# Chapter 3

## The Data Link Layer

## Data Link Layer Design Issues

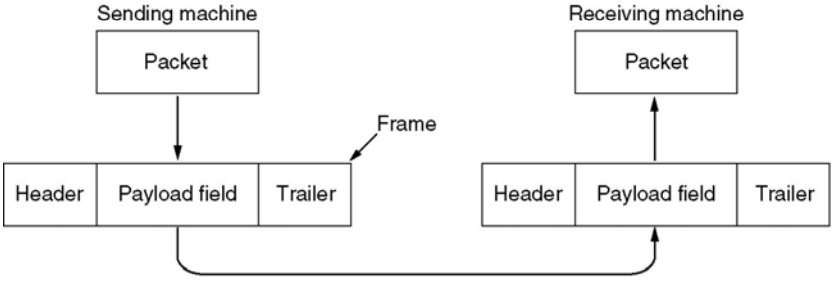
- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control

# Functions of the Data Link Layer

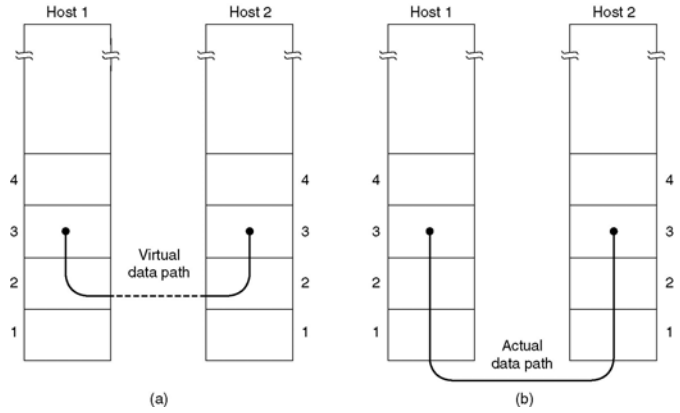
- Provide service interface to the network layer
- Dealing with transmission errors
- Regulating data flow
  - Slow receivers not swamped by fast senders

# Functions of the Data Link Layer (2)

Relationship between packets and frames.

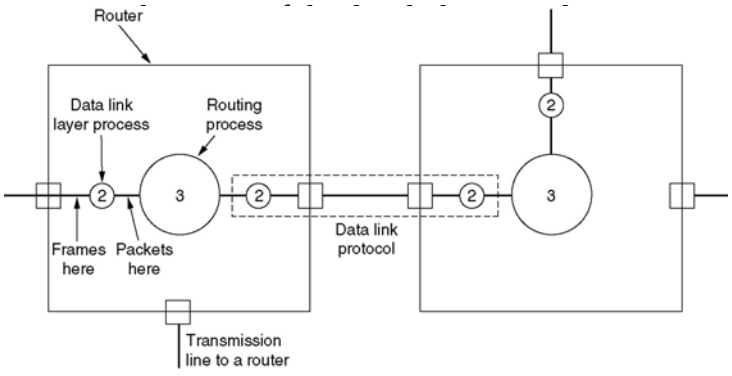


# Services Provided to Network Layer



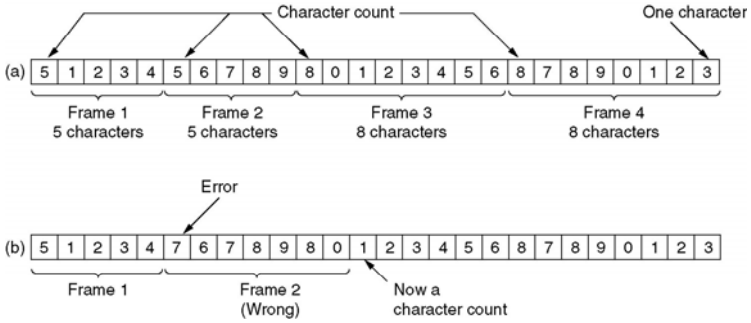

(a) Virtual communication.
(b) Actual communication.

# Services Provided to Network Layer (2)

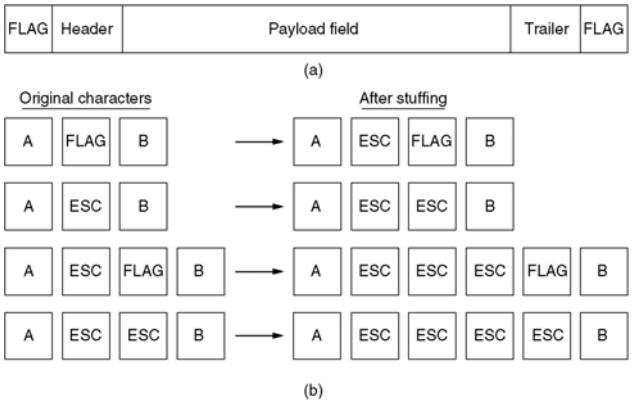


# Framing

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

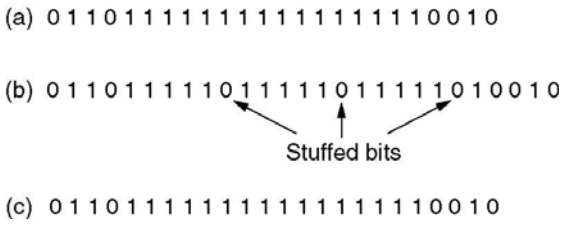



# Framing (2)




(a) A frame delimited by flag bytes.
(b) Four examples of byte sequences before and after stuffing.

# 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.

# Error Detection and Correction

- Error-Correcting Codes
- Error-Detecting Codes

# Error Control

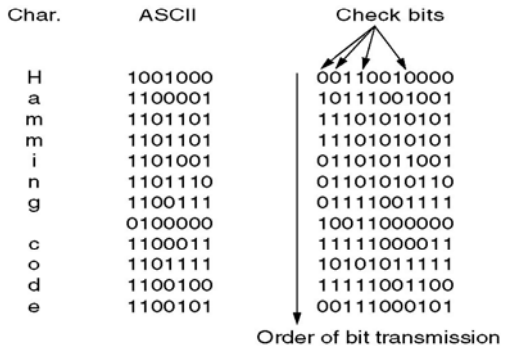
- Acknowledge positive or negative
- Timers and timeouts
- Sequence numbers to avoid duplicates

# Hamming Distance and Error Correction

- No of bits of difference between two codewords
- To find d errors a code with distance d+1 is needed
- To correct d errors 2d+1 distance is needed
  - The original codeword is closer to the wrong one
- Example: parity check
- If we want to design a code to exchange 2^m message we need r redundancy or control bits, so that the total code is m+r bits
  - For each one of the 2^m “legal” message there are n “not-legal” messages with a distance = 1
  - Each message must have n+1 bit combinations dedicated to it
  - We have a total of 2^n possible combination, hence $(n+1)2^m \leq 2^n$
  - We obtain a minimum number of r bits to correct single errors

# Error-Correcting Codes

Use of a Hamming code to correct burst errors.




- Transmission column by column of length K
- Burst error of K bits are corrected since there will be at most 1 single error per raw (codeword)

# Error Detection vs Error Correction

- Used for wired communications (copper, fiber) with low error rate
- Less overhead
- Example (error correction overhead):
  - Error rate: 10^-6 errors/bit
  - With blocks of 1000 bits 10 bits are needed to correct errors
  - 1Mbit of data requires 10000 control bits
- Example (error detection overhead):
  - Error rate: 10^-6 errors/bit
  - 1 parity bit per block
  - Retransmission of 1 extra block each 1000 blocks for this error rate
  - Overhead is 2001 bit each Mbit of data

# Burst Error Detection

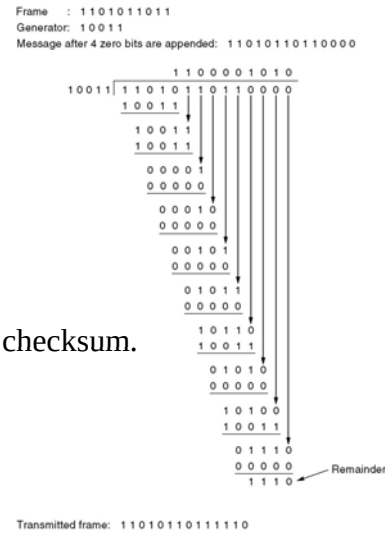
- Block are organized as matrices of kxn elements
- A parity bit is added for each column and another raw is created with parity bits
- The matrix is transmitted raw by raw
- This method can detect burst errors of length n, since only one bit per column is changed

# Polynomial Coding

- Rationale: bits are coefficient of polynomials.
- Source and destination must agree on a generator called G(x)
- The source adds to the m-bit frame to be transmitted a checksum so that modulo (M(x)+checksum, G(x))=0.
- Checksum computation algo:
  1. If r is the degree of G(x), then add r zero-valued bits to the frame such that now it has m+r bits and corresponds to: $x^rM(x)$
  2. Divide G(x) by $x^rM(x)$ (modulo 2)
  3. Subtract the remainder (which contains at most r bits) from $x^rM(x)$. The result is the frame with checksum T(x).

# Error-Detecting Codes

Calculation of the polynomial code checksum.




# Elementary Data Link Protocols

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

# 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.)

```
/* 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
```

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

# 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.)

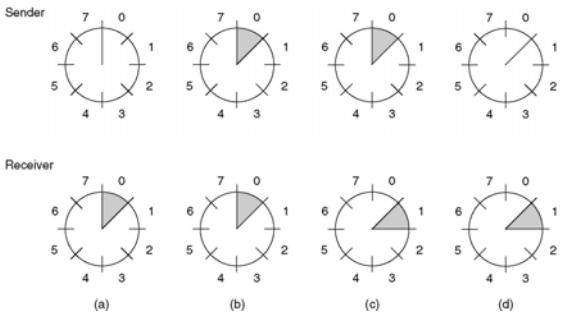
```
void receiver3(void)
{
 seq_nr frame_expected;
 frame r, s;
 event_type event;

 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)




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

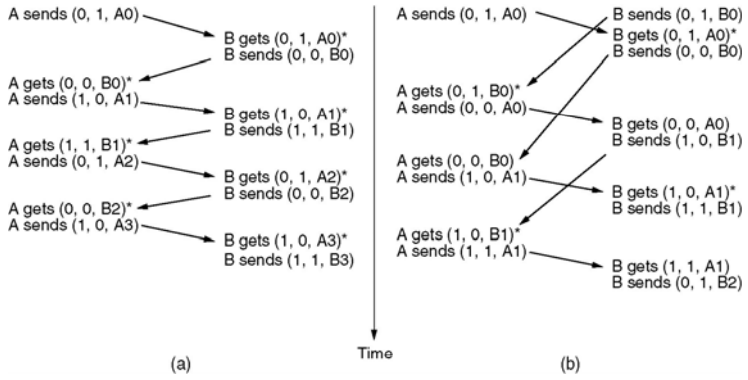
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                       /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;            /* 0 or 1 only */
  seq_nr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

Continued →

# A One-Bit Sliding Window Protocol (ctd.)

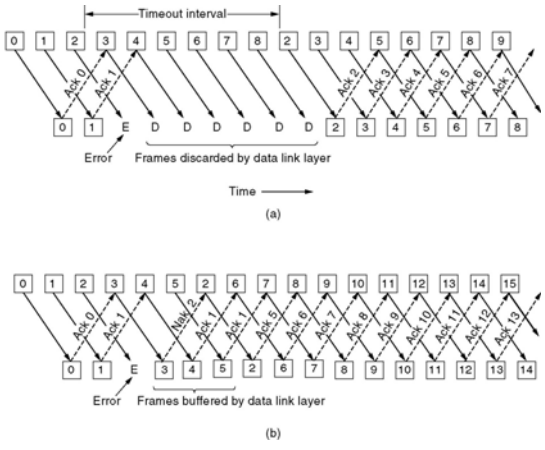
```
  while (true) {
      wait_for_event(&event);           /* frame_arrival, cksum_err, or timeout */
      if (event == frame_arrival) {     /* a frame has arrived undamaged. */
          from_physical_layer(&r);      /* go get it */

          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 */
}
```

Check if seq_no of received frame is in the window

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

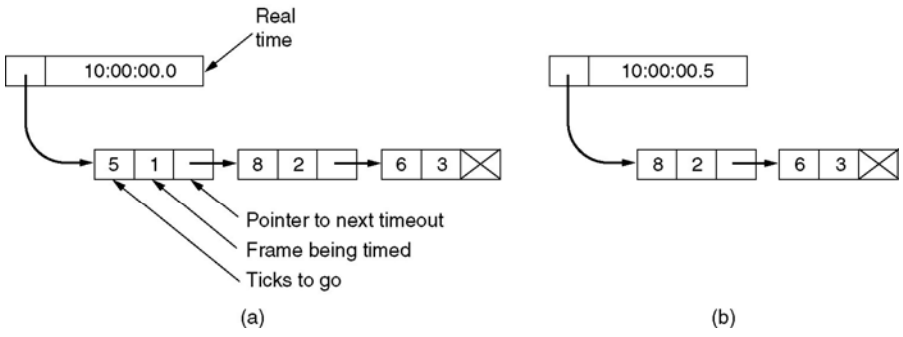
```
              /* Ack n implies n – 1, n – 2, etc.  Check for this. */
              while (between(ack_expected, r.ack, next_frame_to_send)) {
                   /* Handle piggybacked ack. */
                   nbuffered = nbuffered    1; /* one frame fewer buffered */
                   stop_timer(ack_expected); /* 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();
    }
  }
```

# Notes

- There are buffering requirements at the source
  - Frames must be stored until the ack arrives
  - The ack for frame n implies frames n-1, n-2, etc… are acknowledged
- In this protocol we assume there is always traffic to send piggybacking acks
- There is need of multiple timers because each frame can timeout independently from others

# Sliding Window Protocol Using Go Back N (2)

Simulation of multiple timers in software.



## A Sliding Window Protocol Using Selective Repeat

- Main features:
  - Source has a window of size between 0 and MAX_SEQ
  - Destination has a buffer of fixed size MAX_SEQ
  - Usage of ack_timeout to avoid problems with piggypacking
- Destination keeps track of frame for which nak has been sent to avoid multiple retransmissions
  - *no_nak* is true if no NAK has been sent for *frame_expected*
  - If a wrong frame arrives after nak has been sent and lost, *no_nak* will be set to true and the auxiliar timer is started
  - Upon timer expiration an ACK is sent.

## 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 */
}
```

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)

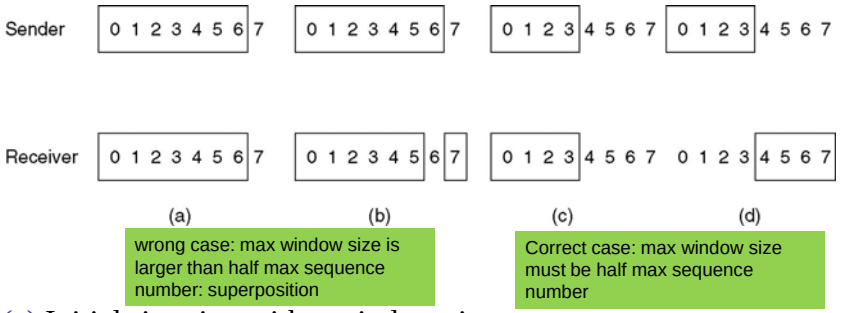
```
        if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next frame to send))
            send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

        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();
 }
}
```

NOTA: il timer ack_timeout serve per gestire il caso in cui non si possa fare piggybacking. Ack timeout deve essere minore del timer dei dati per assicurare che I frames ricevuti trasmettano ack in tempo

## Implications of Non-sequential Reception

- Example: 3 bit sequence number

## 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.

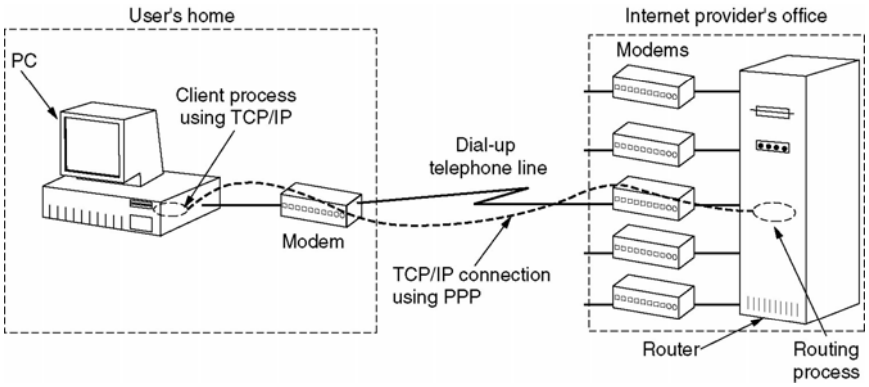
(c) Initial situation with a window size of four.

(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

# The Data Link Layer in the Internet



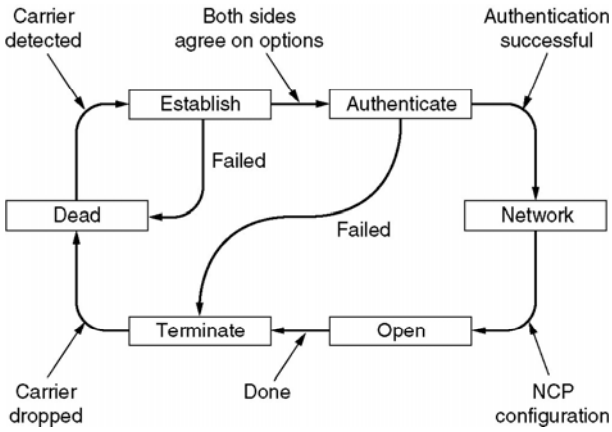

# PPP – Point to Point Protocol

The PPP full frame format for unnumbered mode operation.

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

- Byte level stuffing
- Address and control field are constant
  - No data link addresses
  - Control field indicates no frame number (not used in PPP)
- Protocol type is LCP, NCP, IP, IPX
  - LCP establishes link parameters
  - NCP establishes dynamic IP addresses

# 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) |