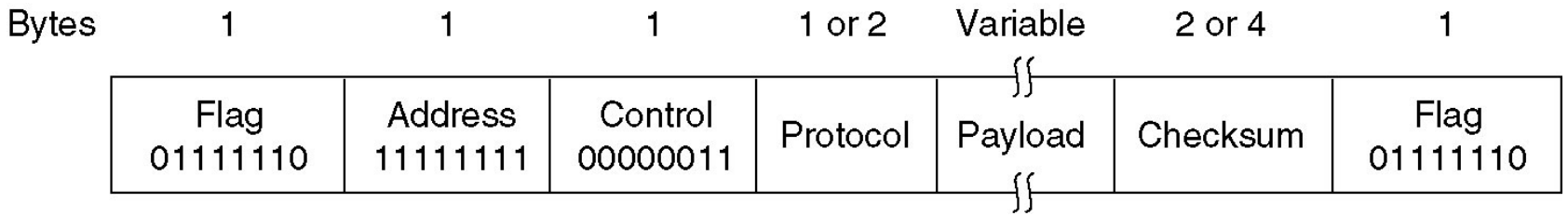
- Type 0 - Acknowledgement - Receiver Ready
- Type 1 - NAC - Reject
- Type 2 - Receiver Not Ready
  - Acks upto but not including Next
  - Sender must stop sending
- Type 3 - Selective Reject
  - Retransmit only the specified frame

The Data Link Layer in the Internet



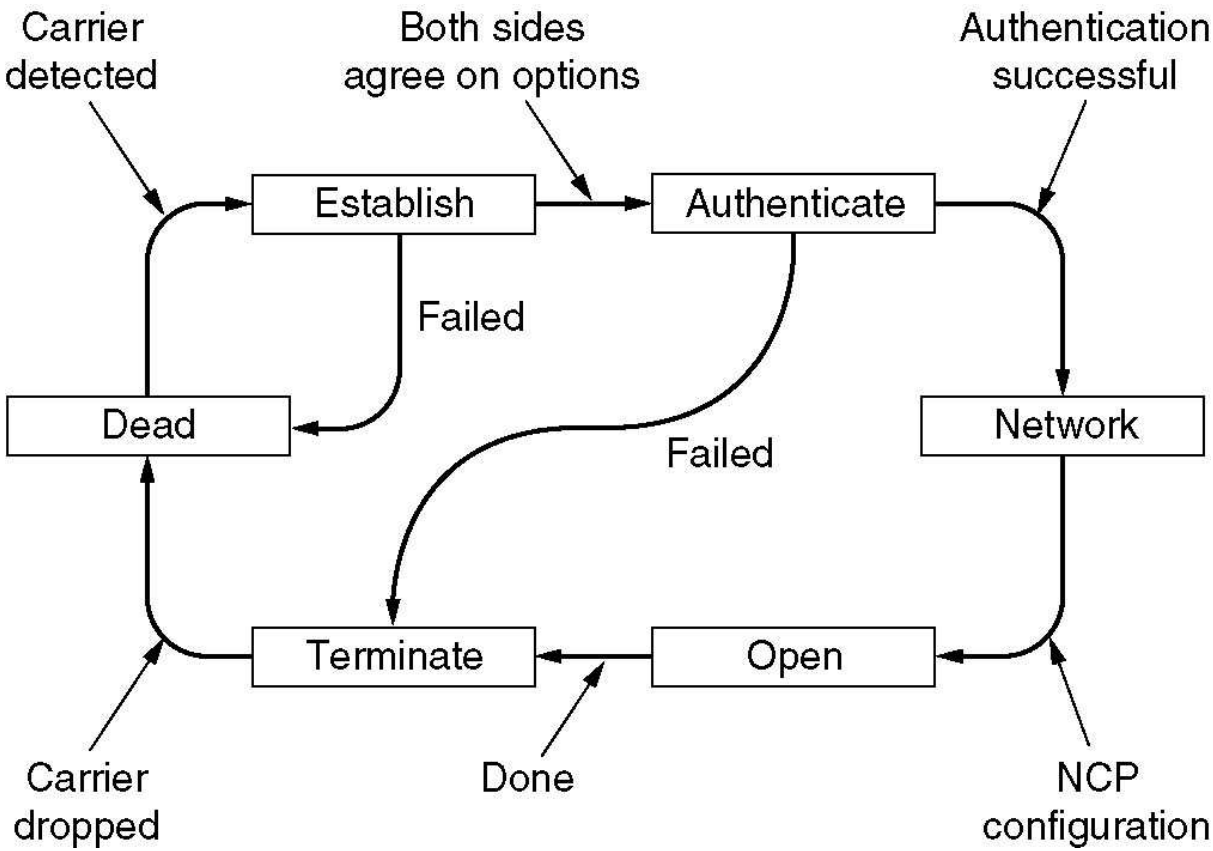
A home personal computer acting as an internet host.

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.

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 frame types.