CSMC 417

Computer Networks

Prof. Ashok K Agrawala

Set 4

The Data Link Layer

Data Link Layer Design Issues

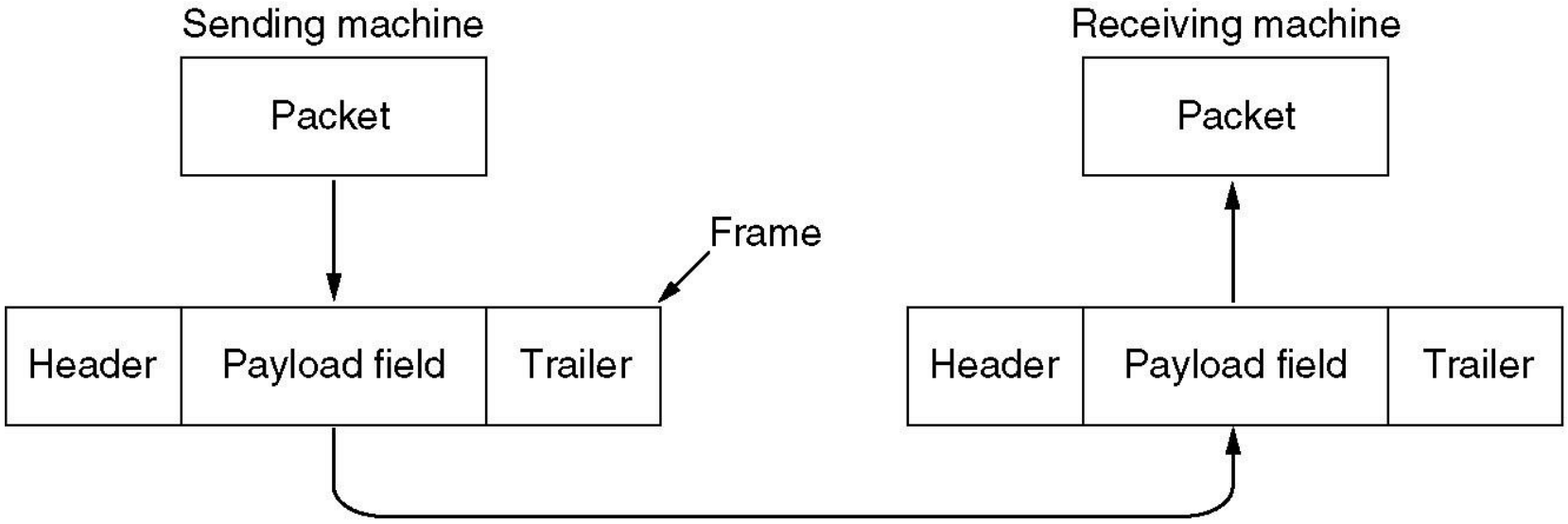
- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control

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)

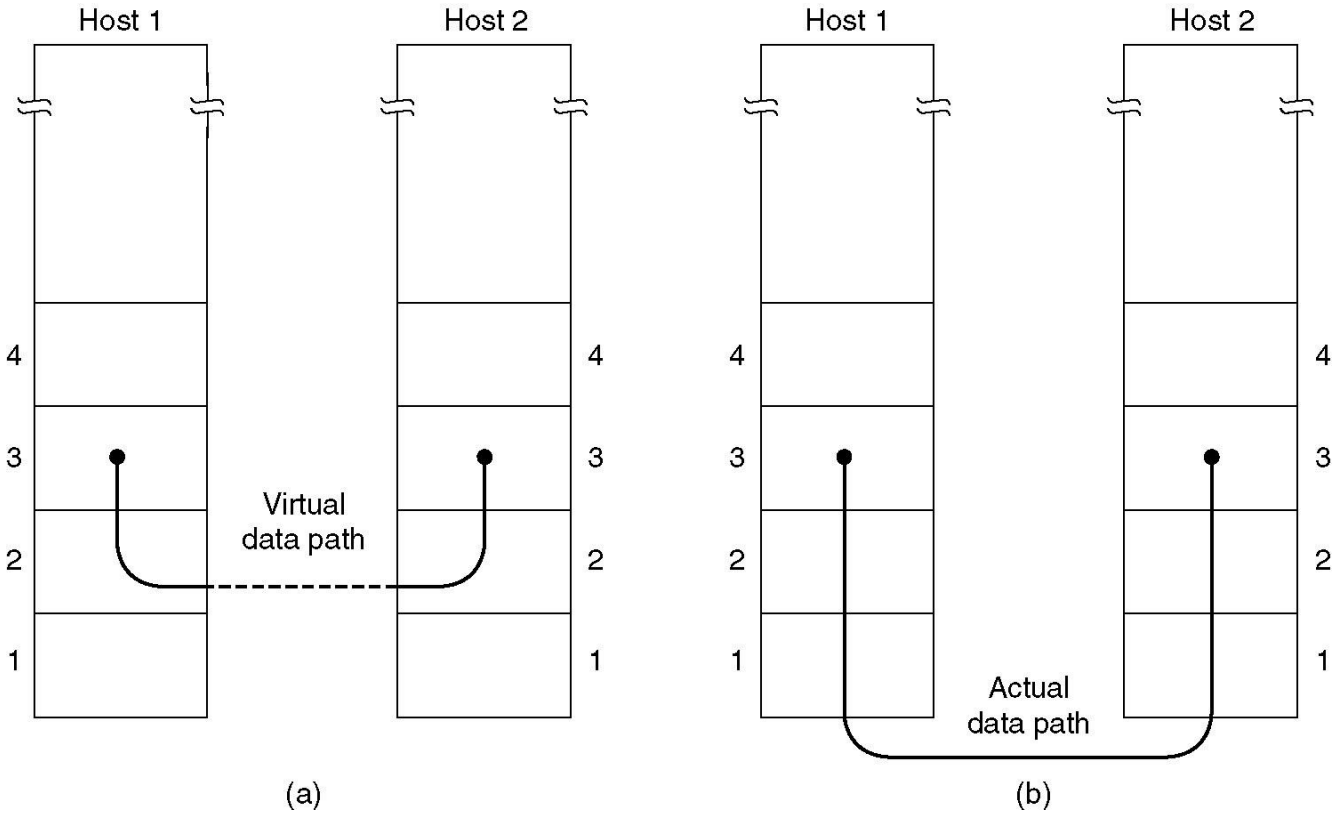
Relationship between packets and frames.



Possible Services Offered

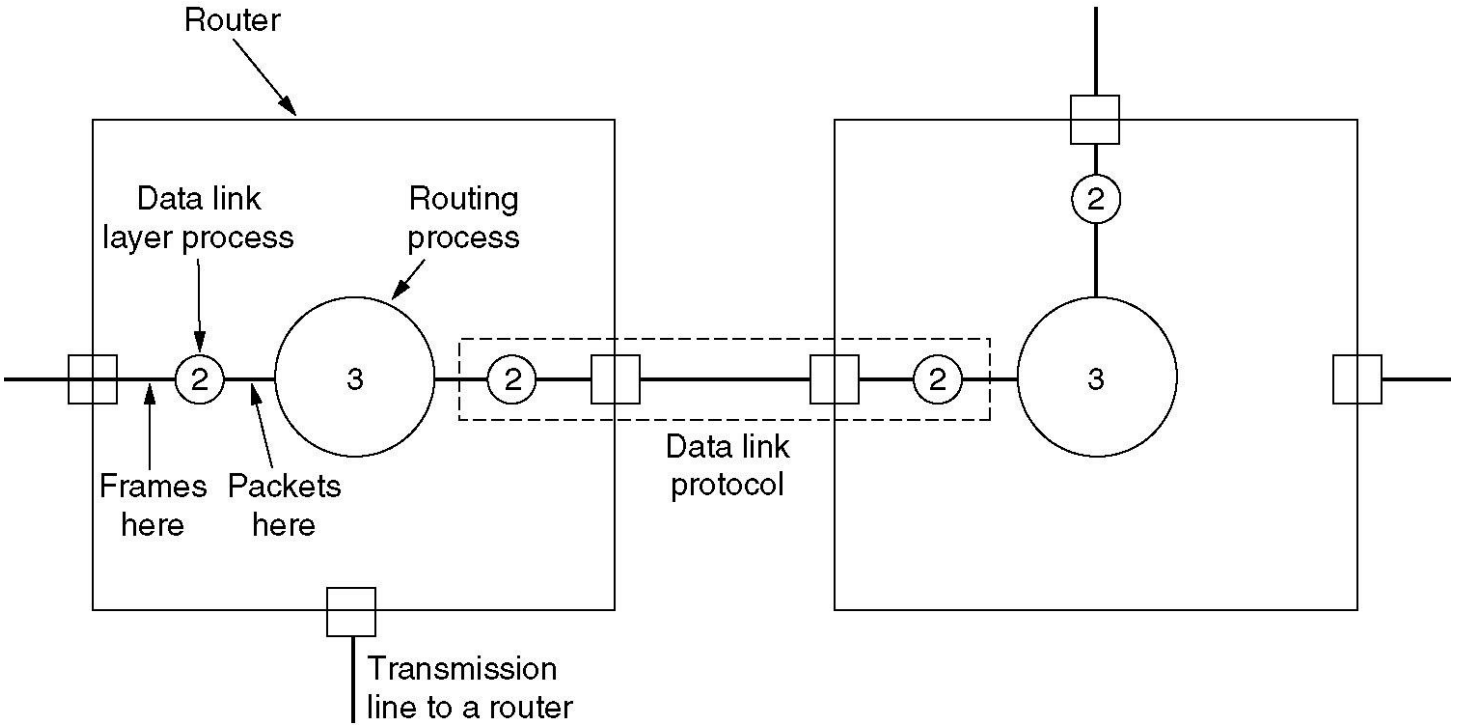
1. Unacknowledged connectionless service.
2. Acknowledged connectionless service.
3. Acknowledged connection-oriented service.

Services Provided to Network Layer



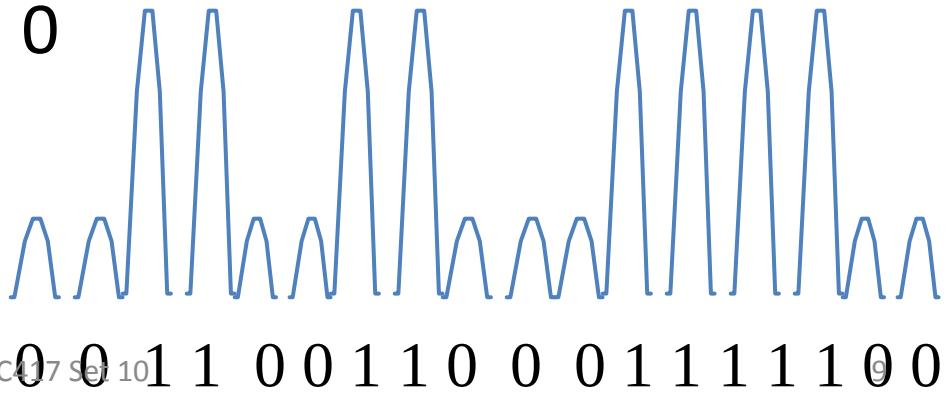
(a) Virtual communication.
(b) Actual communication.

Services Provided to Network Layer (2)



Encoding

- Signals propagate over physical links
  - Source node encodes the bits into a signal
  - Receiving node decodes the signal back into bits
- Simplify some electrical engineering details
  - Assume two discrete signals, high and low
  - E.g., could correspond to two different voltages
- Simple approach
  - High for a 1, low for a 0



Problem With Simple Approach

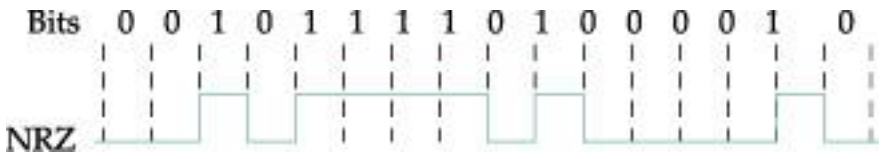
- Long strings of 0s or 1s introduce problems
  - No transitions from low-to-high, or high-to-low
- Receiver keeps average of signal it has received
  - Uses the average to distinguish between high and low
  - Long flat strings make receiver sensitive to small change
- Transitions also necessary for clock recovery
  - Receiver uses transitions to drive its own clock
  - Long flat strings do not produce any transitions
  - Can lead to clock drift at the receiver
- Alternatives
  - Non-return to zero inverted, and Manchester encoding

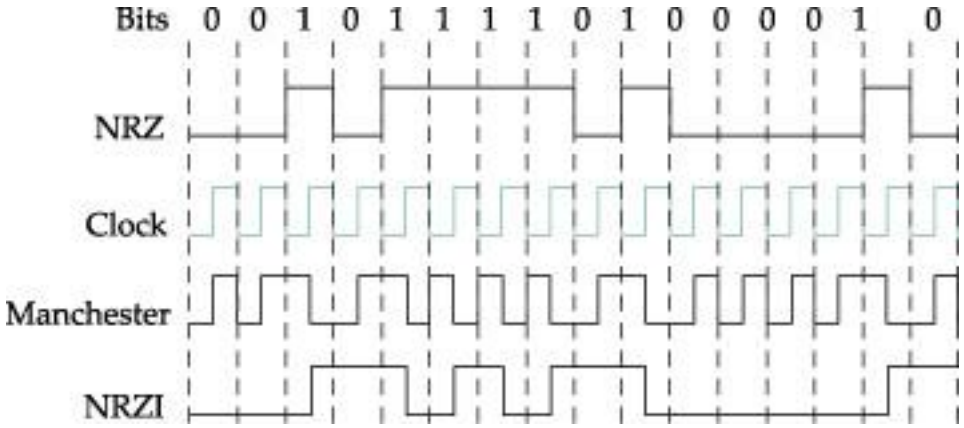
Encoding

- NRZ
- NRZI
- Manchester
- 4B/5B

- Bit Rate and Baud Rate

Encoding


Bits 0 0 1 0 1 1 1 1 0 1 0 0 0 0 1 0
NRZ


Bits 0 0 1 0 1 1 1 1 0 1 0 0 0 0 1 0
NRZ
Clock
Manchester
NRZI

4-bit Symbol	5-bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

4B/5B Encoding

Each 1 has no more than one leading 0 and no more then two trailing 0s.

Sent using NRZI

Framing Methods

1. Byte count.
2. Flag bytes with byte stuffing.
3. Flag bits with bit stuffing.
4. Physical layer coding violations.

Framing

- Break sequence of bits into a frame
  - Typically implemented by the network adaptor
- Sentinel-based
  - Delineate frame with special pattern (e.g., 01111110)

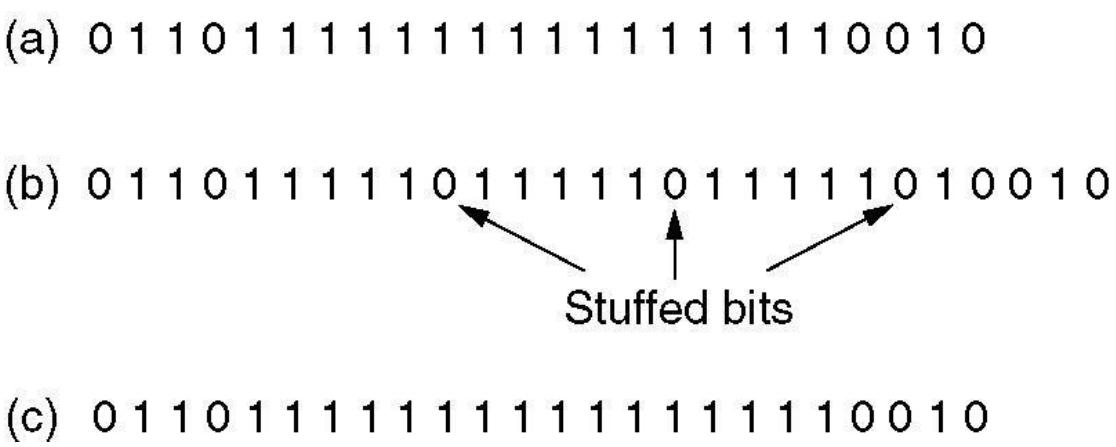
  - Problem: what if special patterns occurs within frame?
  - Solution: escaping the special characters
    - E.g., sender always inserts a 0 after five 1s
    - … and receiver always removes a 0 appearing after five 1s
    - Bit Stuffing
  - Similar to escaping special characters in C programs

Bit Oriented Protocols

- Frame – a collection of bits
  - No Byte boundary
- SDLC – Synchronous Data Link Control
  - IBM
- HDLC – High-Level Data Link Control
  - ISO Standard

HDLC Frame Format

Framing – Bit Oriented



Bit stuffing

(a) The original data.

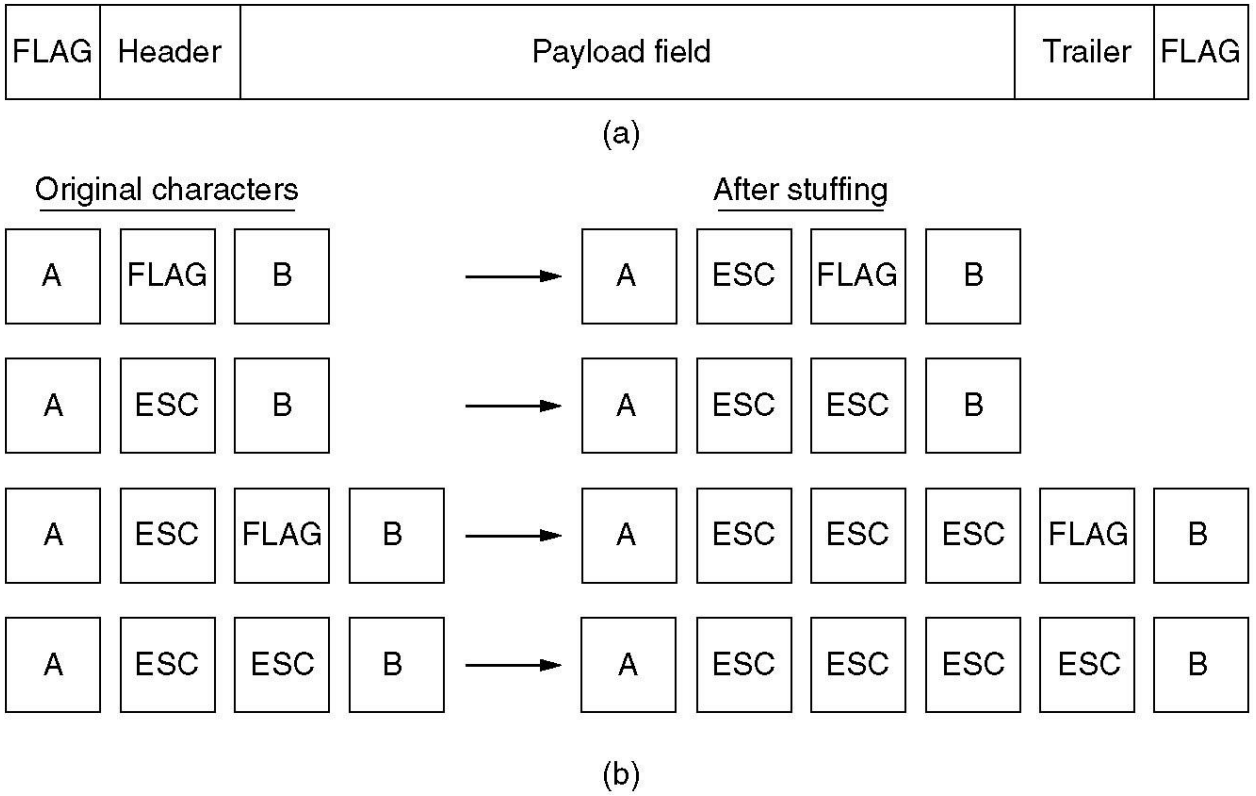
(b) The data as they appear on the line.

(c) The data as they are stored in receiver's memory after destuffing.

Byte-Oriented Protocols

- Frame – a collection of bytes.
- Examples
  - BISYNC – Binary Synchronous Communication – IBM
  - DDCMP – Digital Data Communication Message Protocol
  - PPP – Point-to-Point
- Sentinel Based – Use special character as marker
  - BISYNC
    - SYN and SOH
    - STX and ETX
    - DLE as escape character. - Character Stuffing

Framing

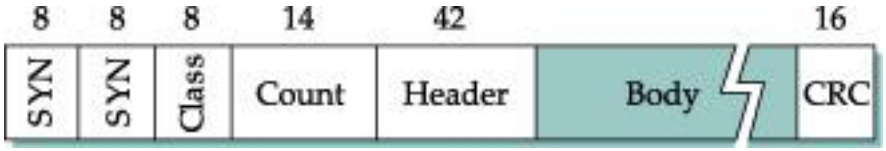


(a) A frame delimited by flag bytes.

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

Frame Structure

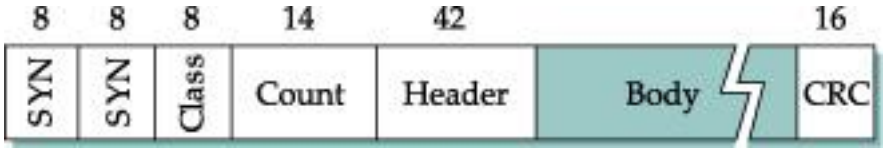
PPP Frame Format



BISYNC Frame Format

Framing (Continued)

- Counter-based
  - Include the payload length in the header
  - … instead of putting a sentinel at the end
  - Problem: what if the count field gets corrupted?
    - Causes receiver to think the frame ends at a different place
  - Solution: catch later when doing error detection
    - And wait for the next sentinel for the start of a new frame



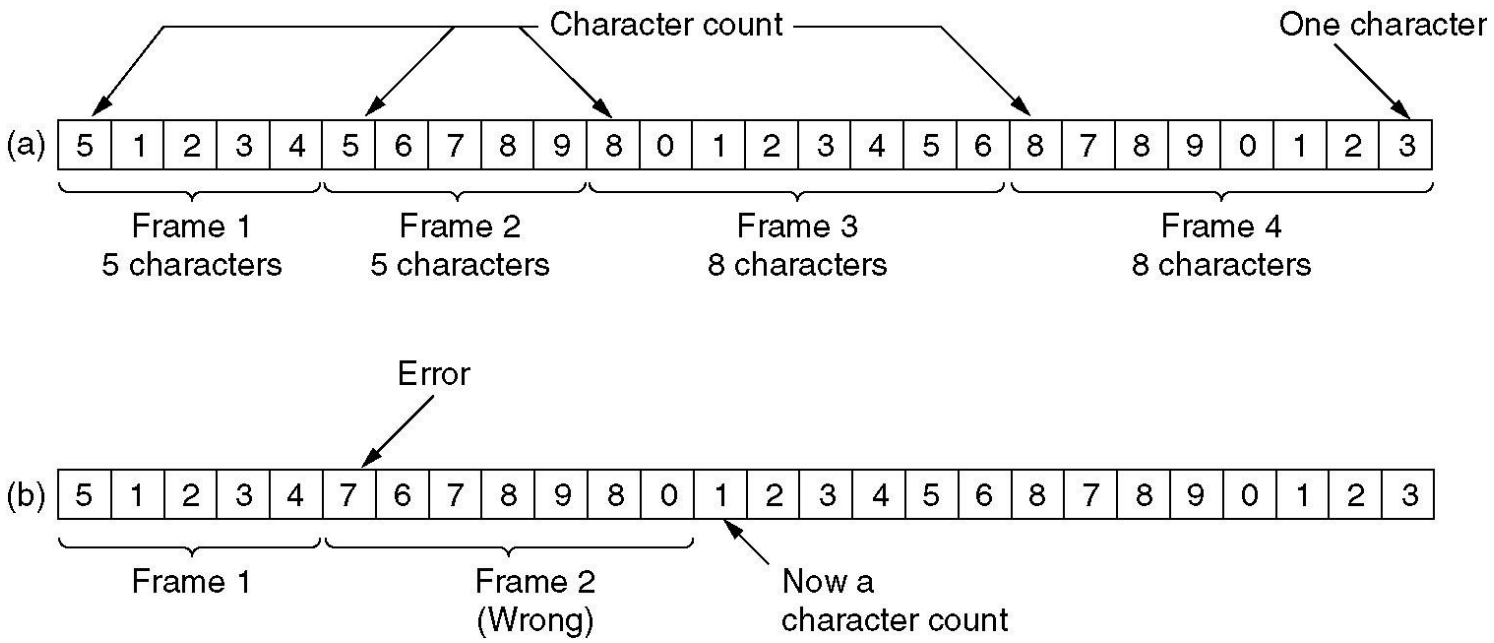
DDCMP Frame Format

Framing

A character stream.

(a) Without errors.

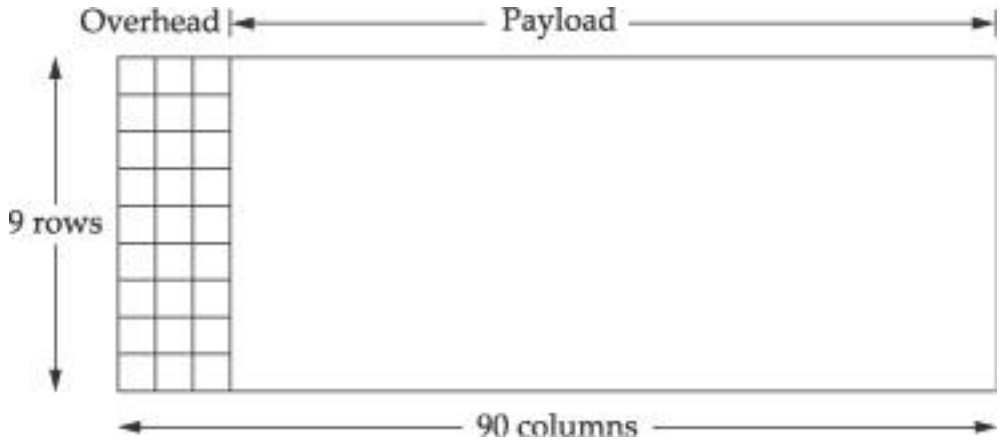
(b) With one error.



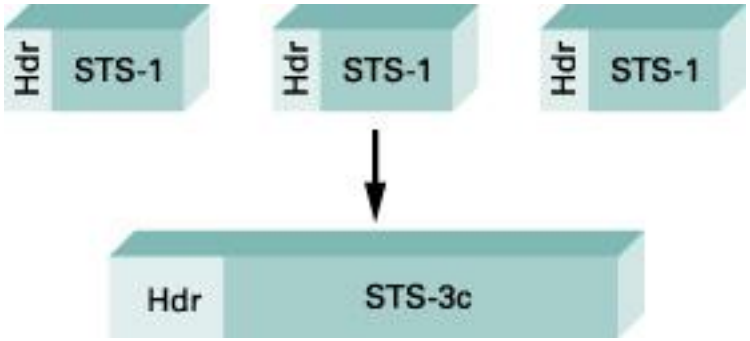
Clock-Based Framing (SONET)

- Clock-based
  - Make each frame a fixed size
  - No ambiguity about start and end of frame
  - But, may be wasteful
- Synchronous Optical Network (SONET)
  - Slowest speed link STS-1 – 51.84 Mbps (810*8*8K)
  - Frame – 9 rows of 90 bytes
    - First 3 bytes of each row are overhead
    - First two bytes of a frame contain a special bit pattern – to mark the start of the frame – check for it every 810 bytes

Sonet Frame



Three STS-1 frames to one STS-3 frame



Error Detection

- Errors are unavoidable
  - Electrical interference, thermal noise, etc.
- Error detection
  - Transmit extra (redundant) information
  - Use redundant information to detect errors
  - Extreme case: send two copies of the data
  - Trade-off: accuracy vs. overhead
- Techniques for detecting errors
  - Parity checking
  - Checksum
  - Cyclic Redundancy Check (CRC)

Error Detection Techniques

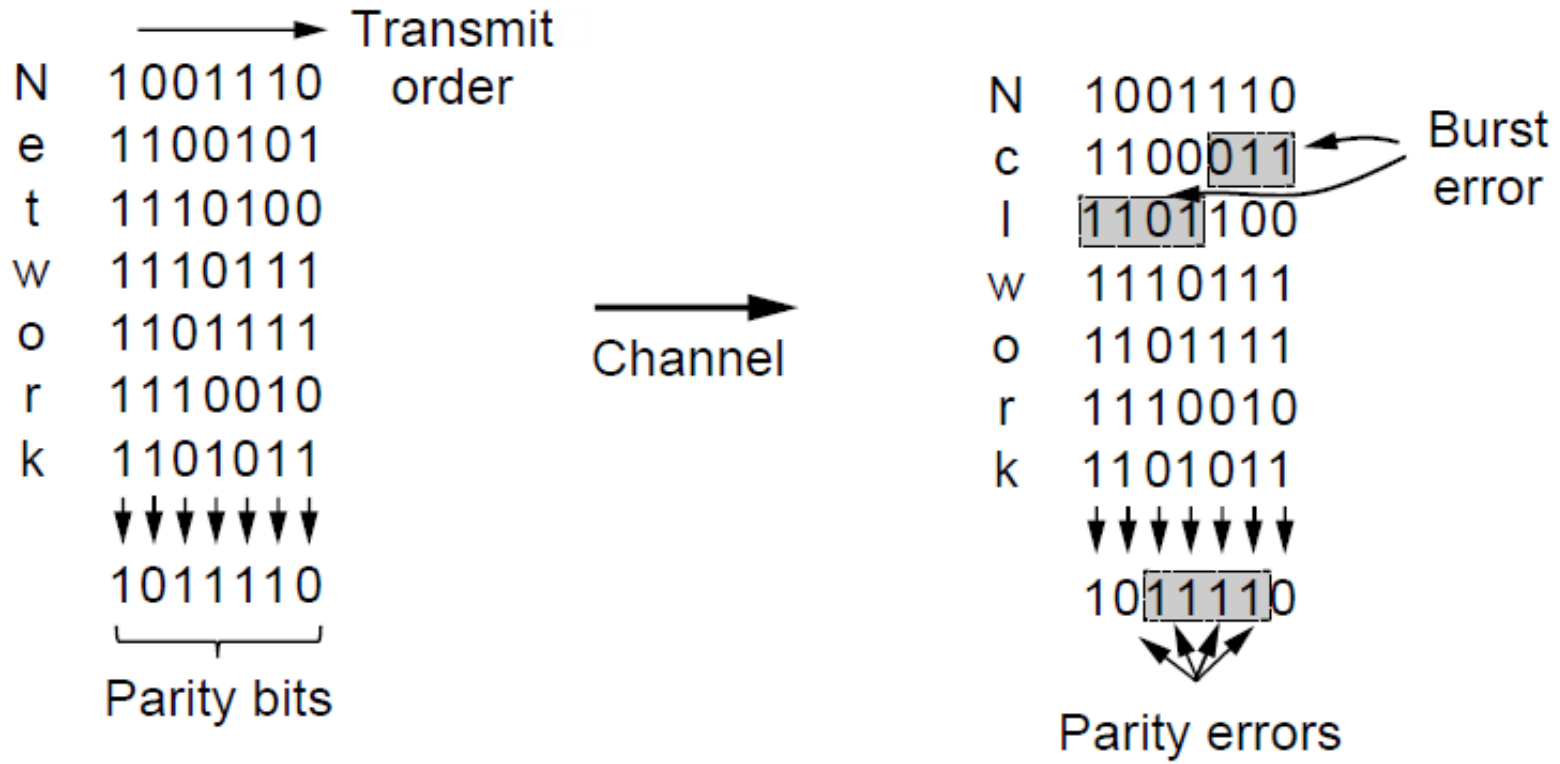
- Parity check
  - Add an extra bit to a 7-bit code
  - Odd parity: ensure an odd number of 1s
    - E.g., 0101011 becomes 01010111
  - Even parity: ensure an even number of 1s
    - E.g., 0101011 becomes 01010110
- Two Dimensional Parity

Error-Detecting Codes (1)

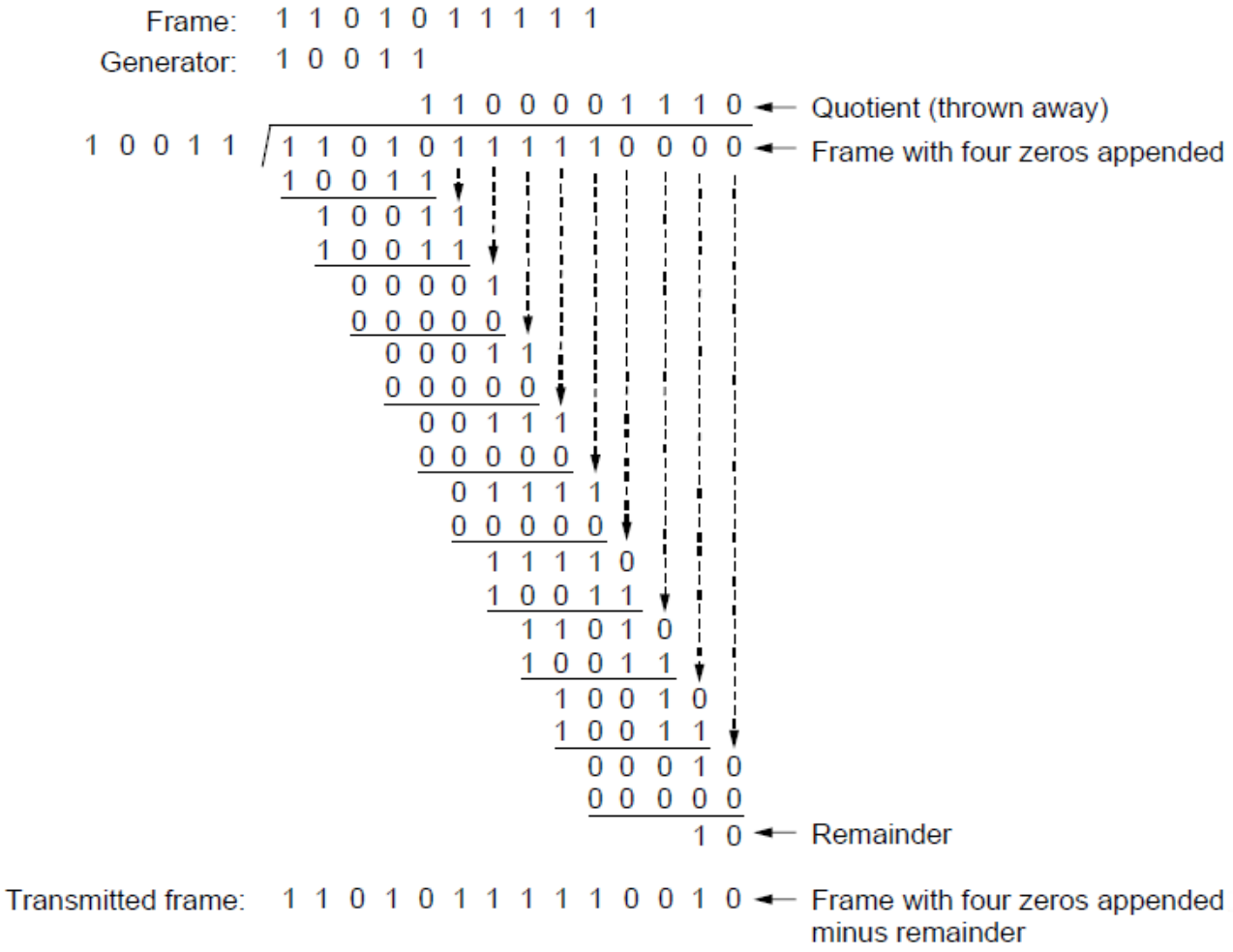
Linear, systematic block codes

1. Parity.
2. Checksums.
3. Cyclic Redundancy Checks (CRCs).

Error-Detecting Codes (2)

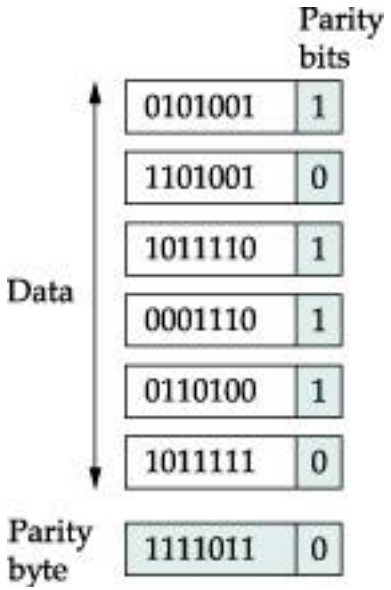


Error-Detecting Codes (3)



Example calculation of the CRC

Two Dimensional Parity



Checksum

- Checksum
  - Treat data as a sequence of 16-bit words
  - Compute a sum of all the 16-bit words, with no carries
  - Transmit the sum along with the packet

Internet Checksum Algorithm

- Consider data as a sequence of 16-bit integers
- Add them together using 16-bit one's complement arithmetic
- Take 1's complement of the sum
- That is the checksum

Cyclic Redundancy Check

- Have to maximize the probability of detecting the errors using a small number of additional bits.
- Based on powerful mathematical formulations – theory of finite fields
- Consider (n+1) bits as n degree polynomial
- Message M(x) represented as polynomial
- Divisor C(x) of degree k
- Send P(x) as (n+1) bits +k bits such that P(x) is exactly divisible by C(x)

$$C(x) = x^3 + x^2 + 1$$

$$M(x) = x^7 + x^4 + x^3 + x^1$$

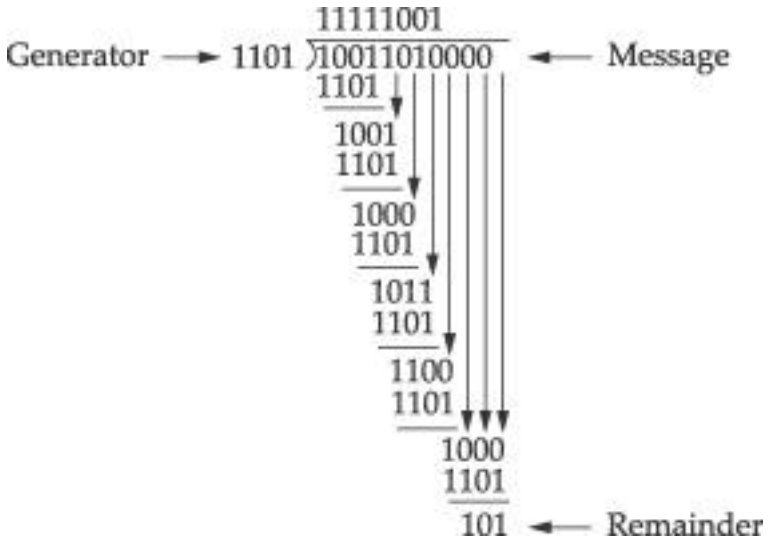
CRC Basis

- Use modulo 2 arithmetic
- Any Polynomial B(x) can be divided by a divisor polynomial C(x) if B(x) is of higher degree than C(x)
- Any polynomial B(x) can be divided once by a divisor polynomial C(x) if they are of the same degree
- The remainder obtained when B(x) is divided by C(x) is obtained by subtracting C(x) from B(x)
- To subtract C(x) from B(x) we simply perform the exclusive-OR operation on each pair of matching coefficients.

CRC Basis

1. Multiply M(x) by x^k , i.e. add k zeros at the end of the message. Call this T(x)
2. Divide T(x) by C(x)
3. Subtract the remainder

from T(x)



- Message sent –

1001101010 101

Cyclic Redundancy Check

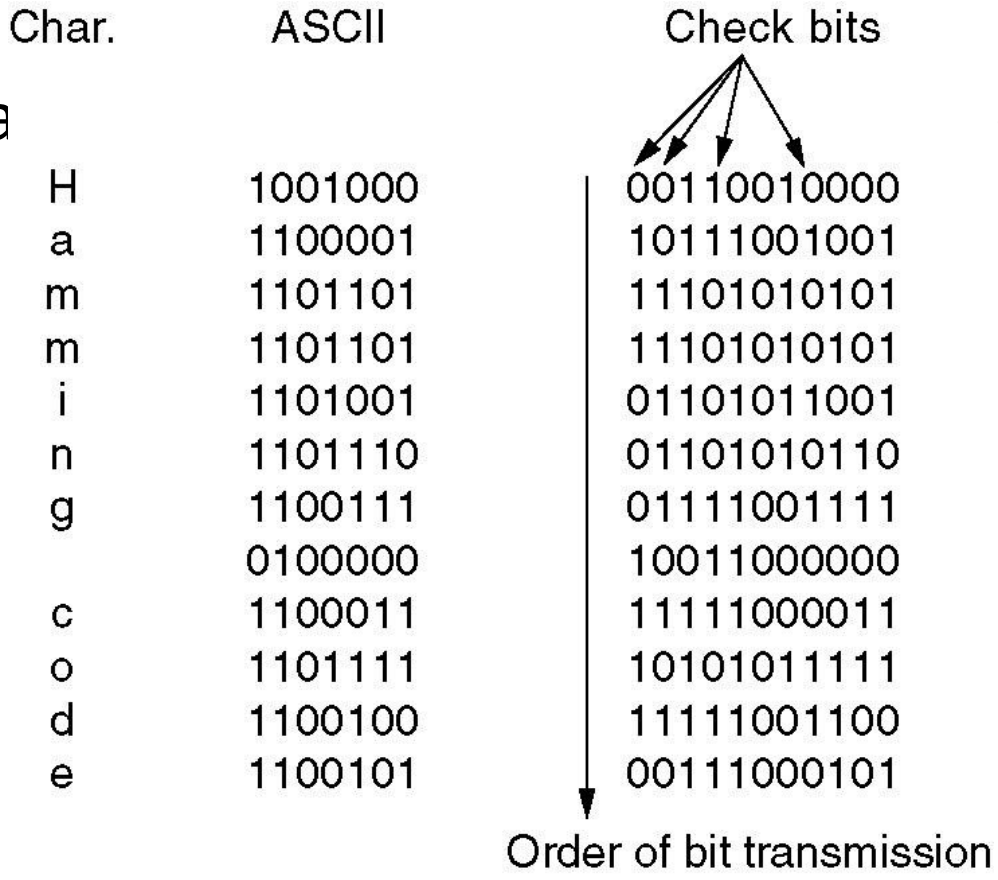
- All single bit errors – if x^k and x^0 terms are nonzero
- All double-bit errors – as long as C(x) has a factor with at least three terms
- Any odd number of errors as long as C(x) has (x+1) as a factor
- Any burst error of length k bits

Common CRC Polynomials

CRC	C(x)
CRC-8	$x^8 + x^2 + X^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$

Error-Correcting Codes

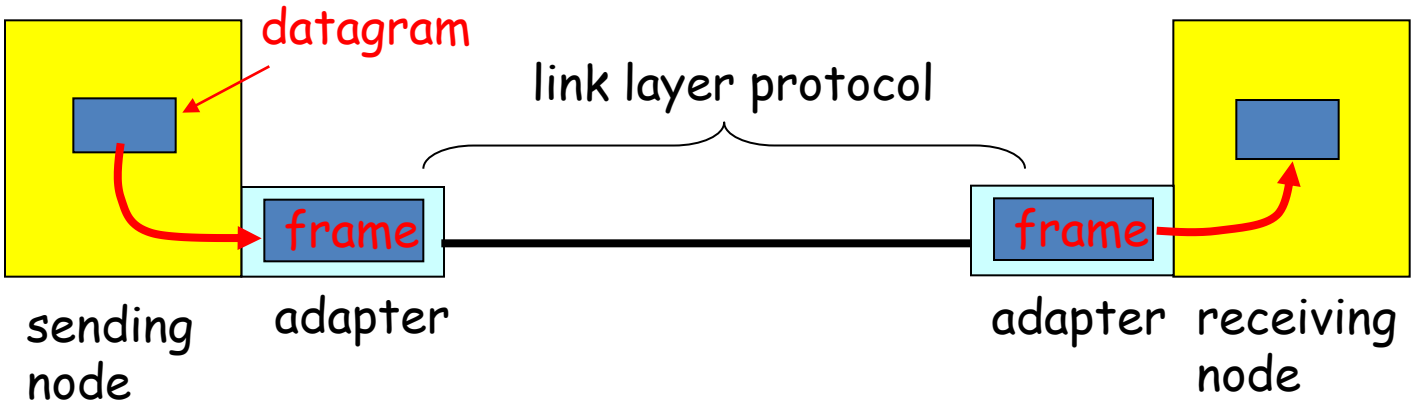
Use of a errors.



Link-Layer Services

- Encoding
  - Representing the 0s and 1s
- Framing
  - Encapsulating packet into frame, adding header, trailer
  - Using MAC addresses, rather than IP addresses
- Error detection
  - Errors caused by signal attenuation, noise.
  - Receiver detecting presence of errors
- Error correction
  - Receiver correcting errors without retransmission
- Flow control
  - Pacing between adjacent sending and receiving nodes

Adaptors Communicating



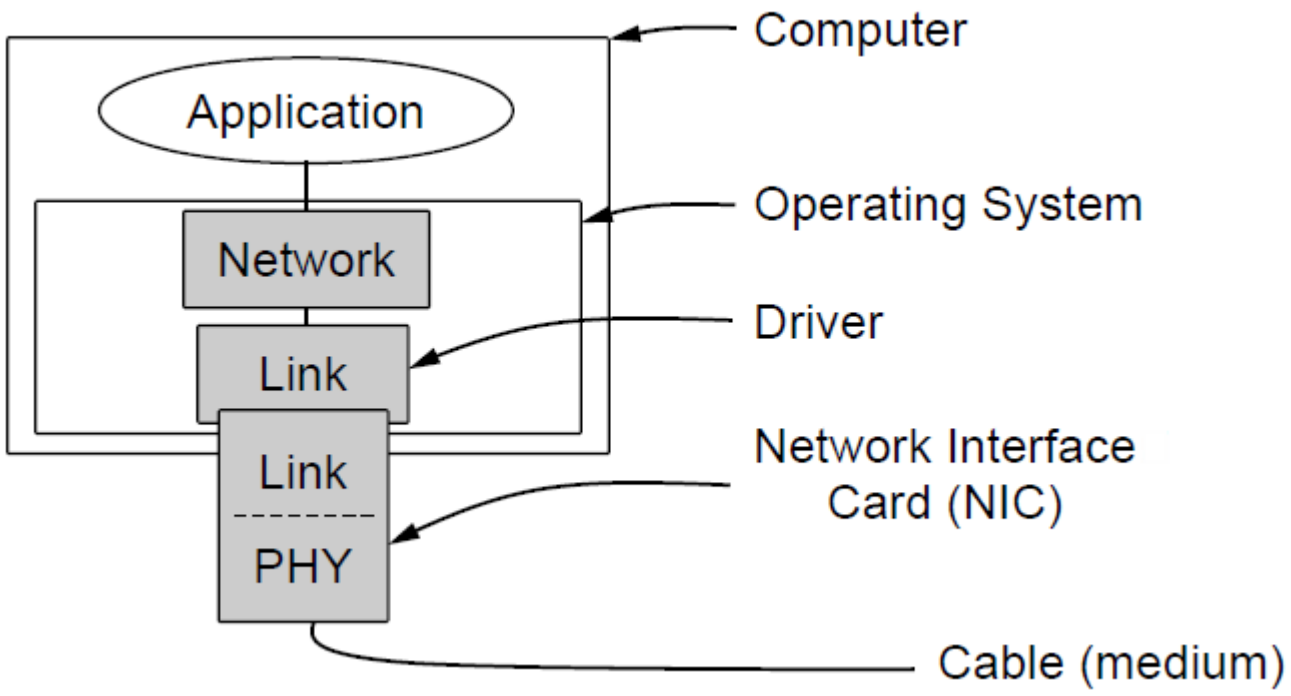
- Link layer implemented in adaptor (network interface card)
  - Ethernet card, PCMCI card, 802.11 card
- Sending side:
  - Encapsulates datagram in a frame
  - Adds error checking bits, flow control, etc.
- Receiving side
  - Looks for errors, flow control, etc.
  - Extracts datagram and passes to receiving node

Elementary Data Link Protocols

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

Elementary Data Link Protocols (2)

Implementation of the physical, data link, and network layers.



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.

Utopian Simplex Protocol (1)

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

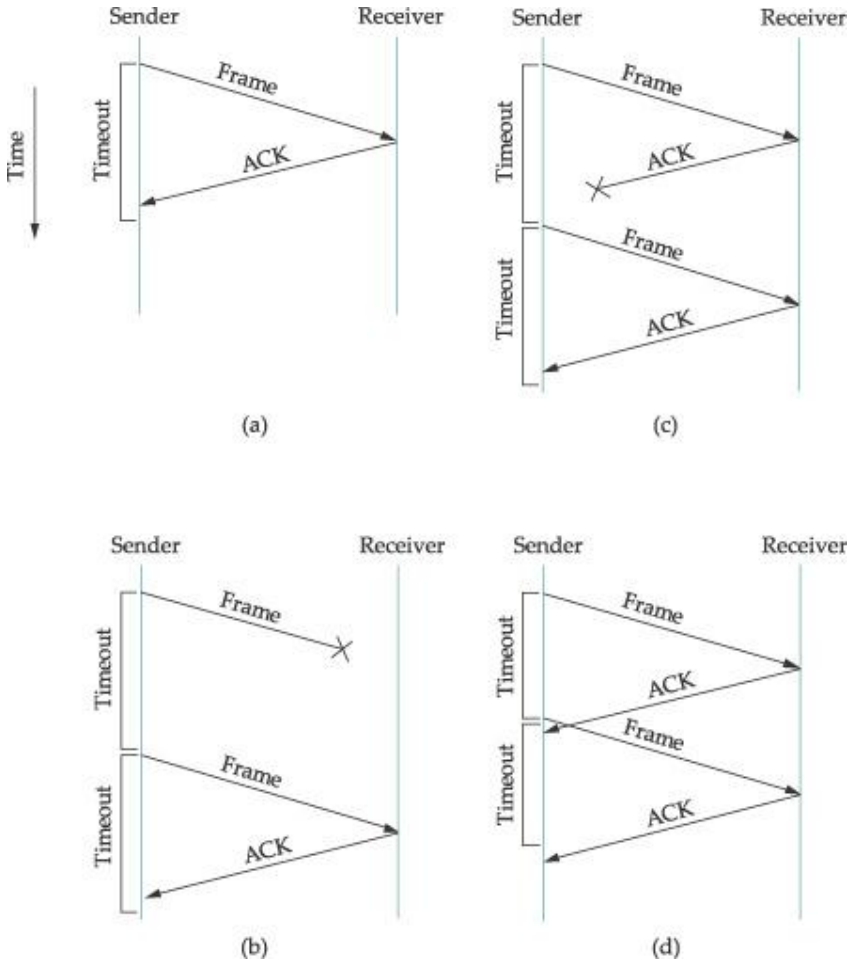
A utopian simplex protocol.

Utopian Simplex Protocol (2)

A utopian simplex protocol.

```
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 */
  }
}
```

Stop and Wait

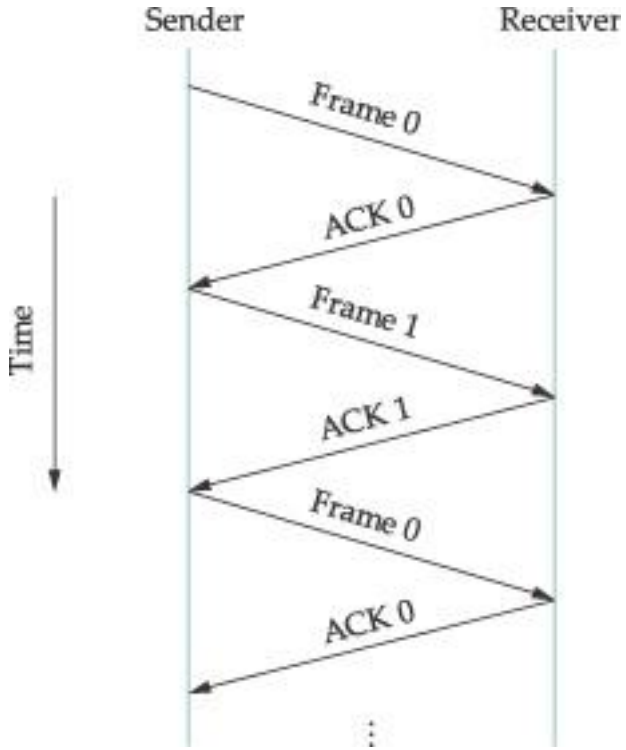


Duplicate Frames

Reliable Transmission

- Transfer frames without errors
  - Error Correction
  - Error Detection
  - Discard frames with error
- Acknowledgements and Timeouts
- Retransmission
- ARQ – Automatic Repeat Request

Stop and Wait with 1-bit Seq No



Stop and Wait Protocols

- Simple
- Low Throughput
  - One Frame per RTT

- Increase throughput by having more frames in flight
  - Sliding Window Protocol

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.

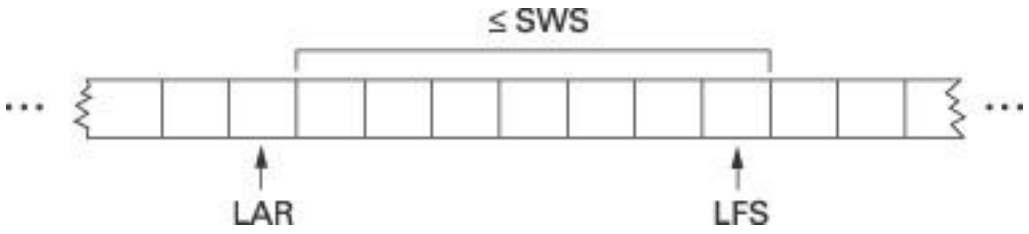
Sliding Window Protocol

- Sender assigns a sequence number –*SeqNum*
- Sender maintains three variables:
  - Send Window Size – SWS
  - Last Ack Received – LAR
  - Last Frame Sent – LFS
- Invariant LFS – LAR $\leq$ SWS
- When ACK arrives sender movers LAR to the right and thereby allowing the sender to transmit another frame
- Associate a timer with each frame it transmits

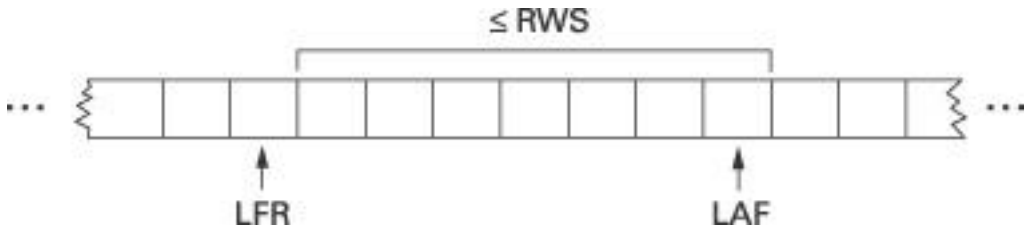
Sliding Window Protocol

- Receiver maintains three variables:
  - Receiver Window Size – RWS
  - Largest acceptable Frame Number – LAF
  - Last Frame Received – LFR
- Invariant LAF –LFR $\leq$ RWR
- When frame with SEQNum arrives
  - If SeqNum < LFR or SeqNum >LAF discard the frame
  - If LFR < SeqNum $\leq$ LAF then accept the frame
- SeqNumtoAck – largest seq no not yet acked.
  - Send this as ack.

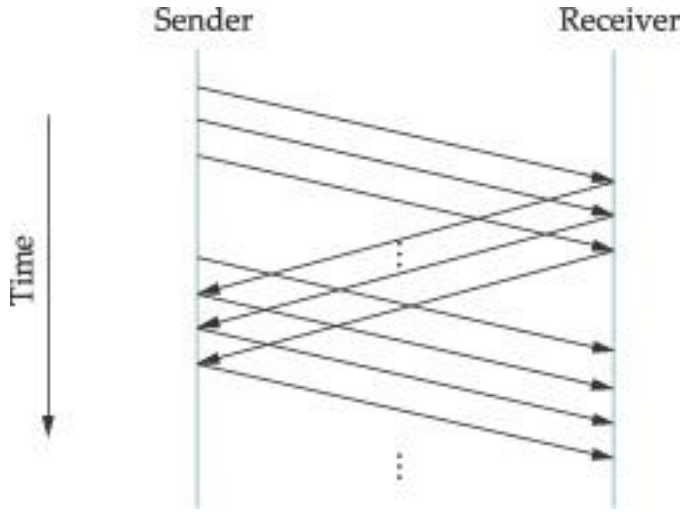
Sliding Window



Sliding Window on Sender



Sliding window on Receiver



Timeline

Sliding Window Protocols (1)

```
/* 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;

     . . .
```

Sliding Window Protocols (2)

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

. . .

Sliding Window Protocols (3)

```
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 */
       }
  }
}
```

Sliding Window

- Throughput – Keep the pipe full
- SWS selected to reflect how many frames we want in transit at any time
- Timeout results in a decrease in the amount of data in transit

- RWS – can be any value
  - If 1 implies the receiver does not buffer any out of order frames

Finite Sequence Numbers

- Can only use a finite number of bits for sequence number
- The number will roll over - MaxSeqNum

- If RWS = 1 then MaxSeqNum $\geq$ SWS + 1 is sufficient
- In general
  - SWS < (MaxSeqNum +1)/2

Functions of Sliding Window Protocol

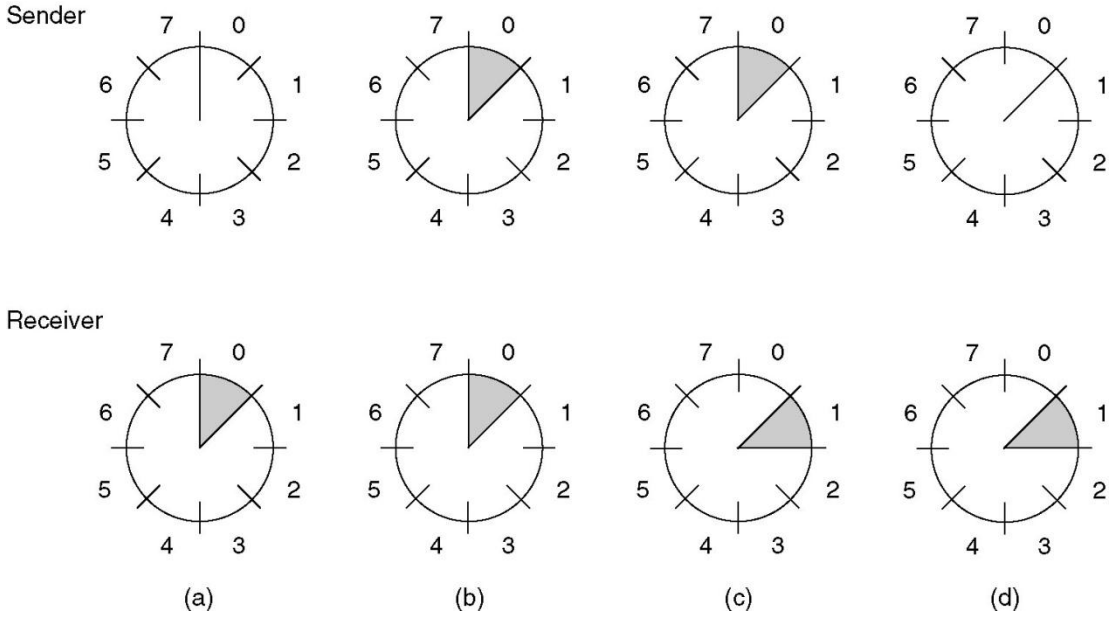
1. Reliably deliver frame across unreliable links
2. Deliver frames to higher levels in sequence
3. Flow Control

Separation of Concerns !!

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

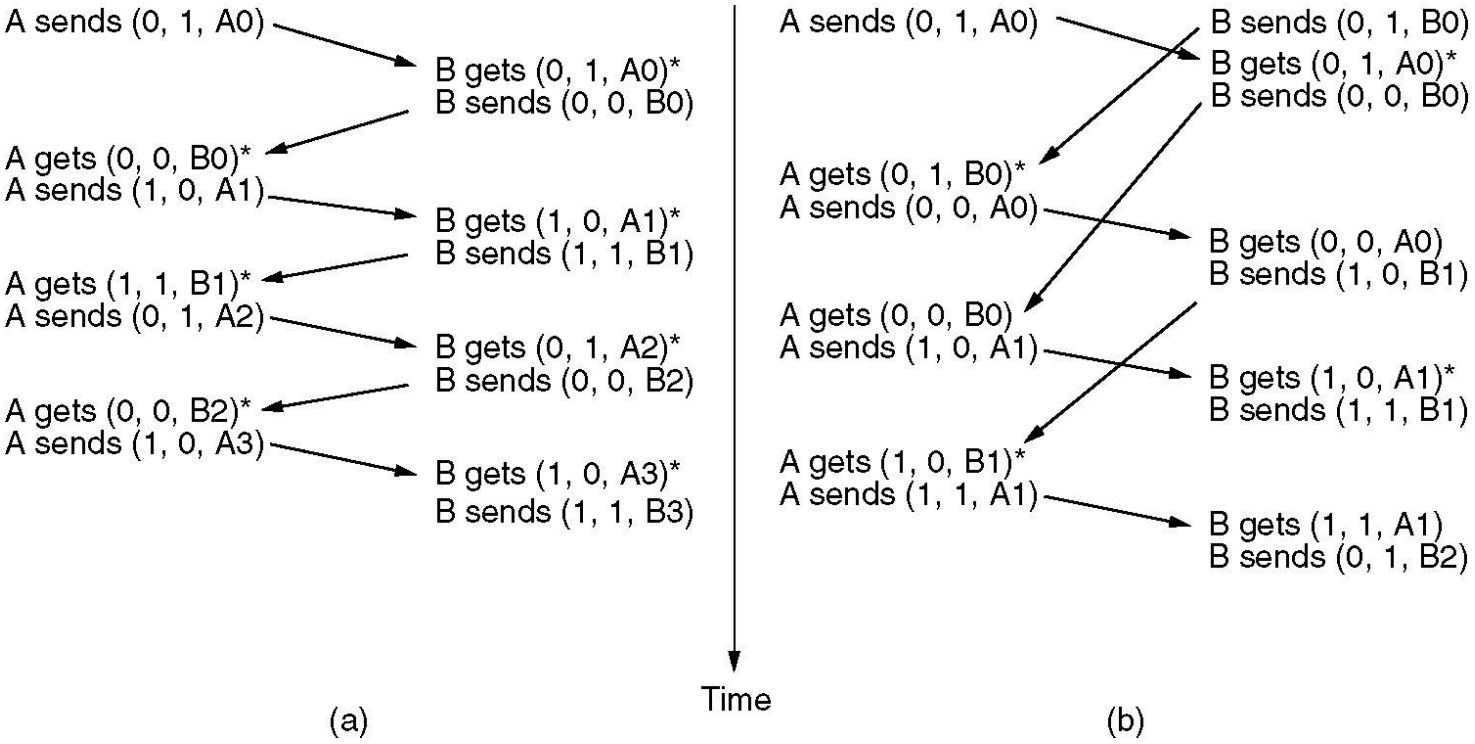
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /*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

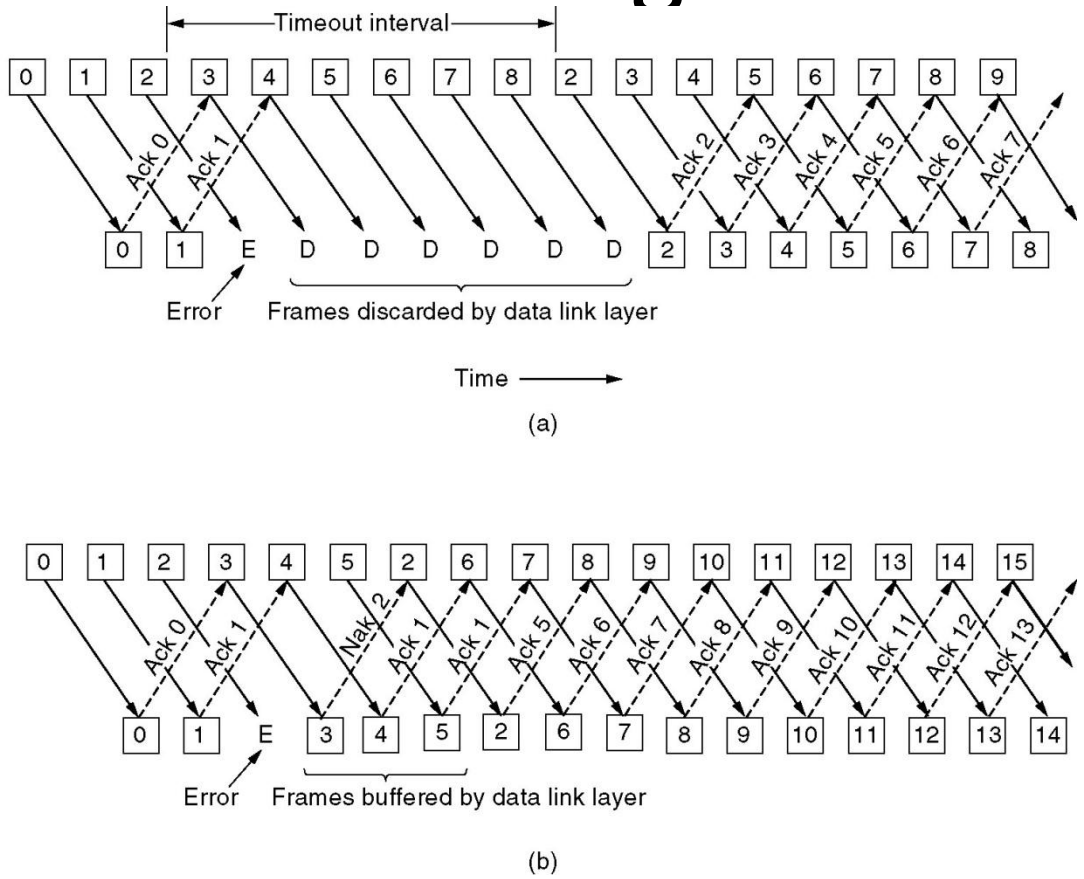
```
while (true) {
    wait_for_event(&event);                 /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {           /* a frame has arrived undamaged. */
        from_physical_layer(&r);            /* go get it */

        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 */
}
```


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

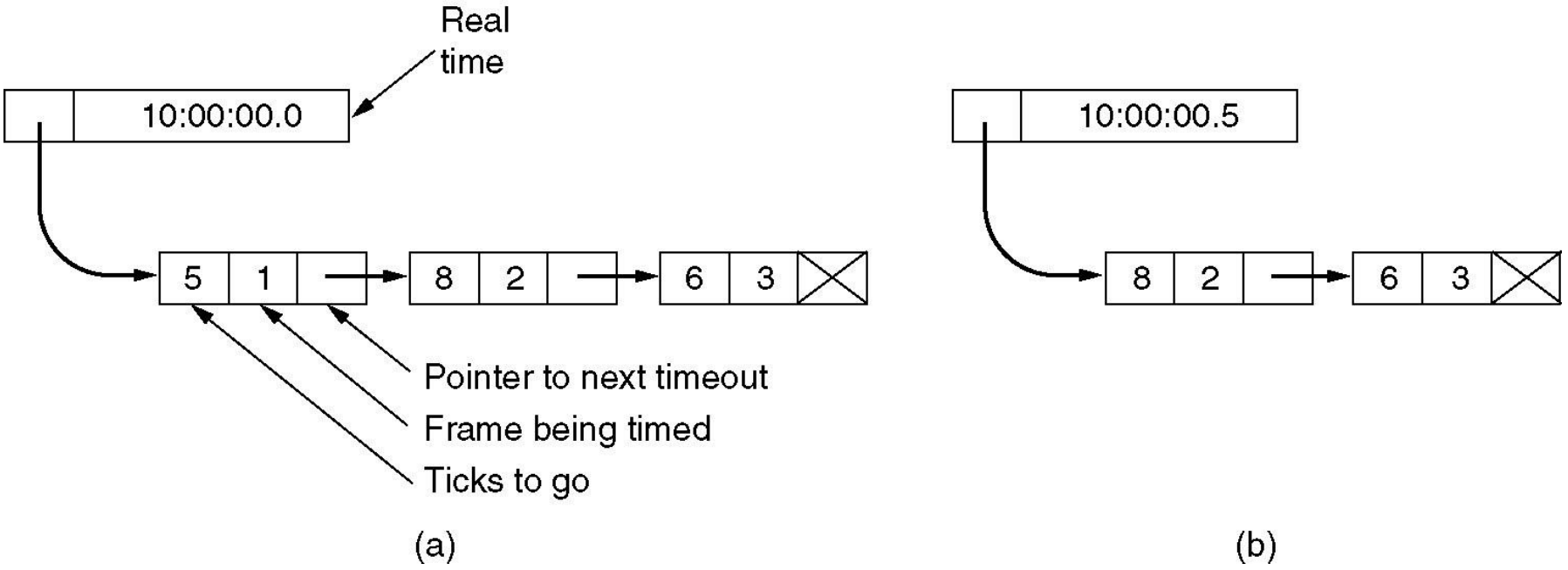
```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  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)



Protocol Using Selective Repeat (1)

A sliding window protocol using selective repeat.

```
/* Protocol 6 (Selective repeat) 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 protocol 5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}
```

Protocol Using Selective Repeat (2)

A sliding window protocol using selective repeat.

```
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 */
}
```

. . .

Protocol Using Selective Repeat (3)

A sliding window protocol using selective repeat.

```
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;
```

Protocol Using Selective Repeat (4)

A sliding window protocol using selective repeat.

```
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;
```

. . .

Protocol Using Selective Repeat (5)

A sliding window protocol using selective repeat.

```
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;
```

Protocol Using Selective Repeat (6)

A sliding window protocol using selective repeat.

```
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 */
```

. . .

Protocol Using Selective Repeat (7)

A sliding window protocol using selective repeat.

```
        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 */
        }
    }
}
```

. . .

Protocol Using Selective Repeat (8)

A sliding window protocol using selective repeat.

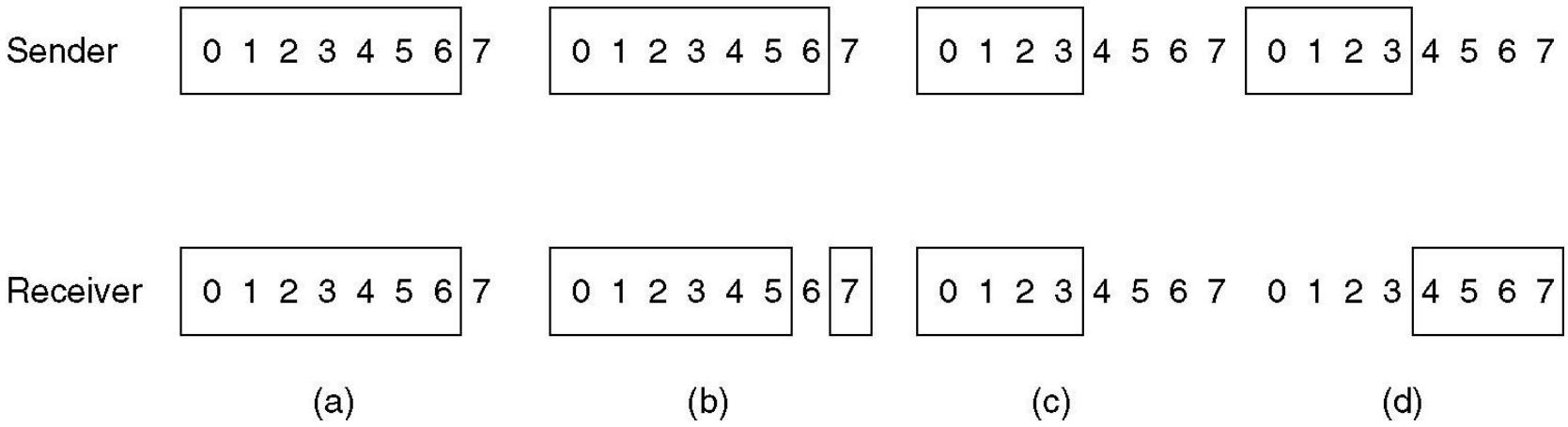
```
    if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
        send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

    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;
 . . .
```

Protocol Using Selective Repeat (9)

A sliding window protocol using selective repeat.

```
    case timeout:
         send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
         break;
    case ack_timeout:
         send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
    }
    if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
  }
}
```

A Sliding Window Protocol Using Selective Repeat (5)



(a) Initial situation with a window size seven.

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

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

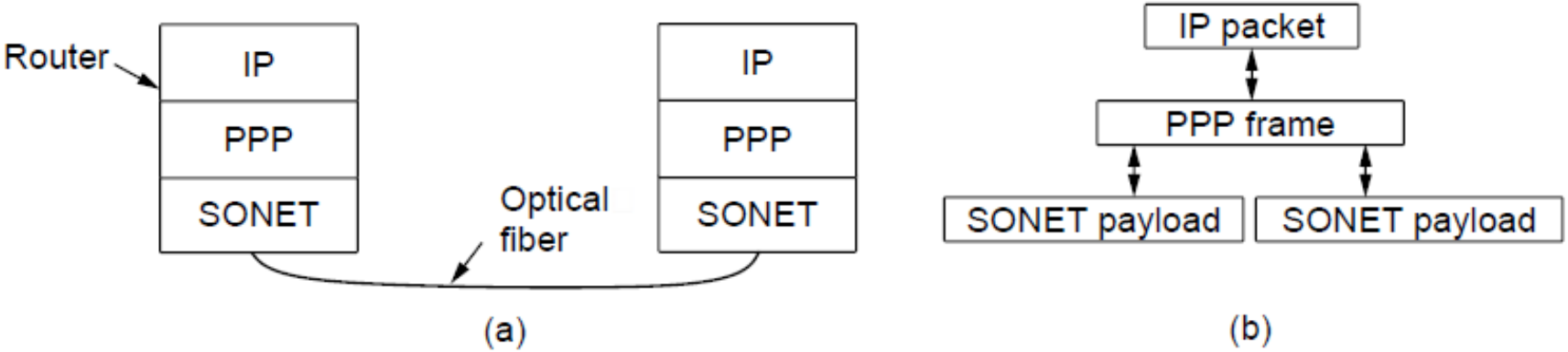
(d) After four frames sent and received, but not acknowledged.

Example Data Link Protocols

1. Packet over SONET
2. ADSL (Asymmetric Digital Subscriber Loop)

Packet over SONET (1)

Packet over SONET. (a) A protocol stack. (b)



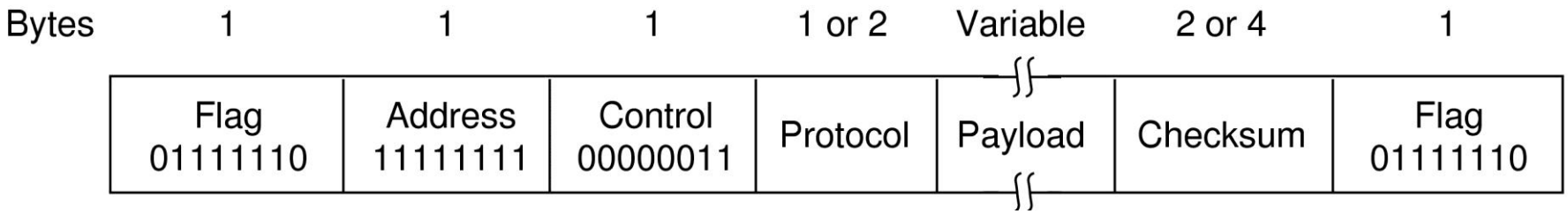
Packet over SONET (2)

PPP Features

1. Separate packets, error detection
2. Link Control Protocol
3. Network Control Protocol

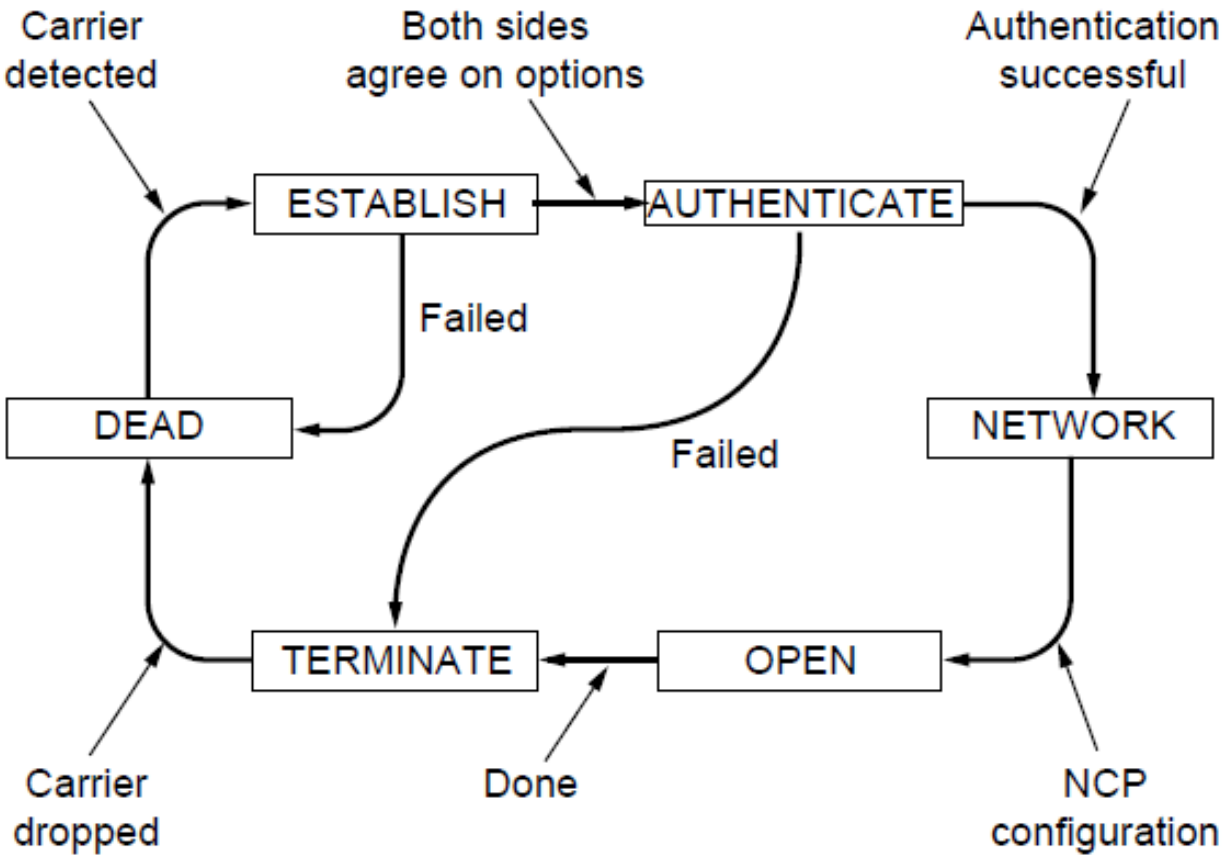
Packet over SONET (3)

The PPP full frame format for unnumbered mode operation



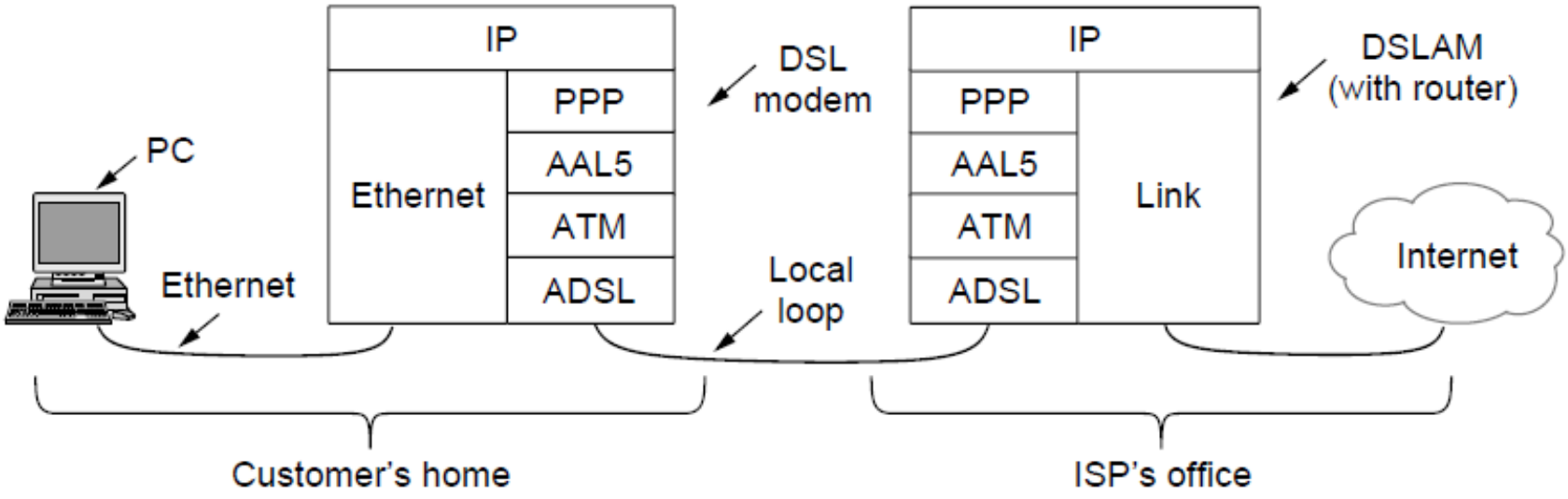
Packet over SONET (4)

State diagram for bringing a PPP link up and down



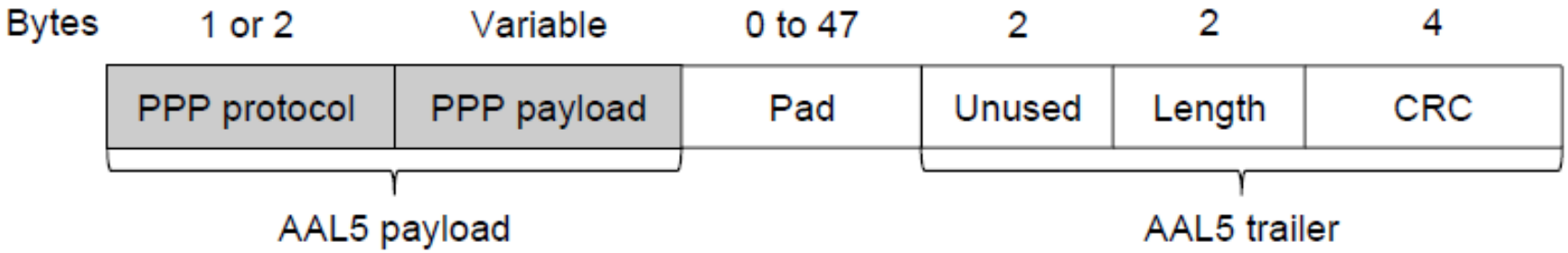
ADSL (Asymmetric Digital Subscriber Loop) (1)

ADSL protocol stacks.



ADSL (Asymmetric Digital Subscriber Loop) (1)

AAL5 frame carrying PPP data

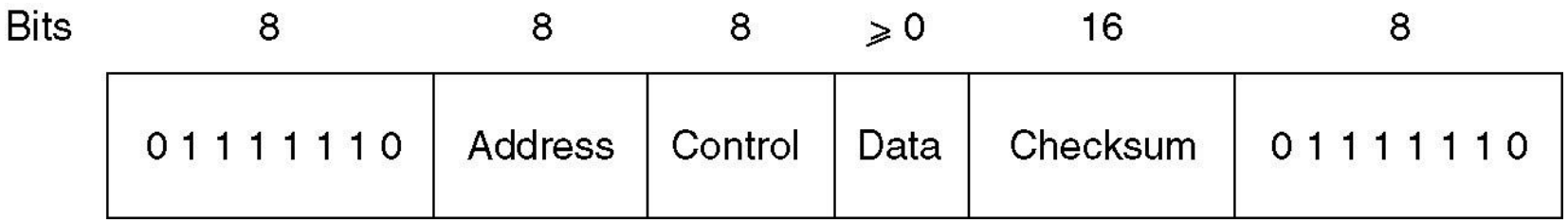


Example Data Link Protocols

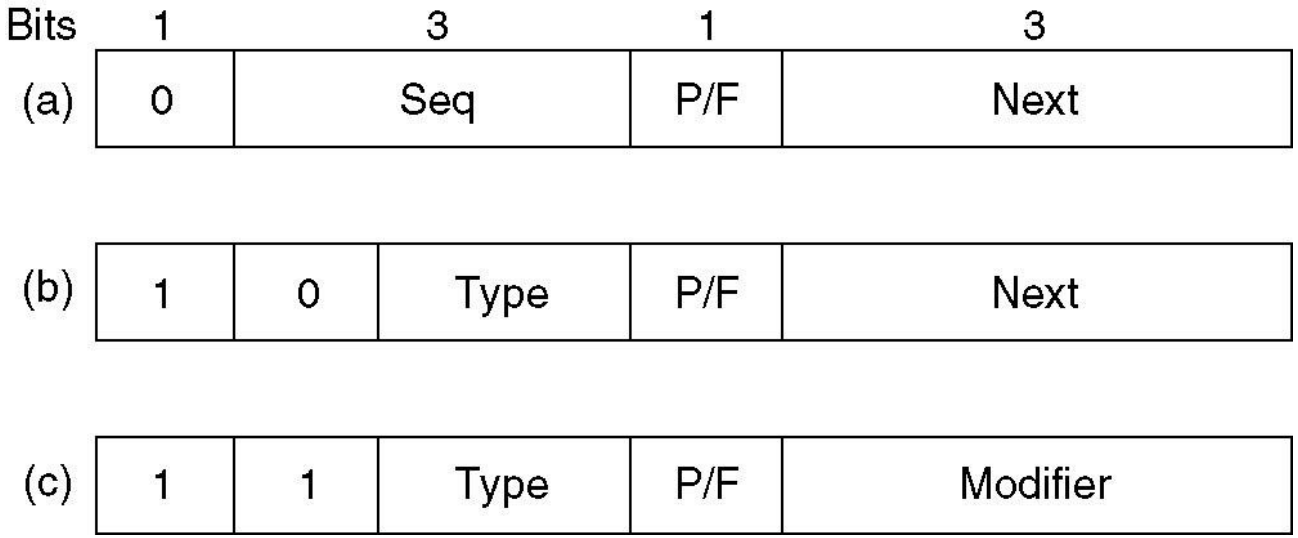
- HDLC – High-Level Data Link Control
- The Data Link Layer in the Internet

High-Level Data Link Control

Frame format for bit-oriented protocols.



High-Level Data Link Control (2)



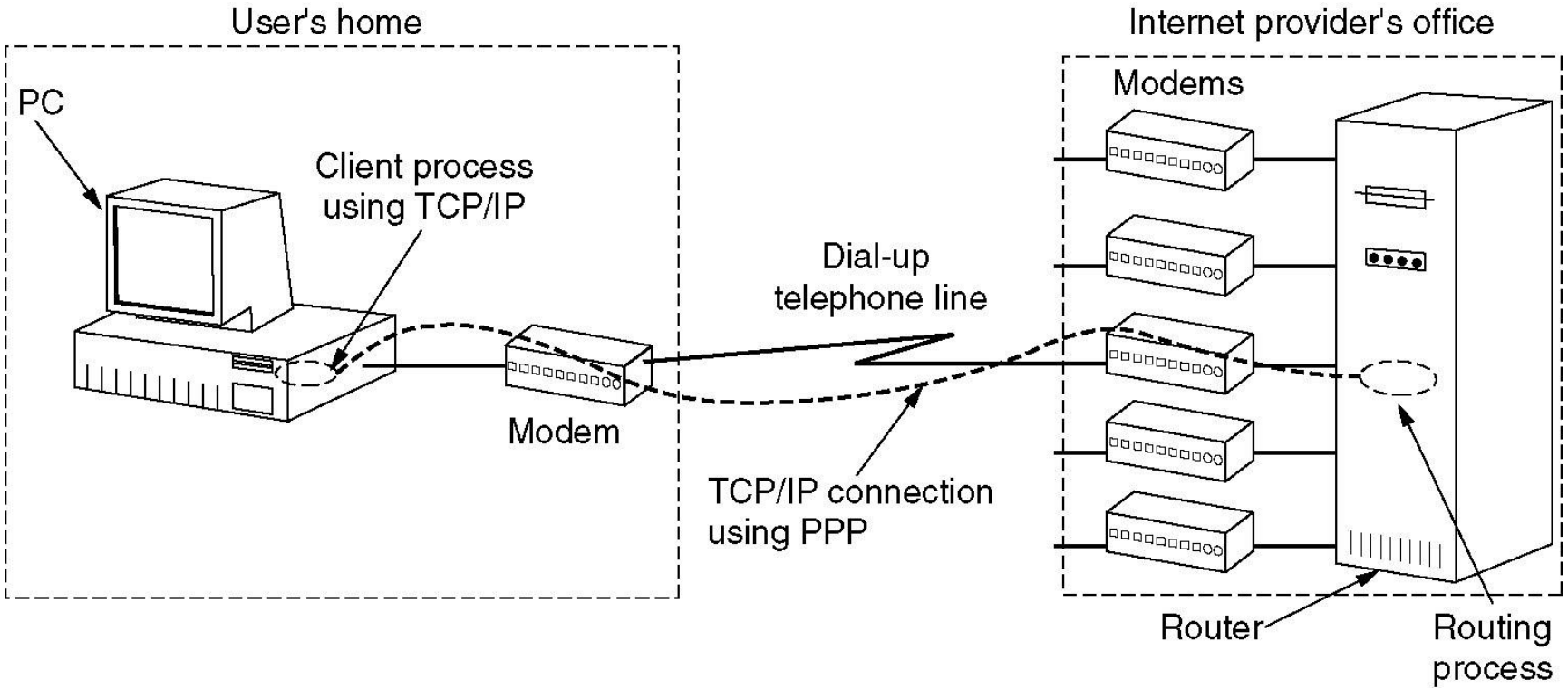
Control field of

(a) An information frame.

(b) A supervisory frame.

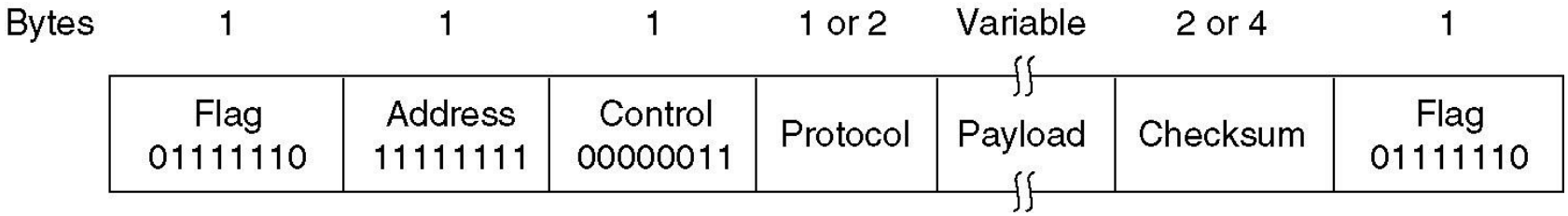
(c) An unnumbered frame.

The Data Link Layer in the Internet

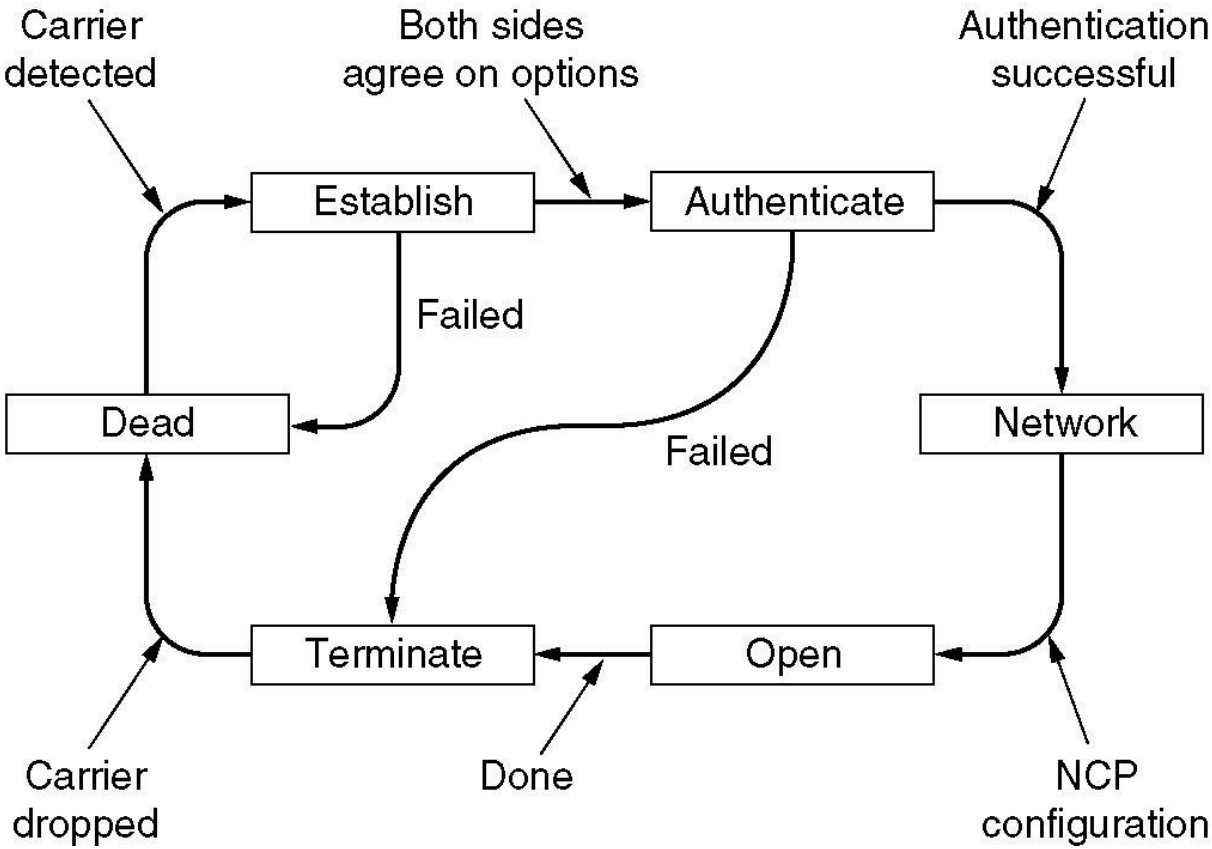


PPP – Point to Point Protocol

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)