- error-free Xmission facility; uses data & ack. frames
  - error correction/detection (damaged frames)
  - retransmission (lost frames); (duplicate frames)
  - can provide different service classes
  - flow ctrl per link
  - broadcast networks: **3** - FDX data handling
    MAC sublayer

THE DATA LINK LAYER

- channel: bits delivered FIFO
- Acks in DLL: an optimization
  - Transport layer can always wait for message to be acked, but
    - * for high error rates (eg wireless) this is very inefficient
      eg M broken into f frames, error rate/frame = r
      P(error-free Xmission of M) = $(1-r)^f$
    - * for low error rates (eg fiber), OK

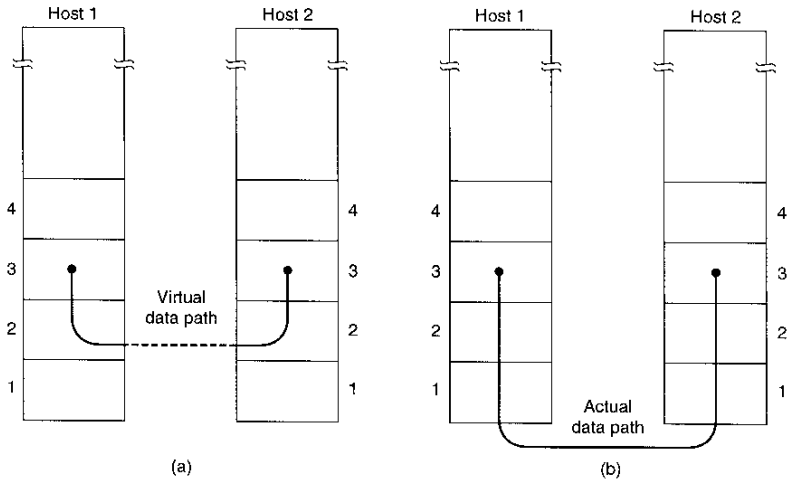
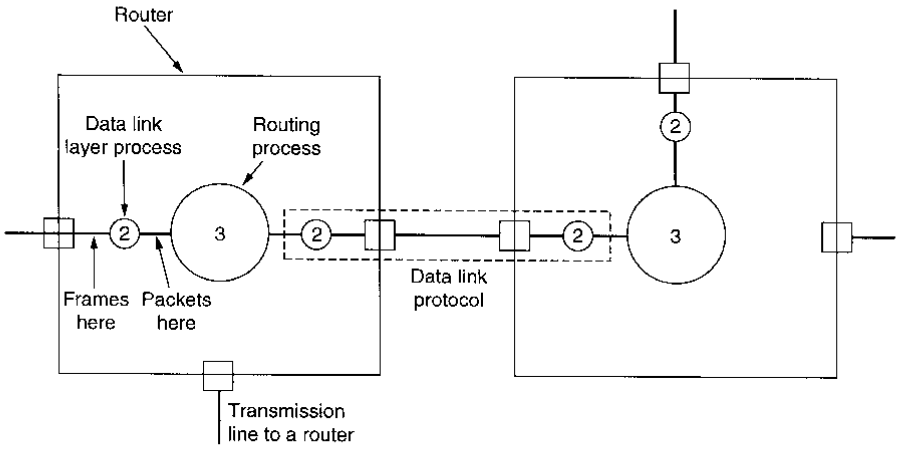


Fig. 3-1. (a) Virtual communication. (b) Actual communication.

- Services to network layer
  - No ack, CL: when low error rate, recovery left to higher layers. Needed for real-time traffic
  - Ack, CL: if ack not recd. by timeout, retransmit frame
  - Ack, CO: frames are numbered, each delivered exactly once (not so for Ack-CL) in FIFO



WAN subnet example

Fig. 3-2. Placement of the data link protocol.

- Frame arrives, HW verifies checksum, passes frame to DLL
- DLL SW verifies this is the frame expected, gives packet in payload to routing SW (N.L.)
- Routing SW selects out-link, gives packet to DLL. It does not want to be bothered by lost packets
- Responsibility of DLL protocol:
  - make unreliable line look reliable/good
  - use services of physical layer, ≡ error-prone bit stream

  - break bit stream into frames, each with checksum
  - cannot use timing gaps; network gives poor timing guarantees

FRAMING : 4 common methods

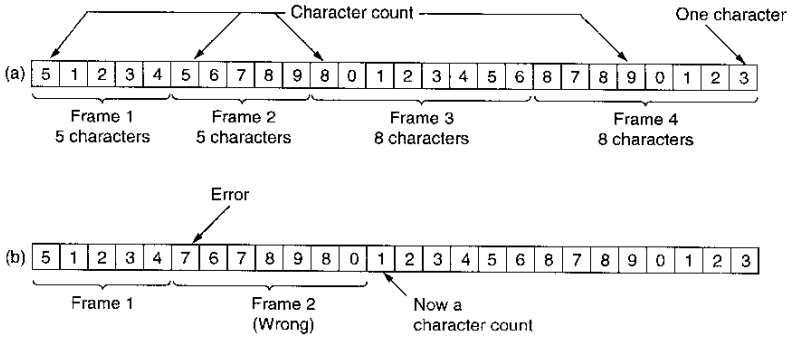


Fig. 3-3. A character stream. (a) Without errors. (b) With one error.

[1] Character count :

field in header gives # chars in frame

Problem : field can get corrupted, no fix

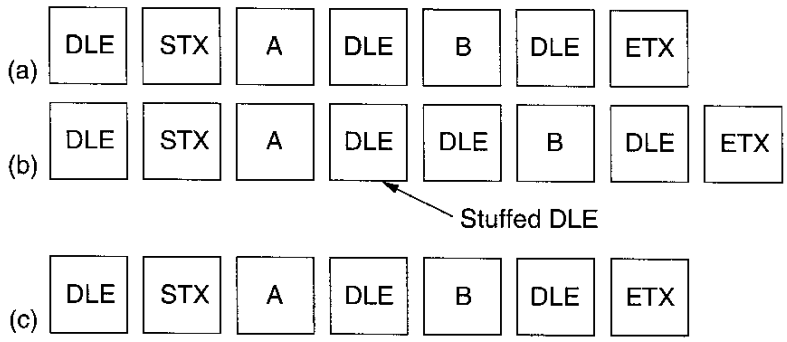
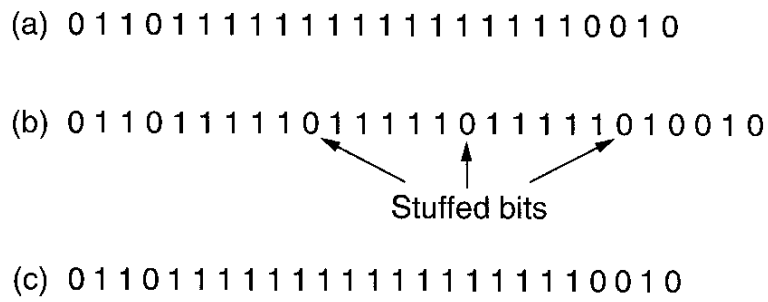


Fig. 3-4. (a) Data sent by the network layer. (b) Data after being character stuffed by the data link layer. (c) Data passed to the network layer on the receiving side.

[II] Starting & Ending Characters, with char. stuffing

- Character stuffing
- Disadvantage: Tied to 8-bit chars, & ASCII code



[III]

Fig. 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

(• allows arbitrary # bits/frame, & char codes w/ arbitrary # bits/char)

- Each frame begins & ends with flag = '0111 1110'
- Sender: ~~user~~ DLL data has 5 consecutive 1's ⇒ stuff a '0' in outgoing bit stream
- Receiver: Delete '0' from any '111 110' pattern
- Transparent to network layer
- User data '0111 1110' sent as '0 111 110 10'

[IV] if encoding on physical medium has redundancy.
eg 1 ≡ high-low pair] high-high & low-low not used
0 ≡ low-high pair] for data. eg 802 LAN standard

- Error control : deliver frames correctly in FIFO
  use timers, seq #s, retransmit
- Digital : rare errors
- Local loops, wireless : high error rates
- Some media (eg wireless) : errors are bursty
  (+) fewer blocks corrupted (−) harder to detect

ERROR-CORRECTING CODES

n-bit codeword = m data bits + r check bits

- Hamming distance : use ⊕
- 2^m legal data messages ; not all 2^n codewords legal
- Hamming distance of a code = minimum Hamming distance between any two legal codewords
- To detect d errors ⇒ need distance (d+1) code
- To correct d errors ⇒ need distance (2d+1) code
- Error-detecting code eg : use parity bit (for single errors)
- Error-correcting code eg : to correct 2 errors,
  {0000000000, 0000011111, 1111100000, 1111111111} legal codewords
- 1-error correcting code : each of 2^m messages has n illegal codewords at distance 1 ⟹
  each of 2^m msgs needs (n+1) bit patterns reserved
  $(n+1)2^m \leq 2^n \Rightarrow (m+r+1) \leq 2^r$ [achievable by Hamming (1950)]

- Check-bits : 1, 2, 4, 8, 16, ----
- Express a position as Σ(powers of 2)
  This bit is "checked" by those bits in this expression

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	11111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	00101011111
d	1100100	11111001100
e	1100101	00111000101

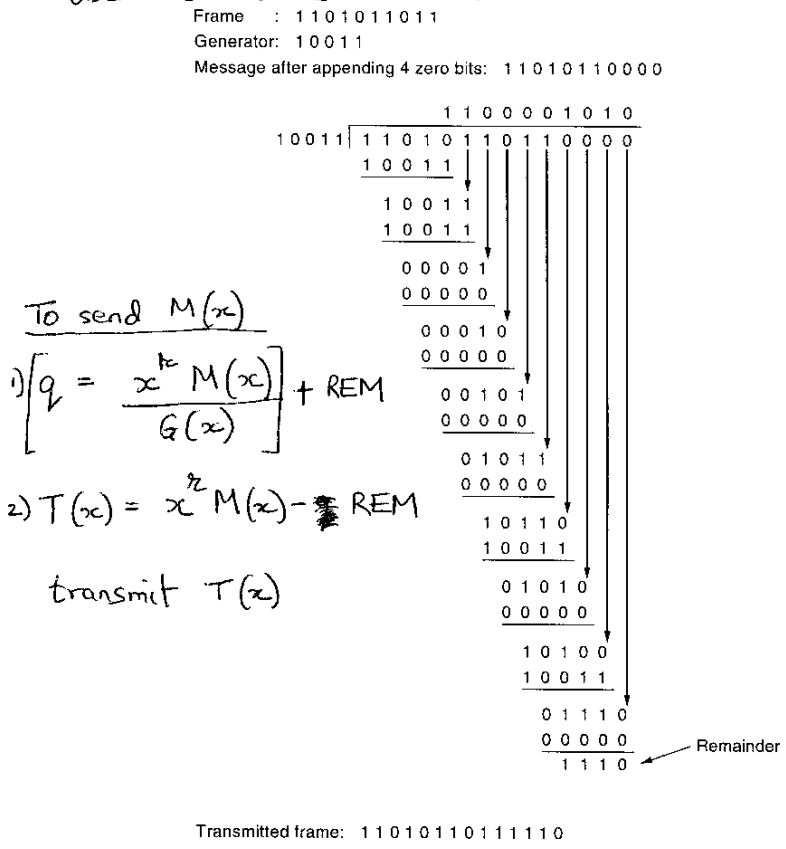
Order of bit transmission

1: 3, 5, 7, 9, 11
2: 3, 6, 7, 10, 11
4: 5, 6, 7 8: 9, 10, 11

Fig. 3-6. Use of a Hamming code to correct burst errors.

- Receiver : Examine each check-bit (k) to see if it has correct parity. If not, add (k) to counter.
  If counter = $\emptyset$, codeword is accepted
  else contains the # of the incorrect bit
- TRICK to correct burst errors of burst $\leq k$,
  — use ($k \times r$) checkbits for blocks of ($k \times m$) data bits
  — use k codewords, transmit columnwise

- View k-bit frame ≡ polynomial w/ k terms x^{K-1} to x^0
- $G(x)$ ≡ generator polynomial, agreed upon. (degree k)
- Use modulo-2 division



To send $M(x)$

1) $\left[q = \frac{x^k M(x)}{G(x)}\right]$ + REM

2) $T(x) = x^k M(x) -$ REM

transmit $T(x)$

Transmitted frame: 11010110111110

Fig. 3-7. Calculation of the polynomial code checksum.

- On receiving $T(x)+E(x)$, error = $\frac{T(x)+E(x)}{G(x)}$
- Arithmatic done in HW.
- Power of polynomial code checksum method : see analysis

From: *Computer Networks*, 3rd ed. by Andrew S. Tanenbaum, © 1996 Prentice Hall

- Std. polynomials : CRC-12, CRC-16 etc.

```
#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;

/* 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
```


Fig. 3-8. Some definitions needed in the protocols to follow.

```
/* 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 fast.
   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 */
  }
}
```

Fig. 3-9. An unrestricted simplex 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 */
  }
}
```

Fig. 3-10. A simplex stop-and-wait protocol.

- guarantee that no combo. of errors causes duplicate frames to network layer. Use ACKs & SEQ_NOs & timers

```
/* 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) {
               from_network_layer(&buffer); /* get the next one to send */
               inc(next_frame_to_send); /* invert next_frame_to_send */
          }
     }
 }
}

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);        /* none of the fields are used */
     }
 }
}
```

- What if no seq-no on ACK?
- Seq_no needed on data & ACK frames

Fig. 3-11. A positive acknowledgement/retransmission protocol. (PAR) or Automatic Repeat ReQuest (ARQ)

From: *Computer Networks*, 3rd ed. by Andrew S. Tanenbaum. © 1996 Prentice Hall

- 1-bit seq No. is necessary & sufficient

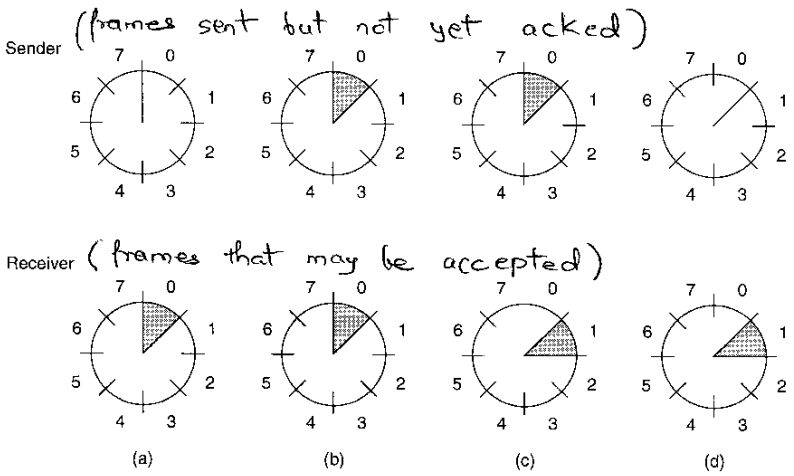


Fig. 3-12. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

- bidirectional ⇒ interleave data & ack frames, use piggybacking (how long to wait for a data frame?)
- Sliding window ⇒ seq nos. $0 \rightarrow MAX = 2^n - 1$
  - sending window, receiving window: diff. limits & size possible
  - window size ⇒ buffers needed to hold unacked. frames
- Stop & wait sliding window ⇒ $n = 1$ (window size = 1)

```
/* Protocol 4 (sliding window) is bidirectional and is more robust than protocol 3. */
#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;                   /* number of frame arriving frame expected */
  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 */
  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 sequence number expected next */
          }
          if (r.ack == next_frame_to_send) { /* handle outbound frame stream. */
              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 */
  }
}
```

Fig. 3-13. A 1-bit sliding window protocol. (uses stop & wait)

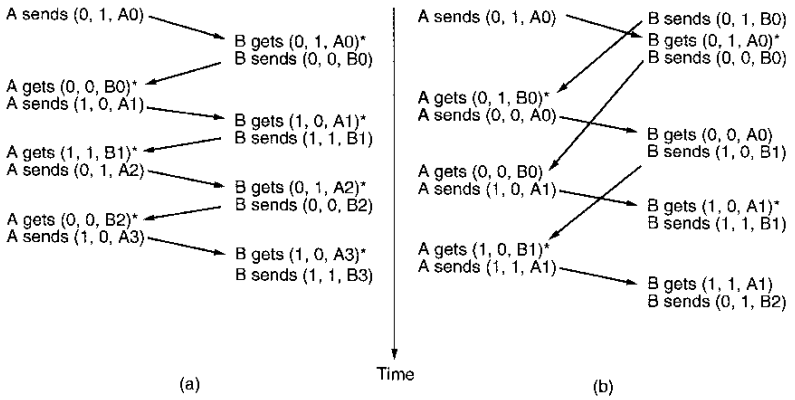
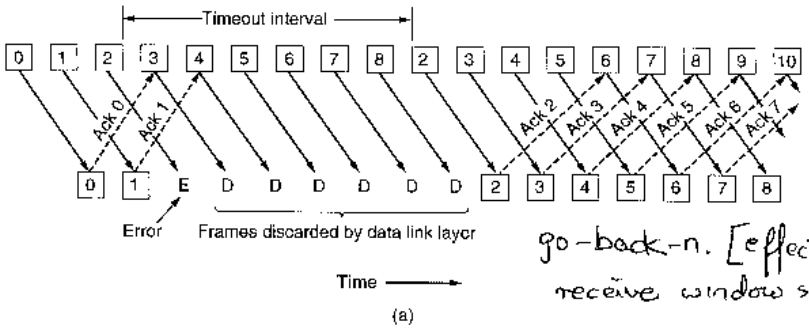


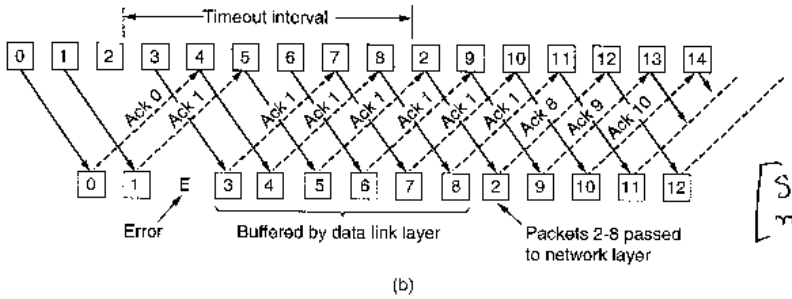
Fig. 3-14. Two scenarios for protocol 4. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

- No combo. of lost frames or premature timeouts causes duplicate packets to the network layer, or lost packets, or deadlock, but --- ---
- if both sides simultaneously send initial packet, half the frames contain duplicates ------ see (b) (similarly if timeouts are premature)

• Need large window size for efficiency : consider propagation times & transmission times



go-back-n. [effectively, receive window size = 1]



[Selective repeat]

Fig. 3-15. (a) Effect of an error when the receiver window size is 1. (b) Effect of an error when the receiver window size is large. (selective repeat)

- use pipelining (send multiple frames w/o having recd. ACK)
- capacity = b bits/s ; frame size = ℓ bits
  round-trip propagation delay = R
  stop & wait line utilization = $\frac{\ell/b}{(\ell/b + R)} = \frac{\ell}{\ell + bR}$
- Frame error → go-back-n [discard subsequent frames & do not send ACK for them]
  → selective repeat

From: *Computer Networks*, 3rd ed. by Andrew S. Tanenbaum, © 1996 Prentice Hall

(bw ↑ ⇒ D.L. buffer space tradeoff)

```
/* 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 */
}

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 */
```

```
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 */
          }

         /* 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();
}
}
```

From: *Computer Networks*, 3rd ed. by Andrew S. Tanenbaum, © 1996 Prentice Hall

Fig. 3-16. A sliding window protocol using go back n.

Each data frame is acked; ack (k) acks k-1, k-2 etc.

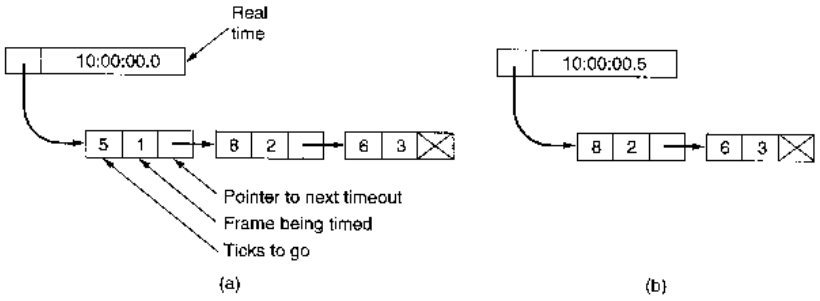


Fig. 3-17. Simulation of multiple timers in software.

- Go-back-n :

  max. # of outstanding frames is $2^n - 1$ ie, MAX_SEQ,
  not MAX_SEQ + 1.

  Why ? counterexample w/ MAX_SEQ = 7

(a) send frames $\phi \rightarrow 7$
(b) piggybacked ack(7) arrives
(c) send frames $\phi - 7$ (new)
(d) piggybacked ack(7) arrives
Were some/all step(c) frames delivered or lost ?
Sender cannot tell !

```
/* 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
   goes off, 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 */ (for frame_expected)
seq_nr oldest_frame = MAX_SEQ + 1;      /* initial value is only for the simulator */ (this will timeout first)

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 */
}

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 */ → full or empty ?
  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;
```

```
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 */
                   }
              }
         }
         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();
 }
}
```

Fig. 3-18. A sliding window protocol using selective repeat. [Protocol 6]

- In protocol 5, ack is only piggybacked on outgoing frame; can be held up (forever) [PROBLEM]
  [Also, not every frame acked explicitly]

- Protocol 6 fixes [PROBLEM] by using start_ack_timer & ack_timeout event
  How to choose appropriate value for this aux. timer?

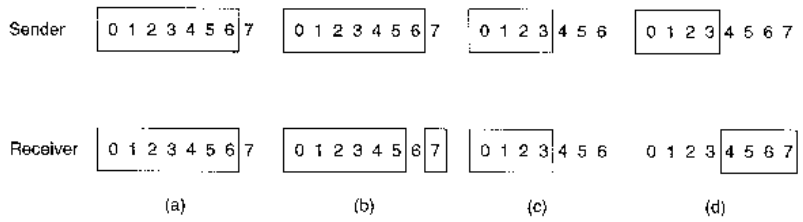
- Protocol 6 deals w/errors more efficiently using NAKs

→ when std. dev. of acknowledgement interval is large, NAKs are very useful as they save unnecessary retransmissions & unnecessary waits
(interval > timeout) (interval < timeout)
ie, save BW

→ no_nack is true if no NACK sent for frame_expected
If NACK lost, sender will timeout & retransmit anyway

→ (timeout & NAKs) which frame caused timeout?
Protocol 5 : always the oldest, ie, ack_expected
Protocol 6 : nontrivial. Eg. 0 1 2 3 4 sent.
0 (TO), 5, 1 (TO), 2 (TO), 6 ⇒ 3 4 0 5 1 2 6
If inbound traffic lost, this is timeout order
⇒ Need complex timer administration for oldest frame

• 3-bit seq. no



Problems due to nonseq. receives [in Selective Repeat]

Fig. 3-19. (a) Initial situation with a window of size seven. (b) After seven frames have been sent and received but not acknowledged. (c) Initial situation with a window size of four. (d) After four frames have been sent and received but not acknowledged.

(b) Acks sent by receiver are lost

(b1) Sender timesout & resends Fr. ϕ

(b2) Receiver accepts this & sends piggybacked Ack (6), as ϕ thru' 6 have been received

(b3) Sender receives Ack(6), advances its window & sends frames 7, 0, 1, 2, 3, 4, 5

(b4) Receiver receives frame 7, gives this & earlier (b1) frame ϕ to network layer $\Rightarrow$ error !

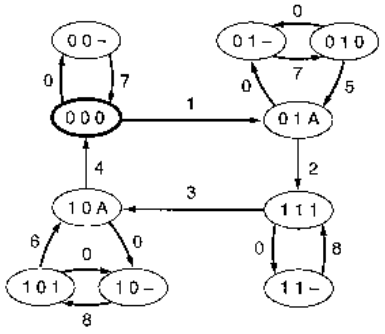
Problem: After receiver advanced window, new range of valid seq. nos. overlapped the old range !

$\therefore$ max window size $\leq (MAX\_SEQ + 1)/2$

• # buffers = # timers = max. window size

PROTOCOL SPECIFICATION & VERIFICATION

- FSM = <States, Frames, Init_States, Transitions>



(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	–

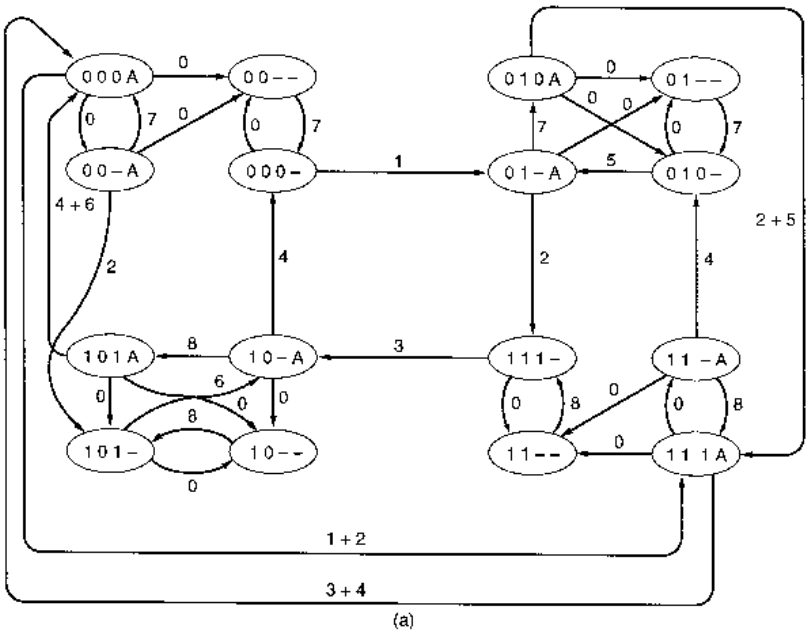
(b)

state ≡ < frame sender is trying to send ; frame expected by receiver ; state of channel >

Fig. 3-20. (a) State diagram for protocol 3. (b) Transitions.

< 0/1 ; 0/1 ; 0/1/A/empty >

- total 16 states, only 9 shown reachable from init
- Reachability analysis used for detecting :
  - incompleteness — deadlock — extraneous transition
- Normal operation: transitions → 1 → 2 → 3 → 4
- Use FSM to verify properties like
  - receiver does not deliver 2 odd packets w/o intervening even packet
  - no path where sender changes state twice while receiver state stays same
  - no deadlock, i.e., no subset B of states
    → no transition out of B, &
    → no transitions in B cause forward progress



(a)

(000–), (01–A), (010A), (111A), (11–A), (010–), (01–A), (111–)

(b)

Fig. 3-21. (a) State graph for protocol 3 and a full-duplex channel. (b) Sequence of states causing the protocol to fail.

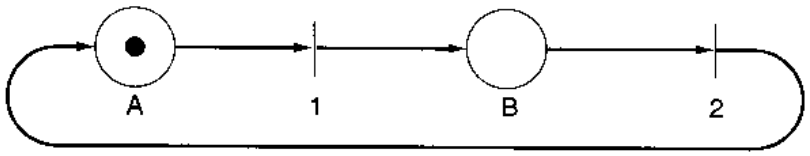


Fig. 3-22. A Petri net with two places and two transitions.

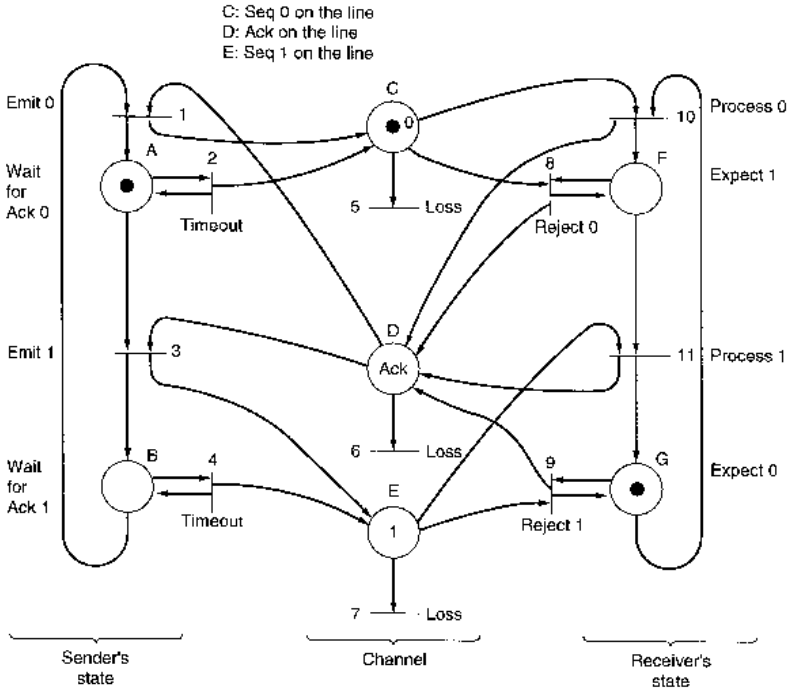


Fig. 3-23. A Petri net model for protocol 3.

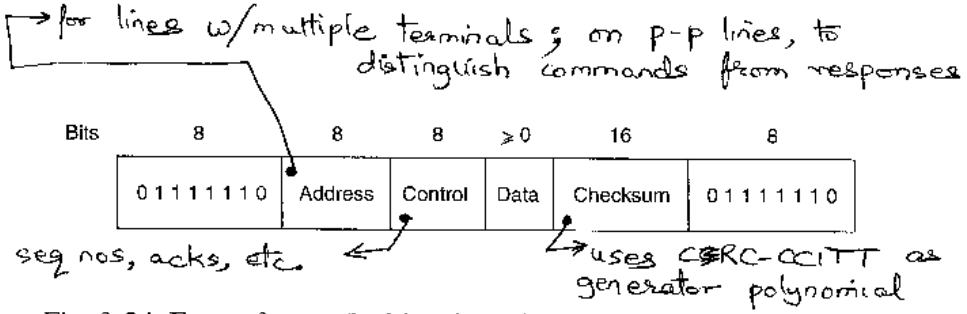


Fig. 3-24. Frame format for bit-oriented protocols. eg HDLC, SDLC

- use bit stuffing (in X.25 for example)
- on idle p-p lines, flag segs transmitted continuously

- sliding-window protocol w/ 3-bit seq. # ⇒ ≤ 7 outstanding unacked frames
- P/F = Poll / Final bit : to poll terminals to invite to send data (P); force partner to send Supervisory frame onto which info about window is piggybacked; etc.
- Supervisory frames
  - ACK → next frame expected in Type field
  - NAK, like protocol 5 (go-back-N)
  - Receive Not Ready : like ACK, + stop sending
  - Selective Reject; retransmit only the frame specified (selective repeat; like protocol 6)

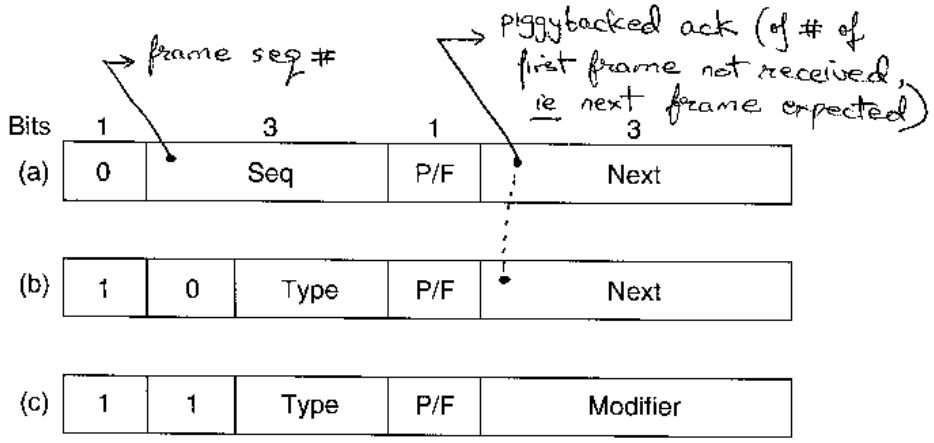
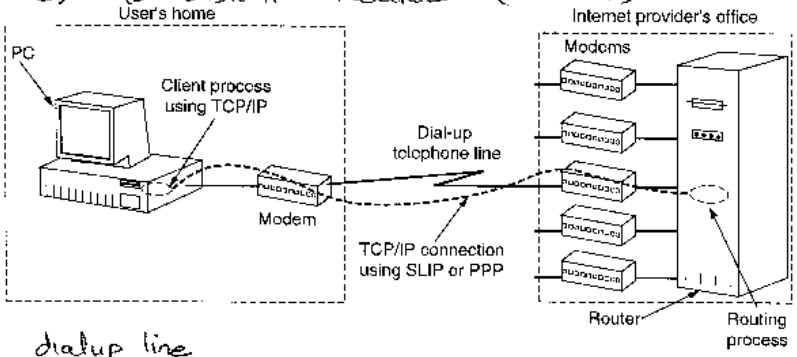


Fig. 3-25. Control field of (a) an information frame, (b) a supervisory frame, (c) an unnumbered frame.

(c) Unnumbered frames
- control purposes;
- data for unreliable connectionless service

Common commands
- DISC (disconnect)
- SNRM (Set Normal Response Mode) / SABM (Set Asynchronous Balance Mode) } reset line & seq. nos.
- FRMR (Frame Reject): correct checksum, but wrong semantics
- UA (Unnumbered Ack): for (the only) outstanding unnumbered frame
- Others: for initializing, polling, status reporting
- UI (Unnumbered Info): arbitrary info.

• DL protocols for p-p lines in Internet
— in organization, outside lines go thru' router w/p-p leased lines to distant routers (subnet)



— dialup line / leased line between PC & router @ ISP
• Thus, router-router leased lines & dial-up host-router connections

Fig. 3-26. A home personal computer acting as an Internet host.

→ SLIP (Serial Line IP); Rick Adams/1984; RFC 1055 (+ RFC 1144)

Sun wkstn ⟷ dial-up line using modem ⟷ Internet

• raw IP packets over line, flags for framing; char. stuffing
TCP, IP hdr compressions (RFC 1144)

• Drawbacks
— no error detection/correction
— supports only IP (eg no Novell LAN support)
— each end must know other end's IP address in advance
— no authentication support (not an issue w/leased lines; certainly issue for dialup lines)
— not approved Internet std (⇒ many versions)

→ PPP (Point-to-Point Protocol); via IETF; RFC 1661 (+1662 +1663)

From: *Computer Networks*, 3rd ed. by Andrew S. Tanenbaum. © 1996 Prentice Hall

overcame drawbacks of SLIP, including the lack of support for dynamic IP addr assignment

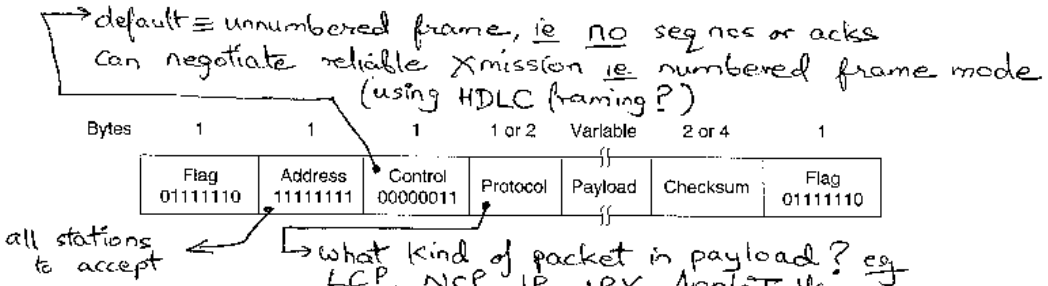


Fig. 3-27. The PPP full frame format for unnumbered mode operation.

- PPP provides
  - framing to mark frame boundaries; error detection
  - LCP (Link Control Protocol) to bring up lines, test them, negotiate options, bring down lines gracefully
  - way to negotiate network-layer options in a way independent of network layer protocol to be used. VIA different NCP (Network Control Protocol) for each network layer supported ed.
- PPP frame format similar to HDLC
  - PPP is char-oriented, not bit-oriented
  - char stuffing, not bit-stuffing
  - can be sent on dial-up phone lines/SONET/bit-oriented HDLC lines for rtr-rtr. conn.
  - using LCP, can negotiate to omit Addr/Ctrl fields in default
  - Checksum, Payload max. lengths are negotiable
- Multiprotocol framing mechanism

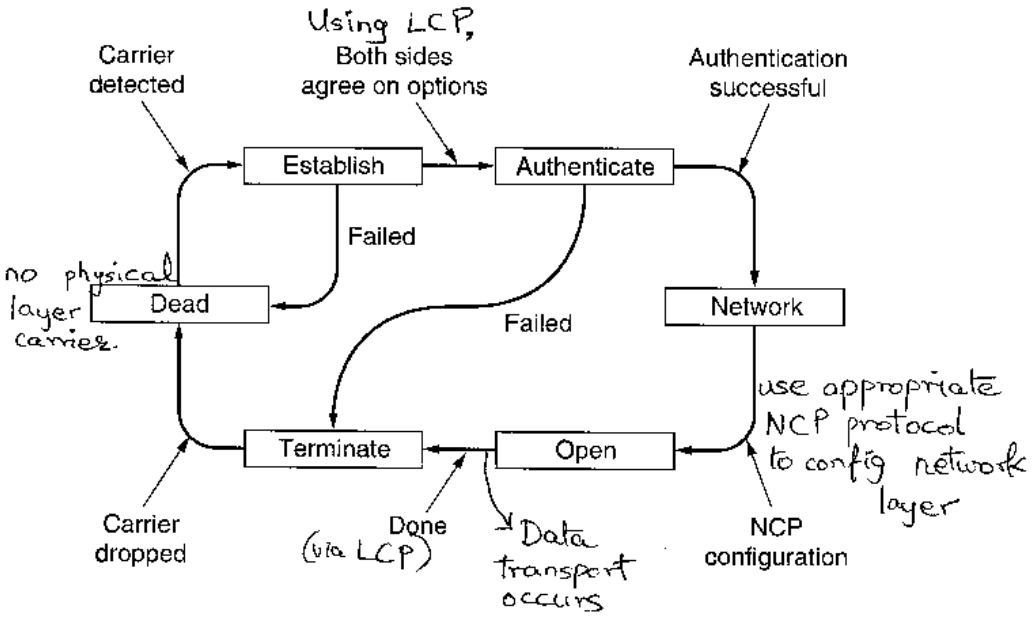


Fig. 3-28. A simplified phase diagram for bringing a line up and down. for modem & rtr-rtr connections

- LCP used to negotiate data link protocol options; not concerned with the options themselves, but with the mechanism for negotiation.
  - initiator makes proposal, responder accepts/rejects partially or wholly
  - to test line quality
  - take down lines

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)

I ≡ Initiator ; R ≡ Responder

Fig. 3-29. The LCP packet types — RFC 1661

- Negotiable options include
  - max payload size
  - enabling authentication & choosing a protocol
  - enabling line quality monitoring
  - selecting hdr. compression options

ATM TC (Transmission Convergence) sublayer

- ATM has no physical layer characteristics
- Cells carried by SONET, FDDI, others

- Cell Xmission
  - ATM layer ⟶ sequence of cells ⟶
  - Add a checksum HEC (Header Error Ctrl) 1 byte
    (for reliable Xmission, Xoring consecutive cells scheme also possible)
  - convert to bit stream, for asynch or synch transmission medium
    (for synch. medium) add idle or OAM cells to match speed of Xmission medium eg SONET
  - generate framing info for underlying Xmission system eg SONET, T1, T3, FDDI etc.

- Cell Reception
  - incoming bit stream ⟶ locate cell boundaries (tough!)
    ⟶ verify headers & discard cells w/ invalid headers
    ⟶ process OAM cells ⟶ fwd data cells to ATM layer
  - To locate cell boundaries
    - physical layer may help, eg SONET: align with synch. payload envelope, use SPE pointer
    - otherwise, use the HEC check for 40-bit windows, implemented by shift registers

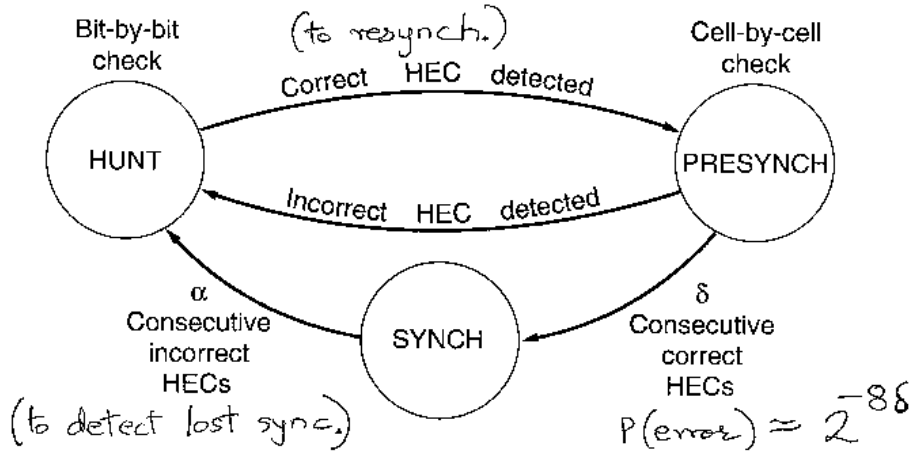
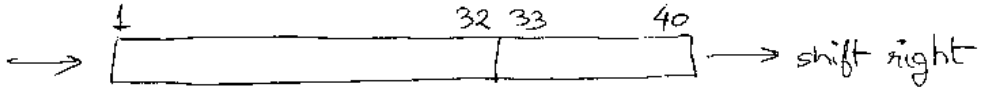


Fig. 3-30. The cell delineation heuristic.

- 8-bit HEC over 32-bit field ⇒ 1/256 valid HEC even with random bits

∴ look for δ consecutive cells w/correct HECs.

- Can use scrambling/descrambling of payload bits to protect against malicious user causing wrong synchr.
- TC sublayer (DL func.) uses ATM layer cell header ⇒ dependency of layers ⇒ poor protocol engineering!



$$\text{REM}\left[\frac{(\text{bits } 1\text{-}32 \text{ poly.})}{x^8+x^2+x+1}\right] + 01010101 = \text{bits } 33\text{-}40?$$