Chapter 3

The Data Link Layer

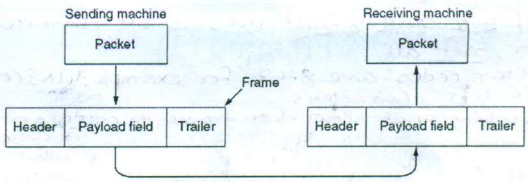
Data Link Layer Design Issues

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

Functions of the Data Link Layer

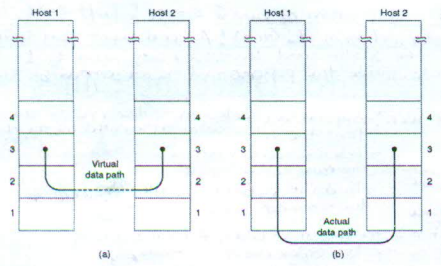
- Provide service interface to the network layer
- Dealing with transmission errors
- Regulating data flow
  - Slow receivers not swamped by fast senders

Functions of the Data Link Layer (2)



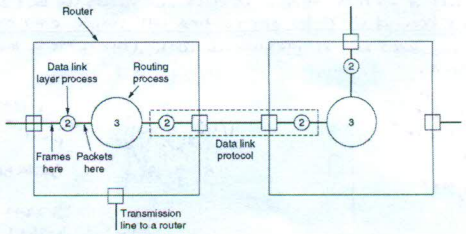
Relationship between packets and frames.

Services Provided to Network Layer



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

Services Provided to Network Layer (2)



Placement of the data link protocol.

Framing ⇒ Breaking up bit streams into frames

↳ intuitive way like our ordinary text writing ⇒ insert time gaps between frames

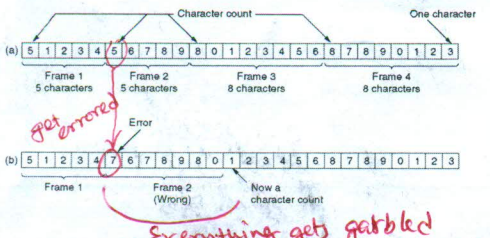
⇒ Problem: Networks rarely make any guarantees about timing, so it is possible these gaps might be squeezed out or other gaps might be inserted during transmission.

Remedy ⇒ four methods of framing

1. Character count
2. Flag bytes with byte stuffing
3. Starting and ending flags, with bit stuffing
4. Physical layer coding violations ⇒ If encoding in physical medium contains some redundancy. For example, in some network 1 data bit is encoded by 2 physical bits. Here, high-low transition refers to 1 and low-high refers to 0, high-high and low-low are unused ⇒ can be a delimiter

8/26

Framing



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

got errored

Everything gets garbled (Synchronization gets lost but? sender & receiver)

⇒ Possible sol^n: Checksum → The receiver will only know that the frame is bad, but, no way to know where the next frame starts

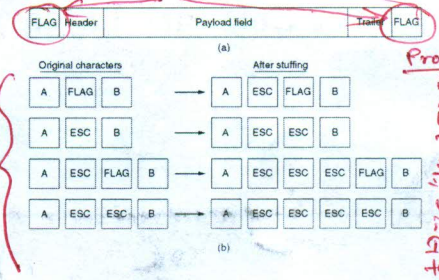
⇒ Another possible sol^n: Sending a frame back to the src asking for a retx → still no way to know how many bits to skip

Framing (2)

Byte stuffing or character stuffing ⇒ omits the synchronization problem

In past ⇒ Starting and ending bytes were different

Now ⇒ Both are same



(a) A frame delimited by flag bytes.

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

Problem: what happens if a flag appears in original data (may happen in binary files)

Sol^n: Insert a special "escape" character just before the flag when it appears in data. In case of destuffing, the extra added escape characters are removed.

New problem: what happens if the "escape" character itself appears in the data??

Sol^n: Again, it is stuffed and destuffed in the same way with an additional "escape" character.

Different situations can occur here.

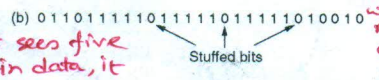
Framing (3)

Problem of byte stuffing ⇒ It uses 8-bit characters. However, not all character codes are 8-bit. For example, UNICODE uses 16-bit characters

Sol^n: Bit stuffing that allows data frames to contain an arbitrary number of bits and allows character codes with an arbitrary number of bits per char.

Flag: 01111110

Stuffing: If sender sees five consecutive 1's in data, it automatically stuffs a 0 into the ongoing stream



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.

Destuffing: If the recvr sees five consecutive 1's followed by a 0, it automatically destuffs or deletes the 0

01111110 → Stuffing (if in data) → 01111101010 → Destuffing → 01111110

Error Detection and Correction

- Error-Correcting Codes → For mostly highly noise medium (wireless)
- Error-Detecting Codes (⇒ Retx) → For mostly low-noise medium (copper, fiber).

Telephone system ⇒ i) switches, ii) Interoffice trunks, iii) local loops

Digital → Analog

⇒ Errors are more common in analog than digital. We may experience this long, as replacing local loops are highly costly. Also, wireless case is far more error prone. So, we need to know how to deal with them.

⇒ Errors may come in bursts or singly.

⇒ Burst: Gives errors in less # of packets

Ex: if error rate is 0.001 per bit and block size is 1000 bits, then most of the blocks will be errored for singly error. For burst errors of 100 bits, then 1 or 2 out of 100 packets will be in error

⇒ Problem of burst error: Difficult to correct

⇒ Error correction: To add redundant or check bits for m bits data. Total n bits. ($n = m+r$) → length of a code word.

⇒ Hamming distance: # of bit positions at which two code words differ.

⇒ To detect d errors, Hamming distance needs to be $(d+1)$

⇒ To correct d errors, Hamming distance needs to be $(2d+1)$

⇒ Calculation of lower bound on r for correcting all single errors

Each of 2^m legal messages has n illegal code words (can get it be inverting each bit pos^n) at a distance 1 from it.

So, each legal message (total 2^m) has $(n+1)$ bit patterns dedicated to it.

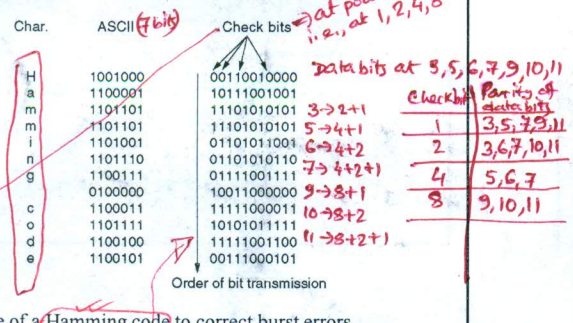
So, $(n+1)2^m \le 2^n$

so, $(m+r+1)2^m \le 2^{m+r}$

$\therefore (m+r+1) \le 2^r$

gives the lower limit on r.

Error-Correcting Codes



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

Use of a Hamming code to correct burst errors.

* check bits ⇒ Give even parity: # of 1's needs to (including itself) be even

⇒ when a code word arrives at a receiver, it initialized a counter to zero. It then add the pos^n k of an errored check bit to the counter.

If counter = 0, then no error

If counter > 0, then the counter gives pos^n of the single error.

⇒ To handle burst error of at most k length, k consecutive code words are arranged as a matrix, one code word per row. Data is transmitted one column at a time. So, in case of k-bits error (in burst), * each column will be affected by only one bit. At the receiver, the matrix is reconstructed, one column at a time. ⇒ Needs (kr) check bits to make blocks of km data bits immune to a single burst error of length k or less.

Error-Detecting Codes

Basic idea: Append a checksum to the end of the frame in such a way that the polynomial represented by the checksummed frame is divisible by G(x)

– when a receiver gets the checksummed frame, it (receiver) divides it by G(x) and check the remainder whether it is 0 or not

(checksummed frame)

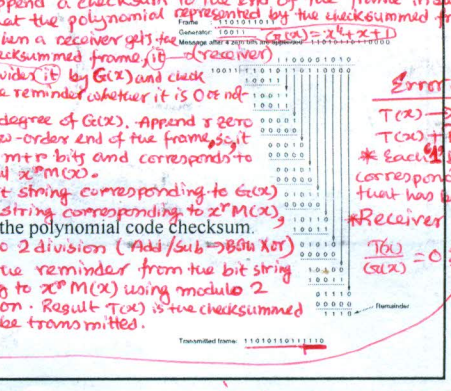
Algo:

1. Let r be the degree of G(x). Append r zero bits to the low-order end of the frame, so it now contains m+r bits and corresponds to the polynomial $x^r M(x)$.
2. Divide the bit string corresponding to G(x) into the bit string corresponding to $x^r M(x)$.

Calculation of the polynomial code checksum.

3. Subtract the reminder from the bit string corresponding to $x^r M(x)$ using modulo 2 subtraction. Result T(x) is the checksummed frame to be transmitted.

Using modulo 2 division (Add/Sub ⇒ Bitwise XOR)



Error detection

T(x) → Sent → no error

T(x) + E(x) → Received

* Each 1 bit in E(x) corresponds to a bit that has been inverted

* Receiver: $\frac{T(x)+E(x)}{G(x)}$

$\frac{T(x)}{G(x)} = 0$; $\frac{E(x)}{G(x)}$ reports error

Cond^n 1: Single-bit error: $E(x) = x^i$; Always detected if G(x) contains 2 or more terms.

Cond^n 2: Two isolated single bit errors: $E(x) = x^i + x^j$, where $i > j$

$E(x) = x^j (x^{(i-j)} + 1)$, ⇒ cond^n: G(x) does not divide $x^k + 1$ for any k up to the maximum value of i−j (i.e., up to the max frame length)

Ex: $x^{15} + x^{14} + 1$ does not divide $x^k + 1$ for $k < 32,768$. (as max frame length)

Cond^n 3: Odd # of bit errors: E(x) contains an odd # of terms.

⇒ No polynomial having an odd # of terms was $(x+1)$ as a factor in the modulo 2 system. So, by making $(x+1)$ as factor of G(x) we can catch all errors consisting an odd # of inverted bits

⇒ proof of no polynomial with an odd # of terms has $(x+1)$ as its factor in modulo 2 system.
→ Let the polynomial can have it. So, $E(x) = (x+1)\,Q(x)$; Now, if $x=1$, then $E(1) = (1+1)\,Q(1) = 0\,Q(1) = 0$
On the other hand, if $E(x)$ contains an odd # of terms putting $x=1$ will give $E(1) = 1$ ← contradiction!

Cond.4 : A polynomial code with r check bits detects all burst (consecutive) errors of length ≤ r.
→ A burst error of length k can be represented as $x^i(x^{k-1} + \dots + 1)$, where i determines how far from the right hand end of the received frame the burst is located. Now, if $G(x)$ contains an x^0 term, then it will not have x^i as a factor.
Besides, if the degree of the polynomial in parenthesized expression is less than the degree of $G(x)$, then the reminder can never be zero.
So, if the degree of the parenthesized expression is less than the degree of $G(x)$, i.e., r, then the the reminder can never be zero. So, it will never divide the error by $G(x)$ with degree ≤ r.

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;
```

control fields (frame header)

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.

seq #

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 */
  }
}
```

NL → PL

PL → NL

No flow control, No error control

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 */
  }
}
```

flow control

like ack

A Simplex Protocol for a Noisy Channel

```
/* 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 positive acknowledgement with retransmission protocol.

ACK and its processing

Retransmission

PAR (Positive Acknowledgement with Retransmission)
or ARQ (Automatic Repeat Request)

A Simplex Protocol for a Noisy Channel (ctd.)

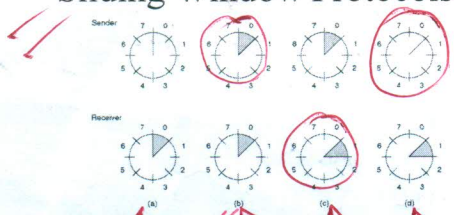
```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  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

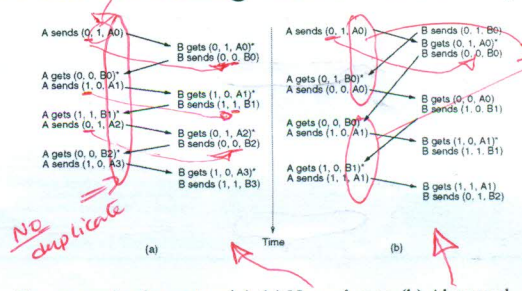
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                    /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;         /* 0 or 1 only */
  seq_nr frame_expected;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

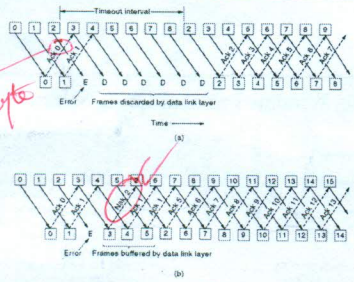
A One-Bit Sliding Window Protocol (2)



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.

In sessional

A Protocol Using Go Back N



last received byte

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

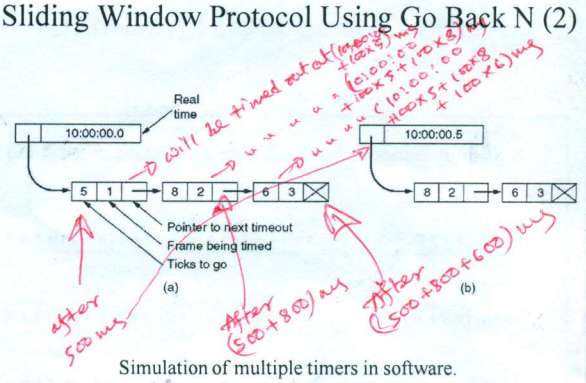
```
while (true) {
 wait_for_event(&event);            /* four possibilities: see event_type above */

 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.

⇒ Because Protocol 5 has multiple outstanding frames, it logically needs multiple timers, one per outstanding frame. However, all the timers can be simulated in s/w using a single h/w clock that causes interrupt periodically. Pending timeouts form a linked list telling the number of clock ticks (at every 500 ms in the example) until the timer expires, the frame being timed, and a pointer to the next node.

Sessional

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)

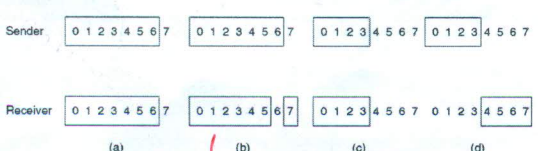
```
while (true) {
  wait_for_event(&event);               /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:           /* accept, save, and transmit a new frame */
         nbuffered = nbuffered + 1;     /* expand the window */
         from_network_layer(&out_buf[next_frame_to_send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

A Sliding Window Protocol Using Selective Repeat (5)



If the ack sent by the receiver gets lost here, then and then the sender sends pckt 0 after timeout, then there might arise a problem. Here the receiver will send an ack 6 thinking that 0 to 6 have already been sent

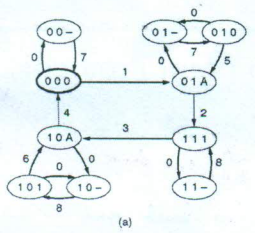
(a) Initial situation with a window size seven.
(b) After seven frames sent and received, but not acknowledged.
(c) Initial situation with a window size of four.
(d) After four frames sent and received, but not acknowledged.

(This happens as the prev acks got lost and the sender has no way to understand which pckt 0 (either new or old) has been acked). This problem arises as there is an overlap between earlier window and new window when an window advances. Its remedy is to make sure that there is no such overlap. To do so, the window size needs to be $\frac{MAX\_SEQ + 1}{2}$.

Protocol Verification

- Finite State Machined Models
- Petri Net Models

Finite State Machined Models

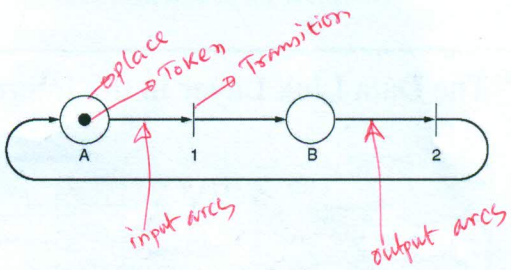


Transition	Who runs?	Frame accepted	Frame emitted	To network layer
		(frame lost)		
0	–			–
1	R	0	A	Yes
2	S	A	1	–
3	R	1	A	Yes
4	S	A	0	–
5	R	0	A	No
6	R	1	A	No
7	S	(timeout)	0	–
8	S	(timeout)	1	–

(b)

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

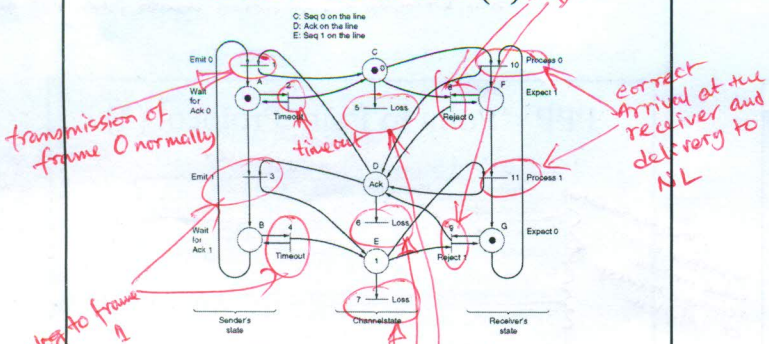
Petri Net Models



A Petri net with two places and two transitions.

place ⇒ represents a state, which the system may be in
Token ⇒ indicates where the system currently in
Transition ⇒ has o/more input arcs, and o/more output arcs
⇒ A transition is enabled if there is at least one input token in each of its input places. Any enabled transition may fire at will, removing one token from each input place and depositing a token in each output place. If # of input and output arcs differs, tokens will not be conserved.

Petri Net Models (2)



A Petri net model for protocol 3.

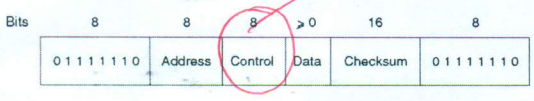
Data frame 0 with wrong seq #
Data ⇒
transmission of frame 0 normally
timeout
correct arrival at the receiver and delivery to NL
Analogy to frame 1
loss of frame 0
loss of ack
loss of frame 1

Example Data Link Protocols

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

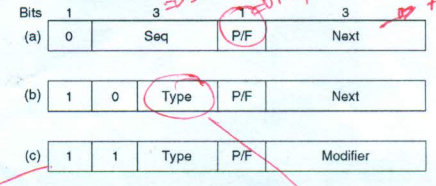
⇒ Both ⇒ bit-oriented
⇒ use bit stuffing

High-Level Data Link Control



Frame format for bit-oriented protocols.

High-Level Data Link Control (2)



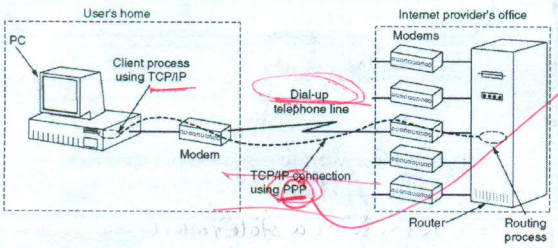
Control field of
(a) An information frame.
(b) A supervisory frame.
(c) An unnumbered frame.

3 kinds of frames

⇒ 3 bits seq #
⇒ P/F ⇒ Poll/final
⇒ polls a group of terminals when set. Remains set to P except for the last (or final) frame, which is set to F.
Piggybacked ACK (next frame expected)
Dissimilar to the protocols, which say the last frame received correctly
0 → Ack or acknowledgment
1 → Reject (NAK) (similar to Protocol 5)
2 → Receive Not Ready
3 → Selective Reject (similar to Protocol 6)

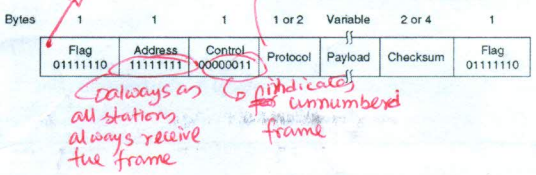
⇒ can be used for both carrying data for unreliable connection and for control purpose.

The Data Link Layer in the Internet



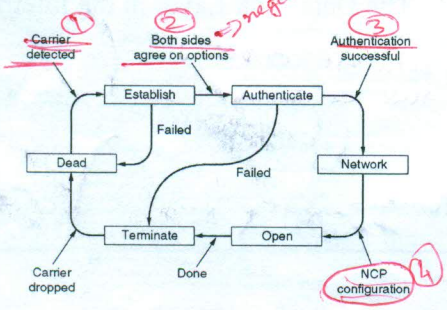
A home personal computer acting as an internet host.

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. (11 frames)