CSMC 417

Computer Networks

Prof. Ashok K Agrawala

Set 4

The Data Link Layer

Chapter 3

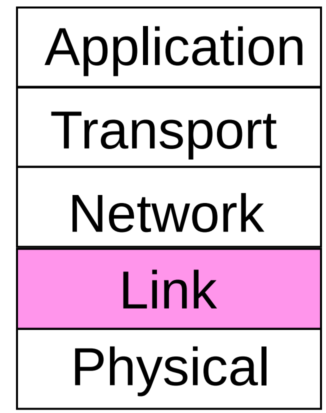
- Data Link Layer Design Issues
- Error Detection and Correction
- Elementary Data Link Protocols
- Sliding Window Protocols
- Example Data Link Protocols

Revised: August 2011

The Data Link Layer

Responsible for delivering frames of information over a single link

- Handles transmission errors and regulates the flow of data

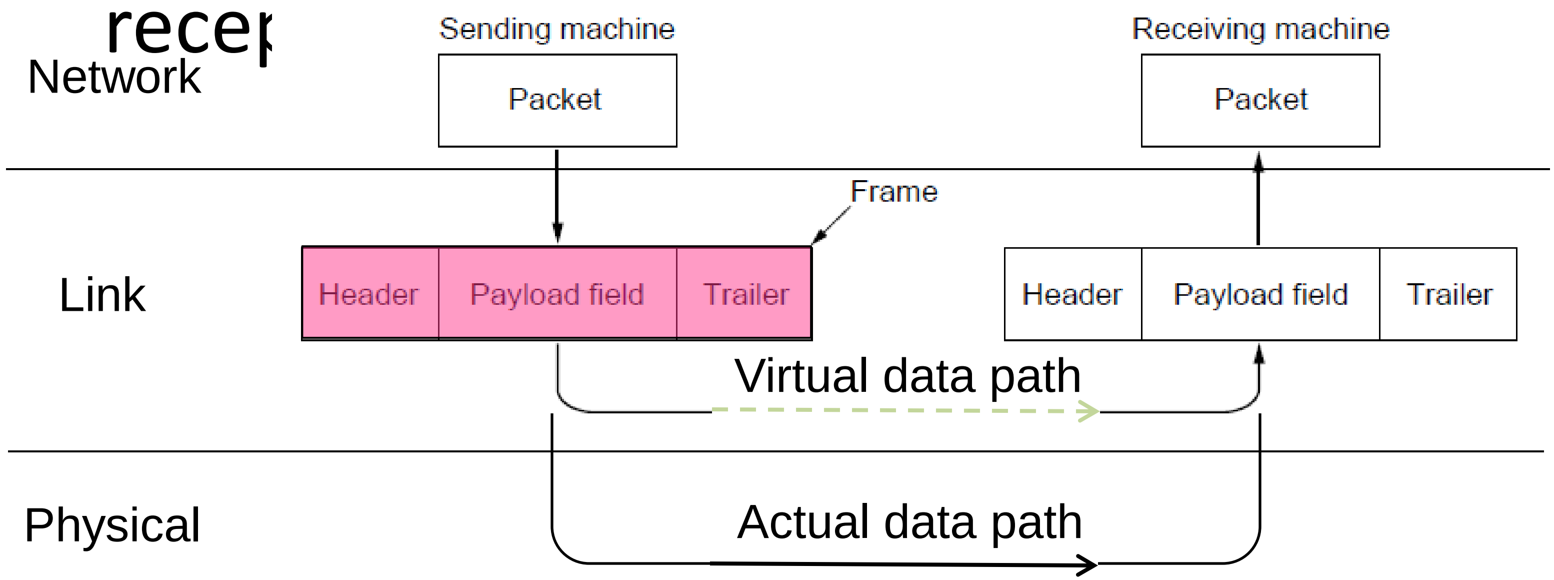


Data Link Layer Design Issues

- Frames »
- Possible services »
- Framing methods »
- Error control »
- Flow control »

Frames

Link layer accepts <u>packets</u> from the network layer, and encapsulates them into <u>frames</u> that it sends using the physical layer; rece



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

Possible Services

Unacknowledged connectionless service

- Frame is sent with no connection / error recovery
- Ethernet is example

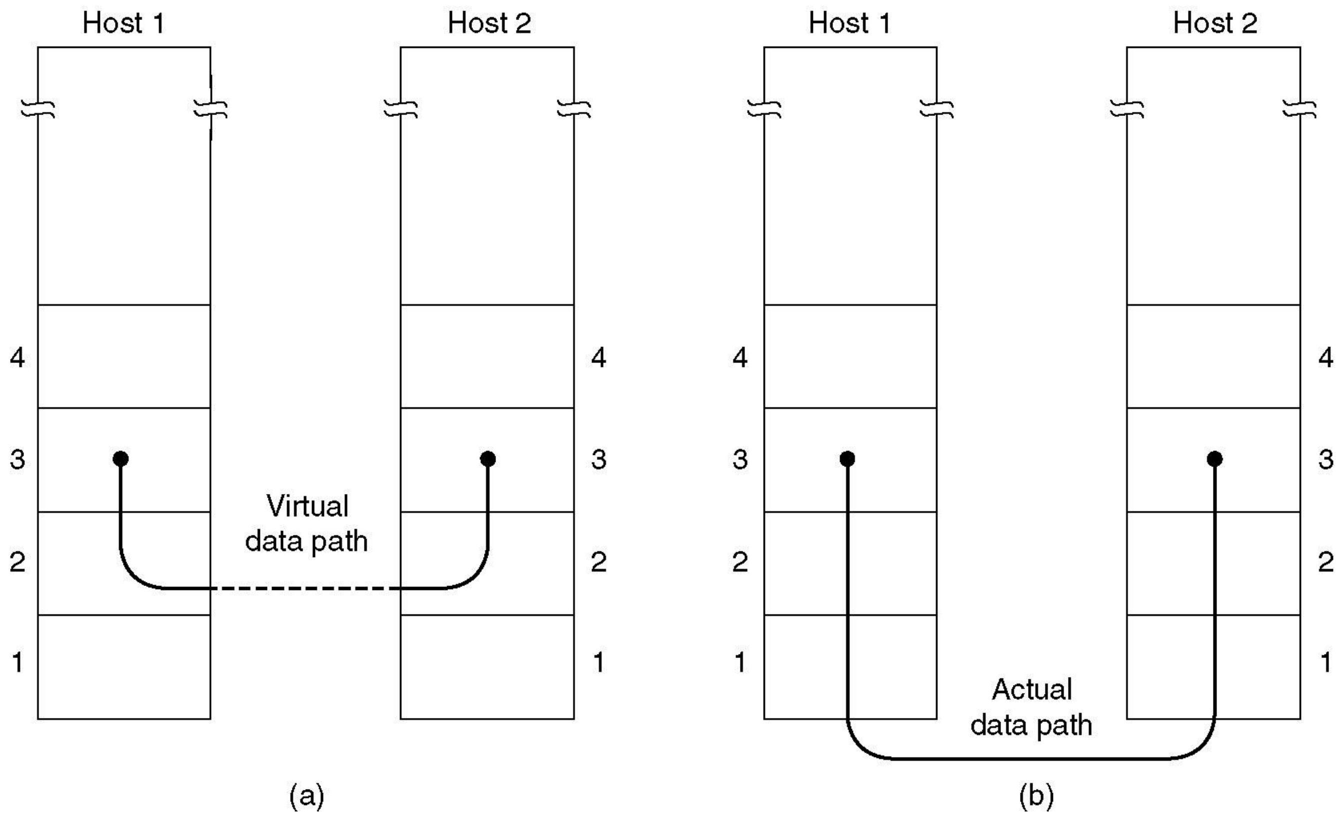
Acknowledged connectionless service

- Frame is sent with retransmissions if needed
- Example is 802.11

Acknowledged connection-oriented service

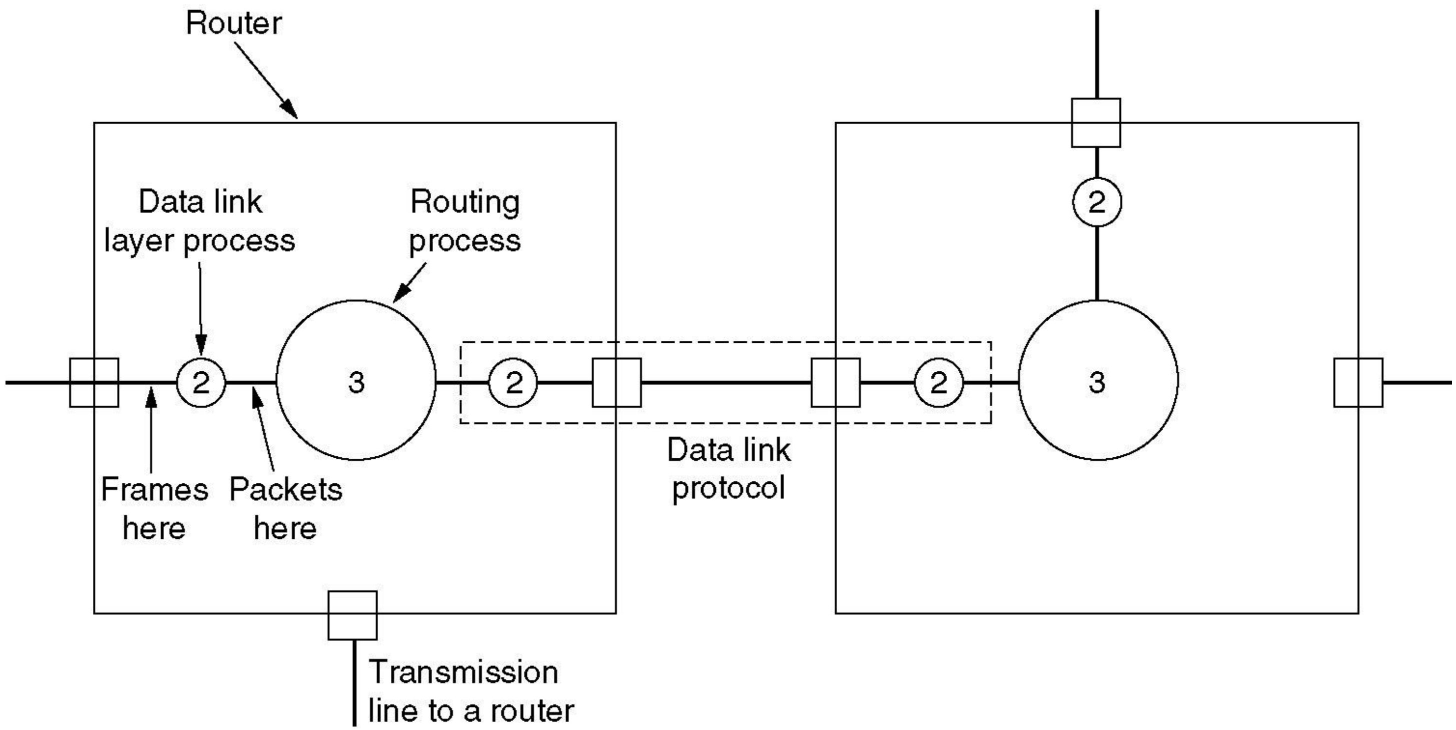
- Connection is set up; rare

Services Provided to Network Layer



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

Services Provided to Network Layer (2)



Framing Methods

- Byte count »
- Flag bytes with byte stuffing »
- Flag bits with bit stuffing »
- Physical layer coding violations
  - Use non-data symbol to indicate frame

Framing

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

01111110	Frame contents	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

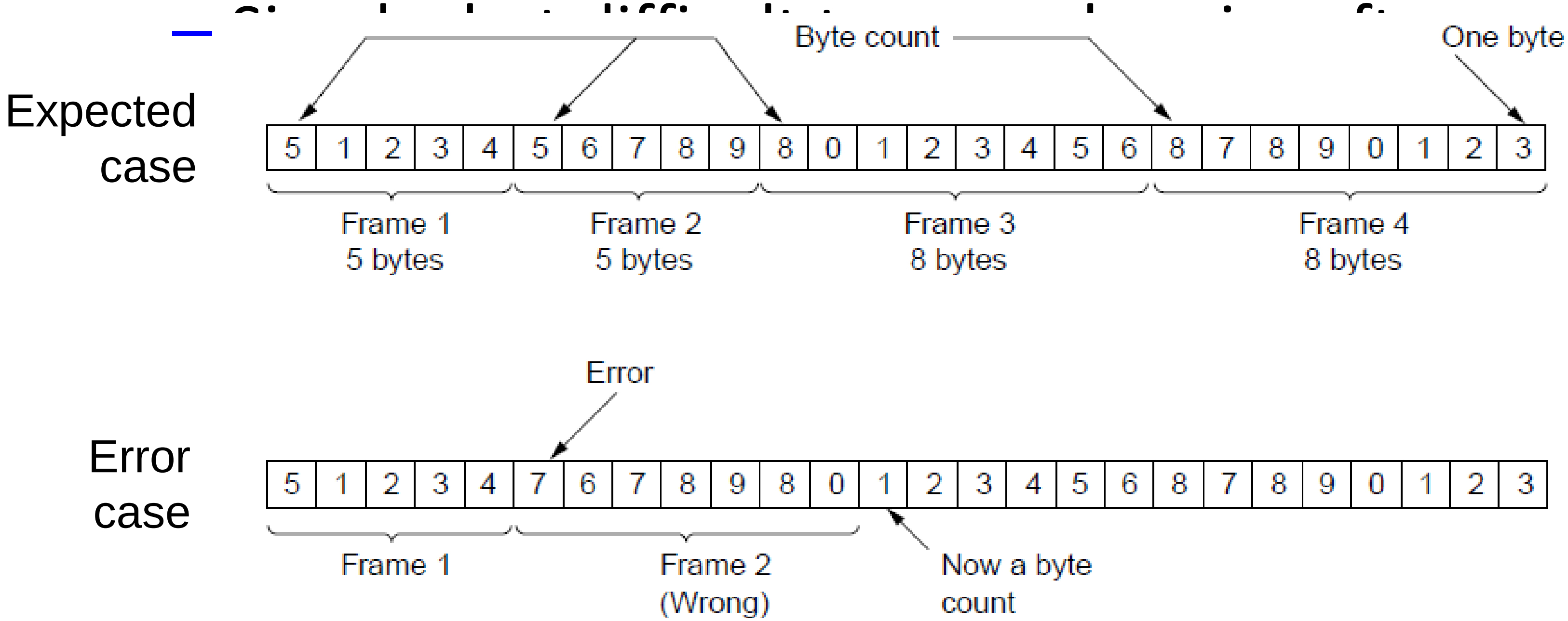
Bit Oriented Protocols

- 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 – Byte count

Frame begins with a count of the number of bytes in it

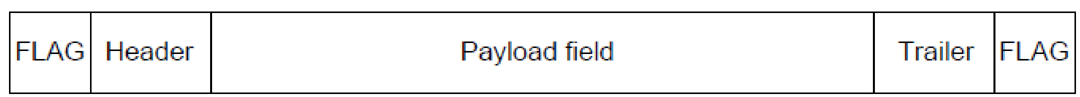


Framing – Byte stuffing

Special flag bytes delimit frames; occurrences of flags in the data must be stuffed (escaped)

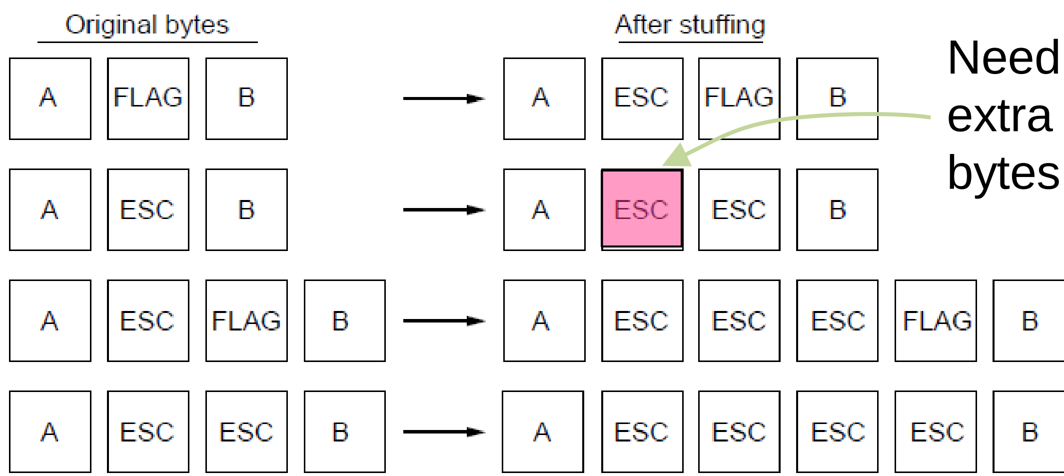
– Long

Frame format



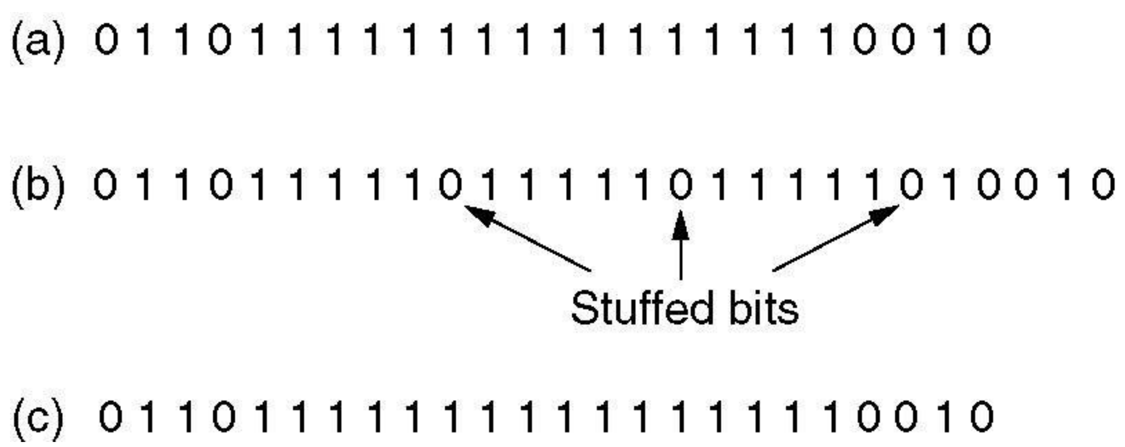
or

Stuffing examples



Need to escape extra ESCAPE bytes too!

Framing – Bit Oriented



Bit stuffing

(a) The original data.

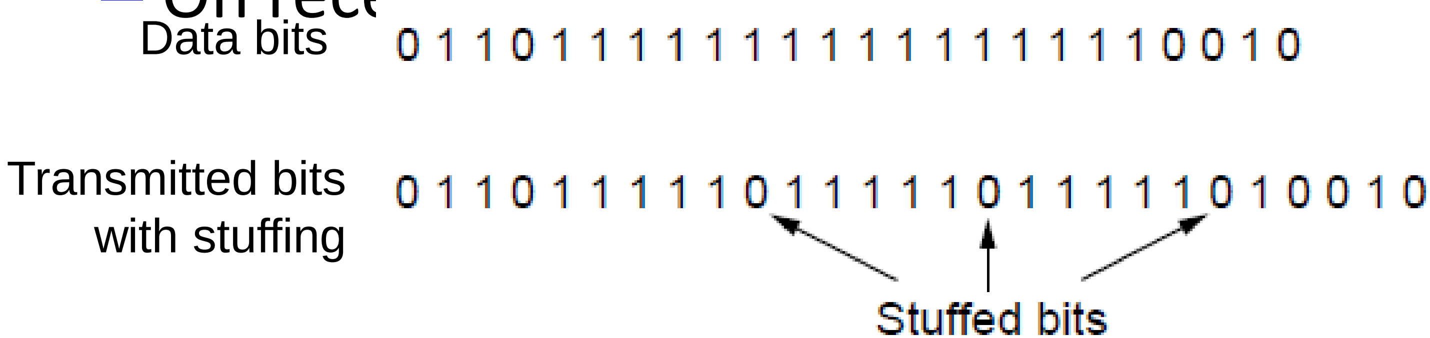
(b) The data as they appear on the line.

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

Framing – Bit stuffing

Stuffing done at the bit level:

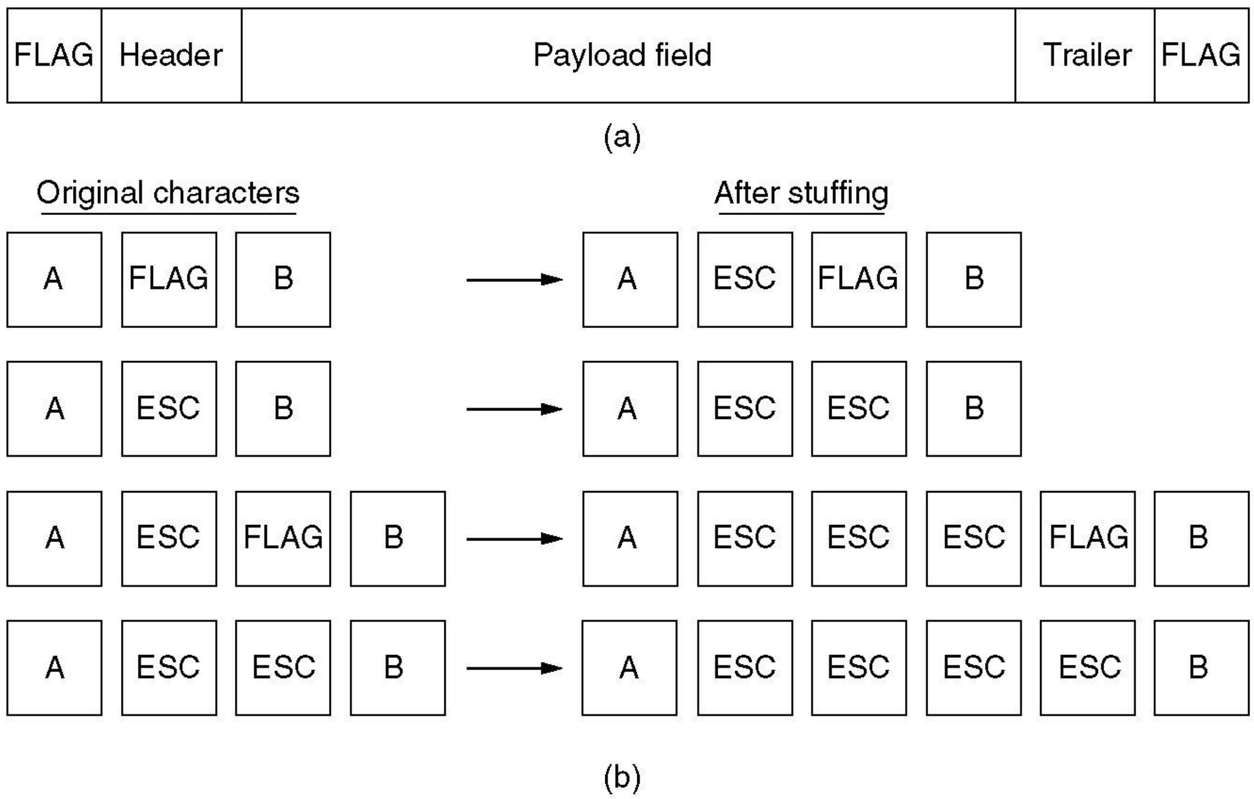
- Frame flag has six consecutive 1s (not shown)
- On transmit, after five 1s in the data, a 0 is added
- On rece



Byte-Oriented Protocols

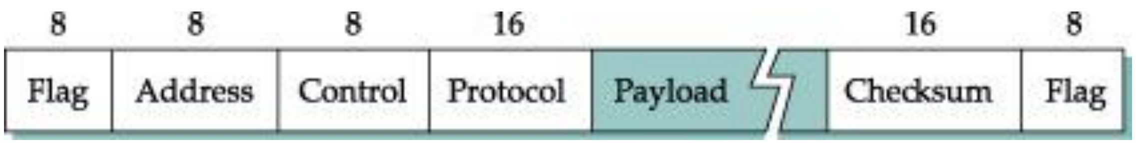
- Frame – a collection of bytes.
- Examples
  - BISYNC – Binary Synchronous Communication – IBM
  - DDCMP – Digital Data Communication Message Protocol
  - PPP – Point-to-Point
- Sentinel Based – Use special character as marker
  - BISYNC
    - SYN and SOH
    - STX and ETX
    - DLE as escape character.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

Framing

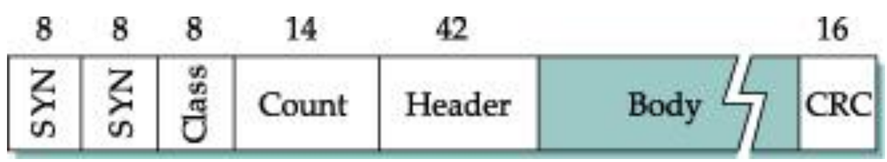


(a) A frame delimited by flag bytes.
(b) Four examples of byte sequences before and after stuffing.

Frame Structure



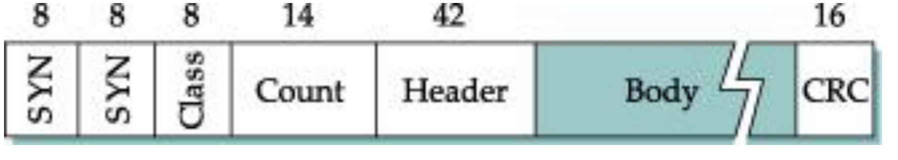
PPP Frame Format



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



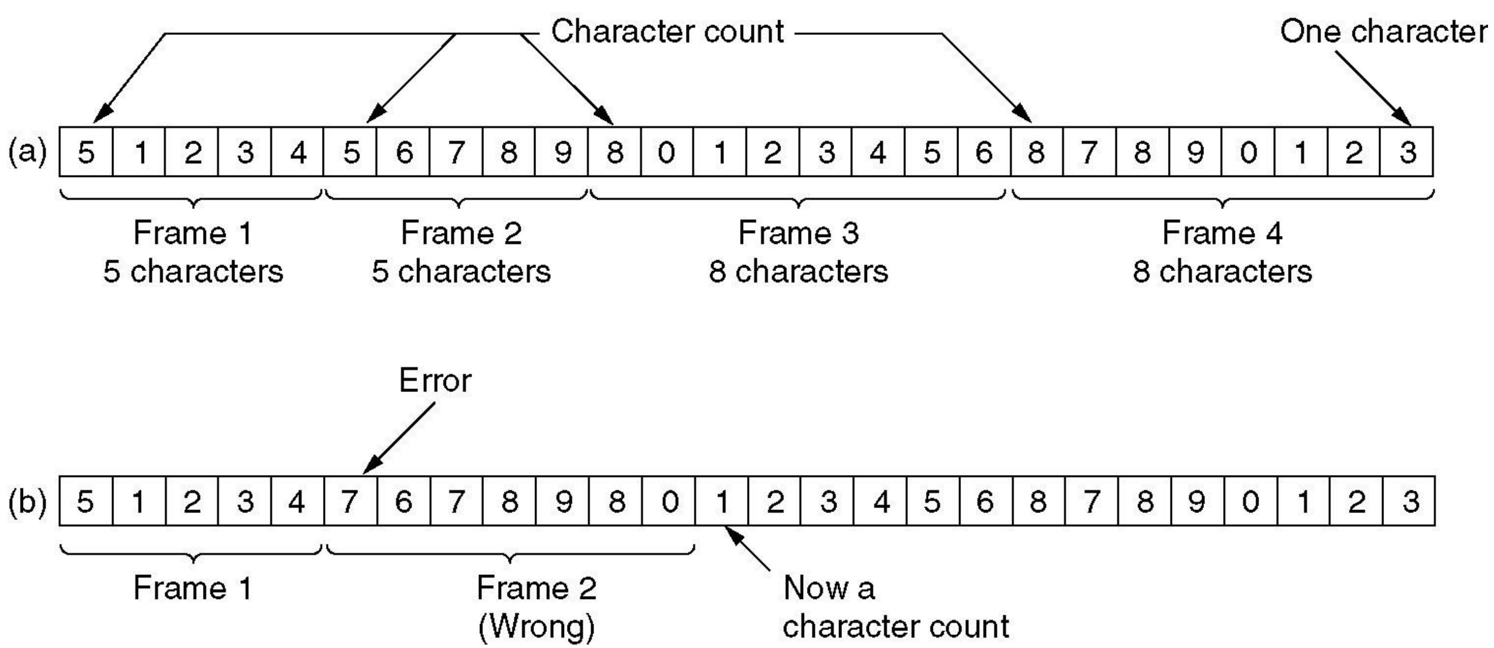
DDCMP Frame Format

Framing

A character stream.

(a) Without errors.

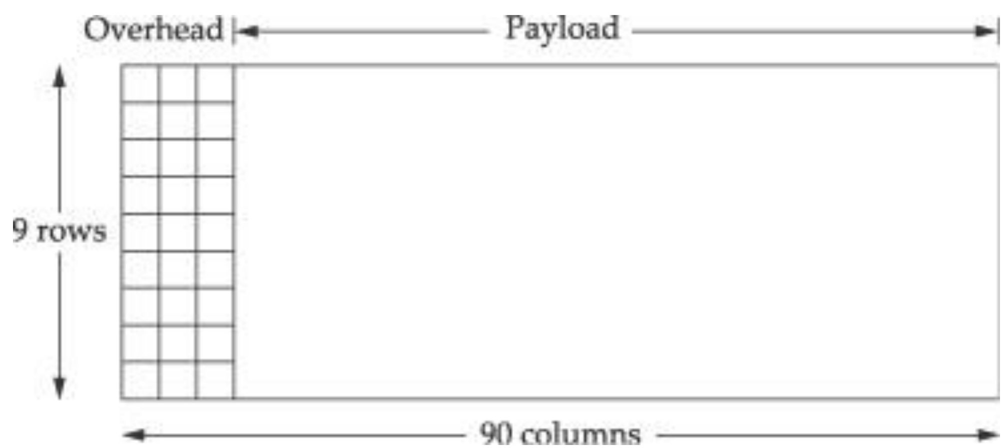
(b) With one error.



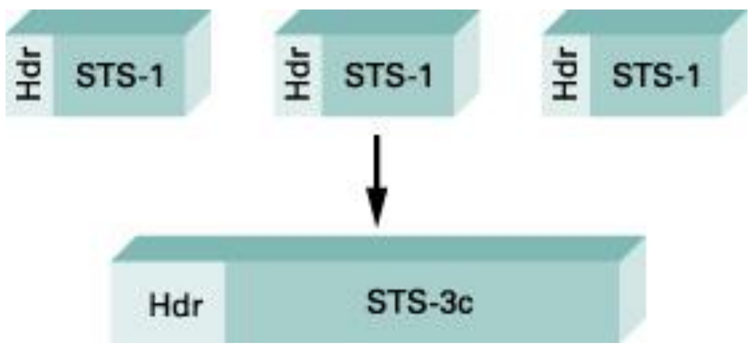
Clock-Based Framing (SONET)

- 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 Control

Error control repairs frames that are received in error

- Requires errors to be detected at the receiver
- Typically retransmit the unacknowledged frames
- Timer protects against lost acknowledgements

Detecting errors and retransmissions are next topics.

Flow Control

Prevents a fast sender from out-pacing a slow receiver

- Receiver gives feedback on the data it can accept
- Rare in the Link layer as NICs run at "wire speed"
  - Receiver can take data as fast as it can be sent

Flow control is a topic in the Link and Transport layers.

Error Detection and Correction

Error codes add structured redundancy to data so errors can be either detected, or corrected.

Error correction codes:

- Hamming codes »
- Binary convolutional codes »
- Reed-Solomon and Low-Density Parity Check codes
  - Mathematically complex, widely used in real systems

Error detection codes:

- Parity »
- Checksums »
- Cyclic redundancy codes »

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

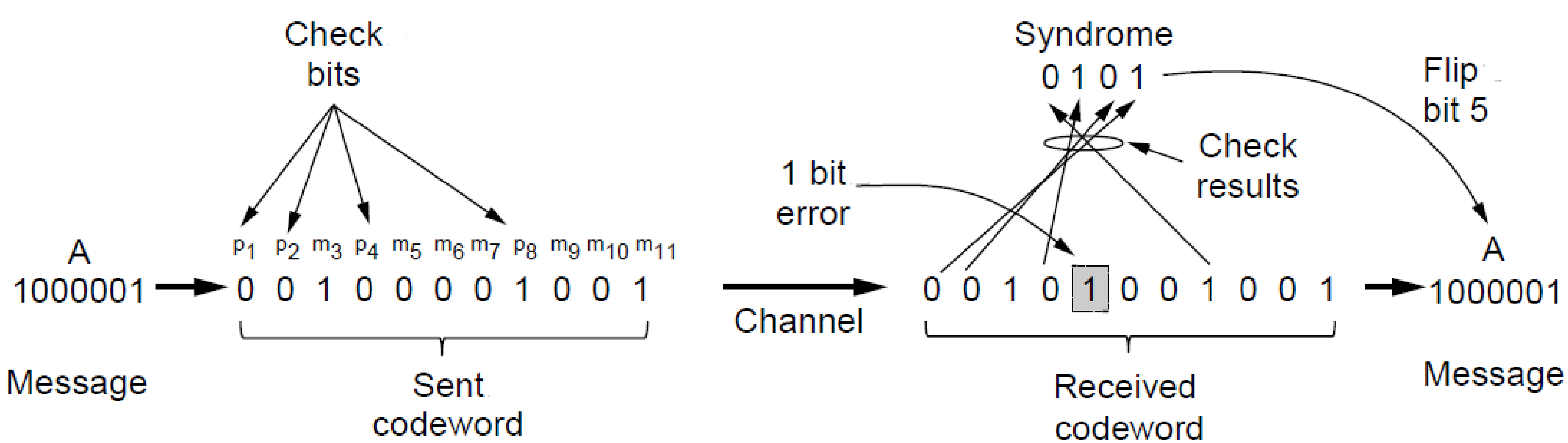
Error Bounds – Hamming distance

- Code turns data of n bits into codewords of n+k bits
- <u>Hamming distance</u> is the minimum bit flips to turn one valid codeword into any other valid one.
  - Example with 4 codewords of 10 bits (n=2, k=8):
    - 0000000000, 0000011111, 1111100000, and 1111111111
    - Hamming distance is 5
- Bounds for a code with distance:
  - 2d+1 – can correct d errors (e.g., 2 errors above)
  - d+1 – can detect d errors (e.g., 4 errors above)

Error Correction – Hamming code

Hamming code gives a simple way to add check bits and correct up to a single bit error:

– Check bits are parity over subsets of the



(11, 7) Hamming code adds 4 check bits and can correct 1 error

Error-Correcting Codes

Use of a

errors.

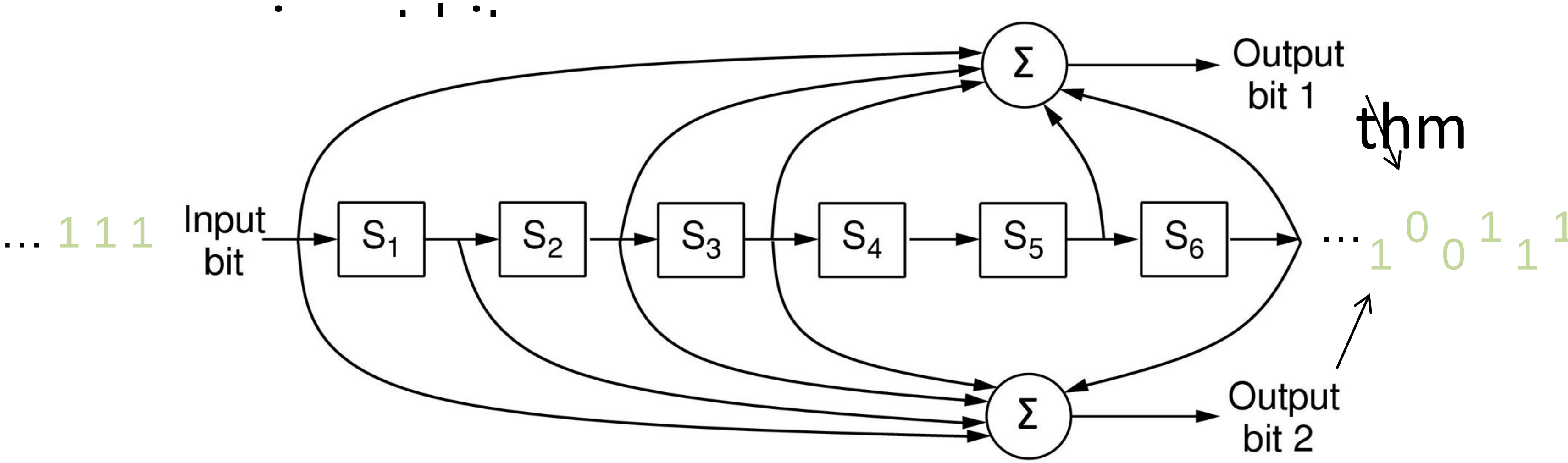
Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Error Correction – Convolutional codes

Operates on a stream of bits, keeping internal state

- Output stream is a function of all preceding



Popular NASA binary convolutional code (rate = ½) used in 802.11

Error Detection – Parity (1)

Parity bit is added as the modulo 2 sum of data bits

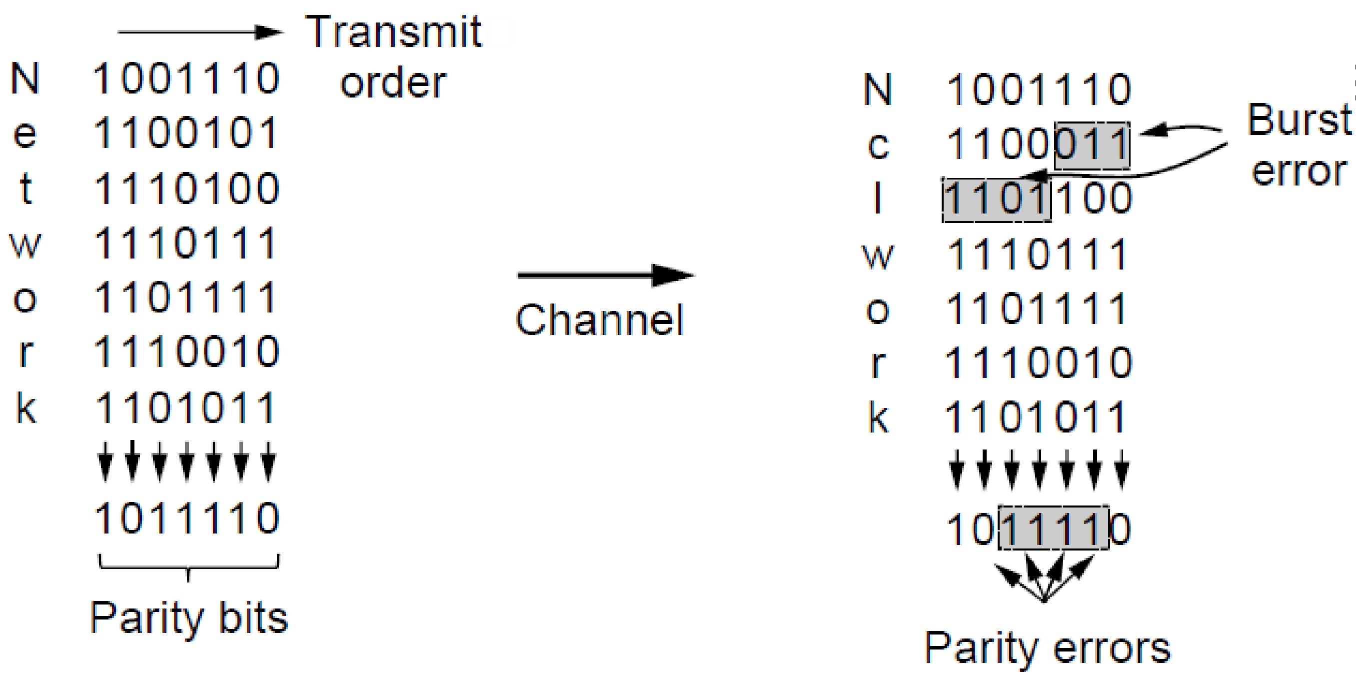
- Equivalent to XOR; this is even parity
- Ex: 1110000 → 11100001
- Detection checks if the sum is wrong (an error)

Simple way to detect an *odd* number of errors

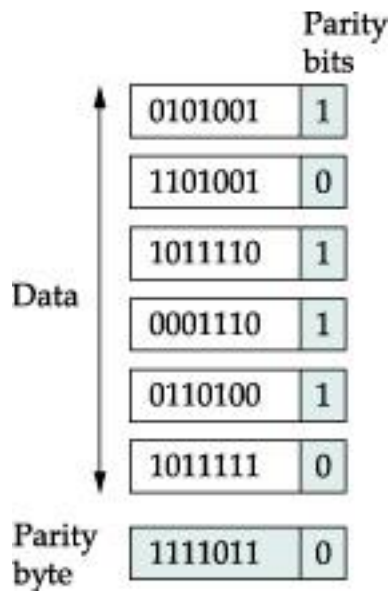
- Ex: 1 error, 11100$\underline{1}$01; detected, sum is wrong
- Ex: 3 errors, 11$\underline{011}$001; detected sum is wrong
- Ex: 2 errors, 1110$\underline{11}$01; *not detected*, sum is right!
- Error can also be in the parity bit itself
- Random errors are detected with probability ½

Error Detection – Parity (2)

- Interleaving of N parity bits detects burst errors up to N
  - Each parity sum is made over non-adjacent bits
  - se it



Two Dimensional Parity



Error Detection – Checksums

Checksum treats data as N-bit words and adds N check bits that are the modulo 2^N sum of the words

- Ex: Internet 16-bit 1s complement checksum

Properties:

- Improved error detection over parity bits
- Detects bursts up to N errors
- Detects random errors with probability $1\text{-}2^N$
- Vulnerable to systematic errors, e.g., added zeros

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)

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

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

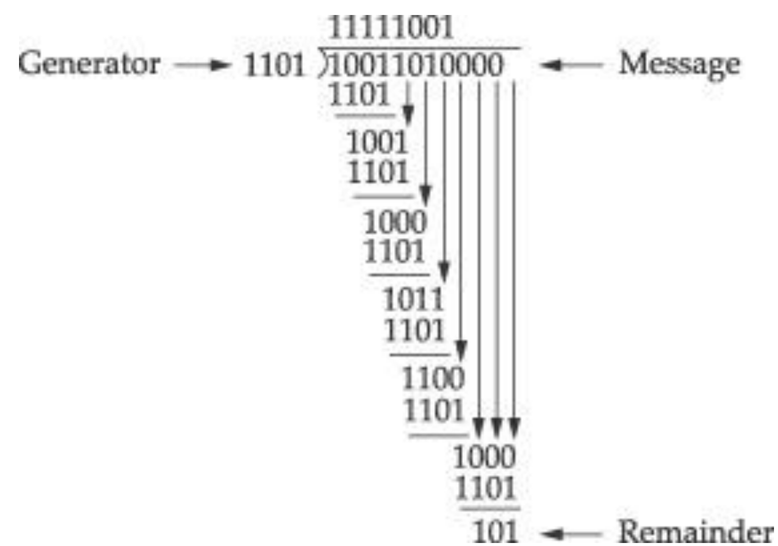
CRC Basis

- 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

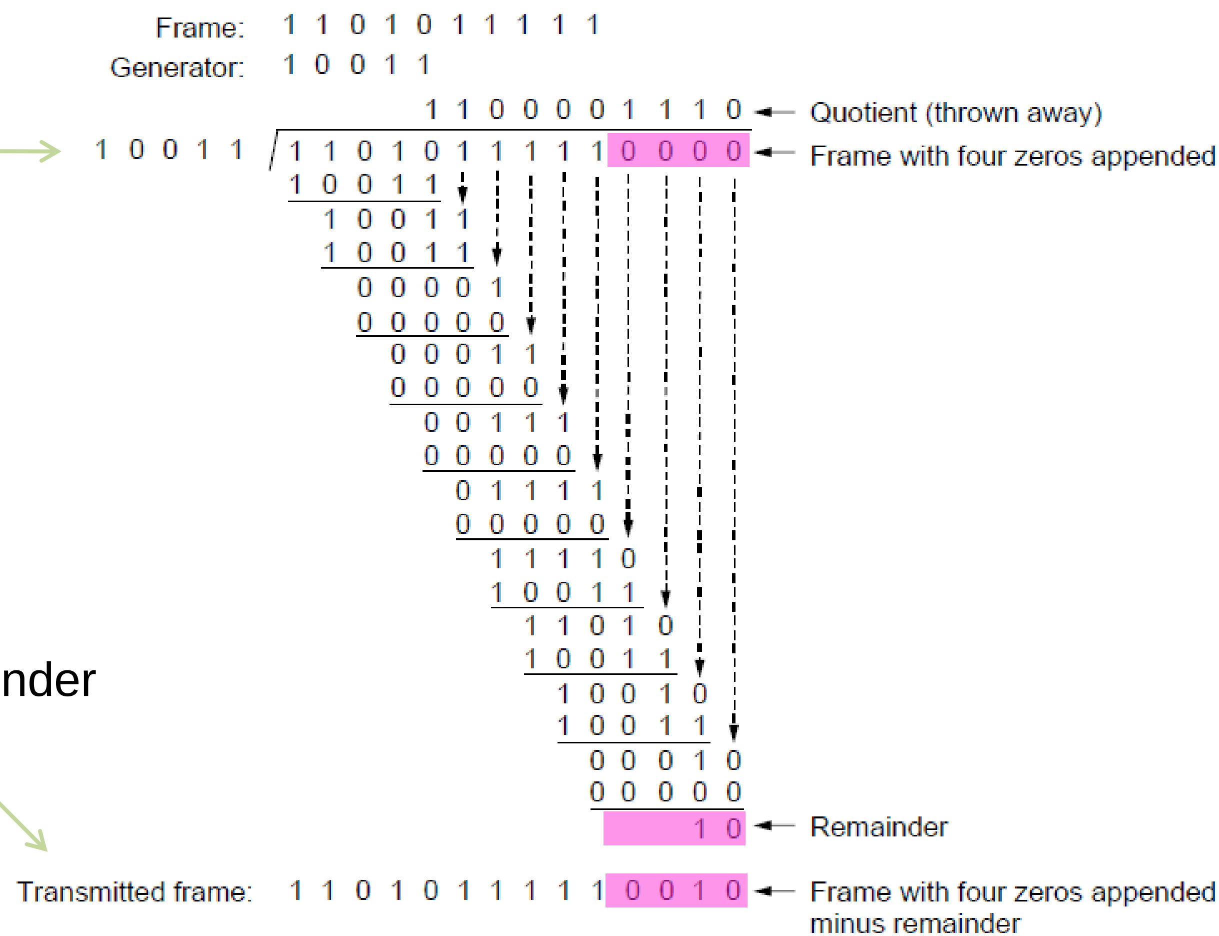
CRC	C(x)
CRC-8	$x^8 + x^2 + X^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2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

Error Detection – CRCs (1)

- Adds bits so that transmitted frame viewed as a polynomial is polynomial

Start by adding 0s to frame and try dividing

Offset by any reminder to make it evenly divisible



Error Detection – CRCs (2)

Based on standard polynomials:

- Ex: Ethernet 32-bit CRC is defined by:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + 1$$

- Computed with simple shift/XOR circuits

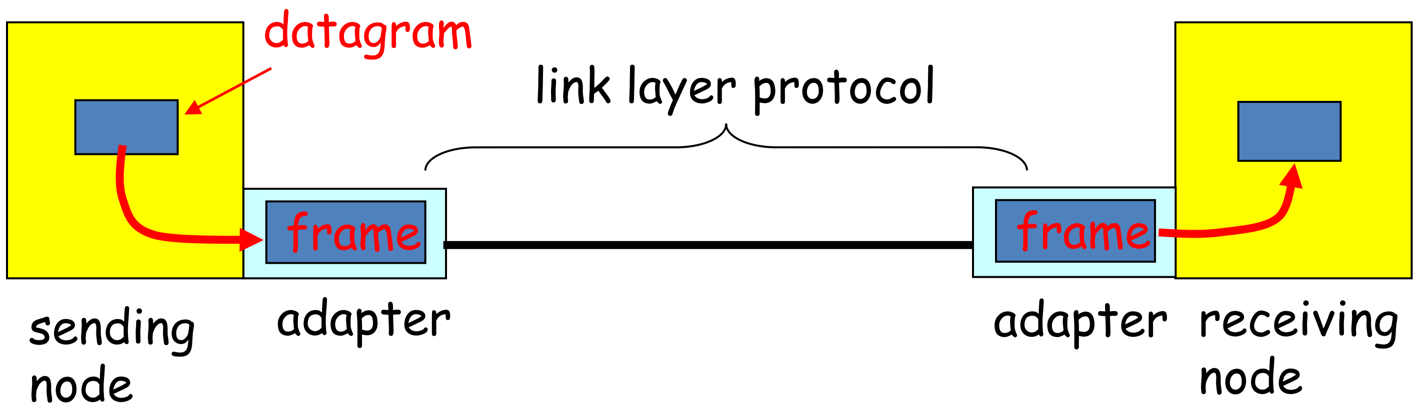
Stronger detection than checksums:

- E.g., can detect all double bit errors
- Not vulnerable to systematic 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



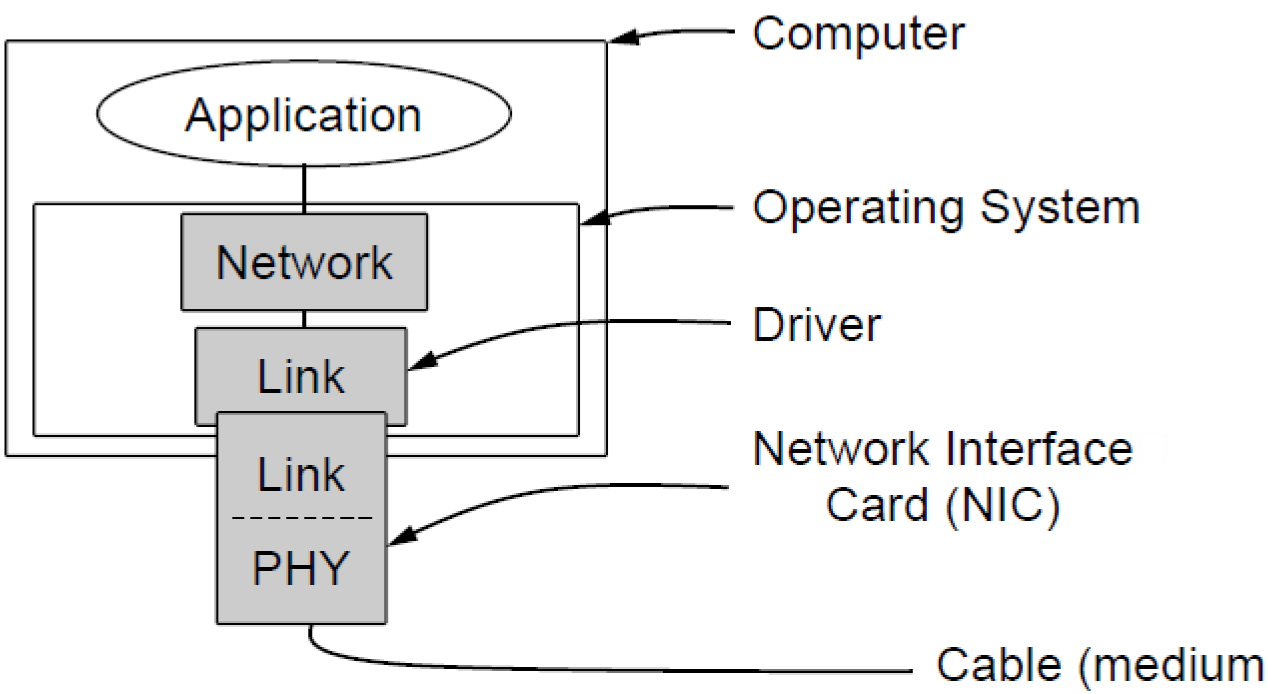
- Link layer implemented in adaptor (network interface card)
  - Ethernet card, PCMCI card, 802.11 card
- Sending side:
  - Encapsulates datagram in a frame
  - Adds error checking bits, flow control, etc.
- Receiving side
  - Looks for errors, flow control, etc.
  - Extracts datagram and passes to receiving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

Elementary Data Link Protocols

- Link layer environment »
- Utopian Simplex Protocol »
- Stop-and-Wait Protocol for Error-free channel »
- Stop-and-Wait Protocol for Noisy channel »

Link layer environment (1)

Commonly implemented as NICs and OS drivers: network layer (IP) is often OS



Link layer environment (2)

- Link layer protocol implementations use library functions
  - See code (`protocol.h`) for more details

Group	Library Function	Description
Network layer	from_network_layer(&packet) to_network_layer(&packet) enable_network_layer() disable_network_layer()	Take a packet from network layer to send Deliver a received packet to network layer Let network cause “ready” events Prevent network “ready” events
Physical layer	from_physical_layer(&frame) to_physical_layer(&frame)	Get an incoming frame from physical layer Pass an outgoing frame to physical layer
Events & timers	wait_for_event(&event) start_timer(seq_nr) stop_timer(seq_nr) start_ack_timer() stop_ack_timer()	Wait for a packet / frame / timer event Start a countdown timer running Stop a countdown timer from running Start the ACK countdown timer Stop the ACK countdown timer

Protocol Definitions

```
#define MAX_PKT 1024                                    /* determines packet size in bytes */

typedef enum {false, true} boolean;                     /* boolean type */
typedef unsigned int seq_nr;                            /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;/*    packet definition */
typedef enum {data, ack, nak} frame_kind;               /* frame_kind definition */

typedef struct {                                        /* frames are transported in this layer */
  frame_kind kind;                                      /* what kind of a frame is it? */
  seq_nr seq;                                           /* sequence number */
  seq_nr ack;                                           /* acknowledgement number */
  packet info;                                          /* the network layer packet */
} frame;
```

Continued →

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

Protocol Definitions (ctd.)

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

```
/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: Increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Utopian Simplex Protocol

An optimistic protocol (p1) to get us started

- Assumes no errors, and receiver as fast as sender
- Considers one-way data transfer

```
void sender1(void)
{
  frame s;
  packet buffer;

  while (true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
  }
}
```

Sender loops blasting frames

```
void receiver1(void)
{
  frame r;
  event_type event;

  while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
  }
}
```

Receiver loops eating frames

- That's it, no error or flow control …

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)

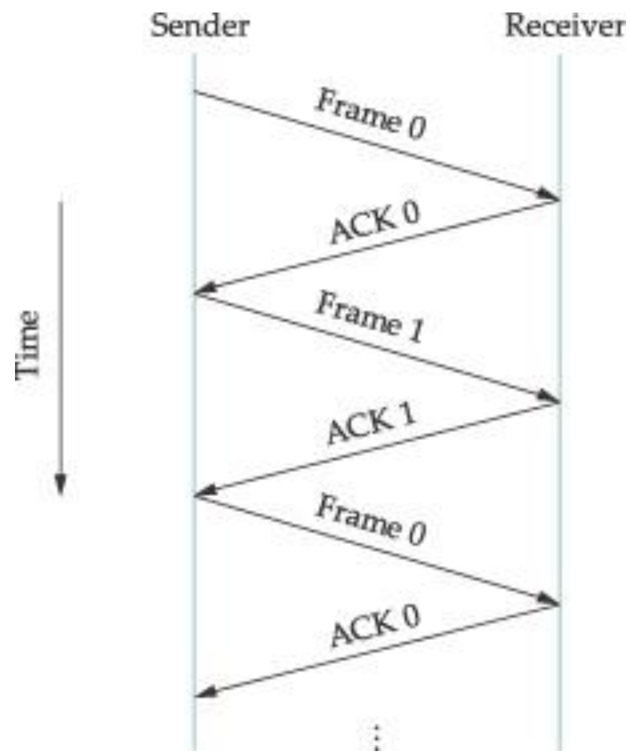
A utopian simplex protocol.

```
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 */
  }
}
```

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

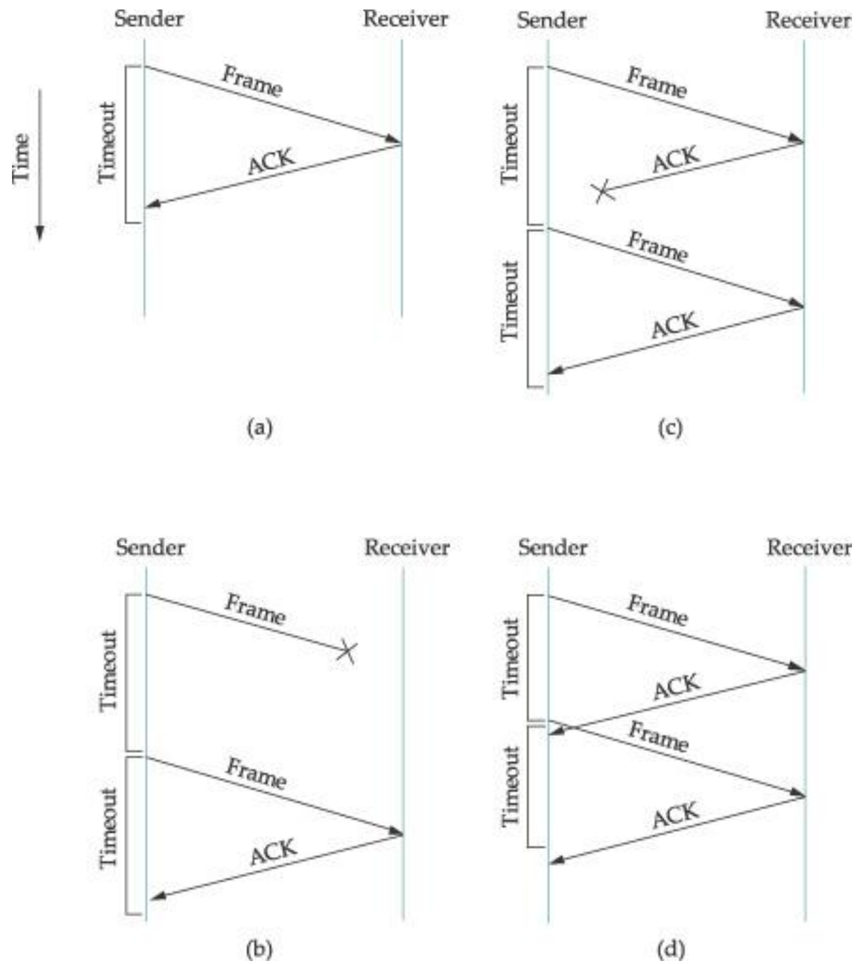


Stop and Wait Protocols

- Simple
- Low Throughput
  - One Frame per RTT

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

Stop and Wait



Duplicate Frames

Stop-and-Wait – Error-free channel

Protocol (p2) ensures sender can't outpace receiver:

– Receiver returns a dummy frame (ack) when ready

ut at ... o-and- ... ntrol

```
void sender2(void)
{
  frame s;
  packet buffer;
  event_type event;

  while (true) {
      from_network_layer(&buffer);
      s.info = buffer;
      to_physical_layer(&s);
      wait_for_event(&event);
  }
}
```

Sender waits to for ack after passing frame to physical layer

```
void receiver2(void)
{
  frame r, s;
  event_type event;
  while (true) {
      wait_for_event(&event);
      from_physical_layer(&r);
      to_network_layer(&r.info);
      to_physical_layer(&s);
  }
}
```

Receiver sends ack after passing frame to network layer

Stop-and-Wait – Noisy channel (1)

ARQ (Automatic Repeat reQuest) adds error control

- Receiver acks frames that are correctly delivered
- Sender sets timer and resends frame if no ack)

For correctness, frames and acks must be numbered

- Else receiver can't tell retransmission (due to lost ack or early timer) from new frame
- For stop-and-wait, 2 numbers (1 bit) are sufficient

Stop-and-Wait – Noisy channel (2)

Sender loop (p3):

Send frame (or retransmission)
Set timer for retransmission
Wait for ack or timeout

If a good ack then set up for the next frame to send (else the old frame will be retransmitted)

```
void sender3(void) {
  seq_nr next_frame_to_send;
  frame s;
  packet buffer;
  event_type event;

  next_frame_to_send = 0;
  from_network_layer(&buffer);
  while (true) {
      s.info = buffer;
      s.seq = next_frame_to_send;
      to_physical_layer(&s);
      start_timer(s.seq);
      wait_for_event(&event);
      if (event == frame_arrival) {
            from_physical_layer(&s);
            if (s.ack == next_frame_to_send) {
                  stop_timer(s.ack);
                  from_network_layer(&buffer);
                  inc(next_frame_to_send);
            }
      }
  }
}
```

Stop-and-Wait – Noisy channel (3)

Receiver loop (p3):

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

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);
      if (event == frame_arrival) {
          from_physical_layer(&r);
          if (r.seq == frame_expected) {
              to_network_layer(&r.info);
              inc(frame_expected);
          }
          s.ack = 1 – frame_expected;
          to_physical_layer(&s);
      }
  }
}
```

Wait for a frame

If it's new then take it and advance expected frame

Ack current frame

Sliding Window Protocols

- – Sliding Window concept »
- – One-bit Sliding Window »
- – Go-Back-N »
- – Selective Repeat »

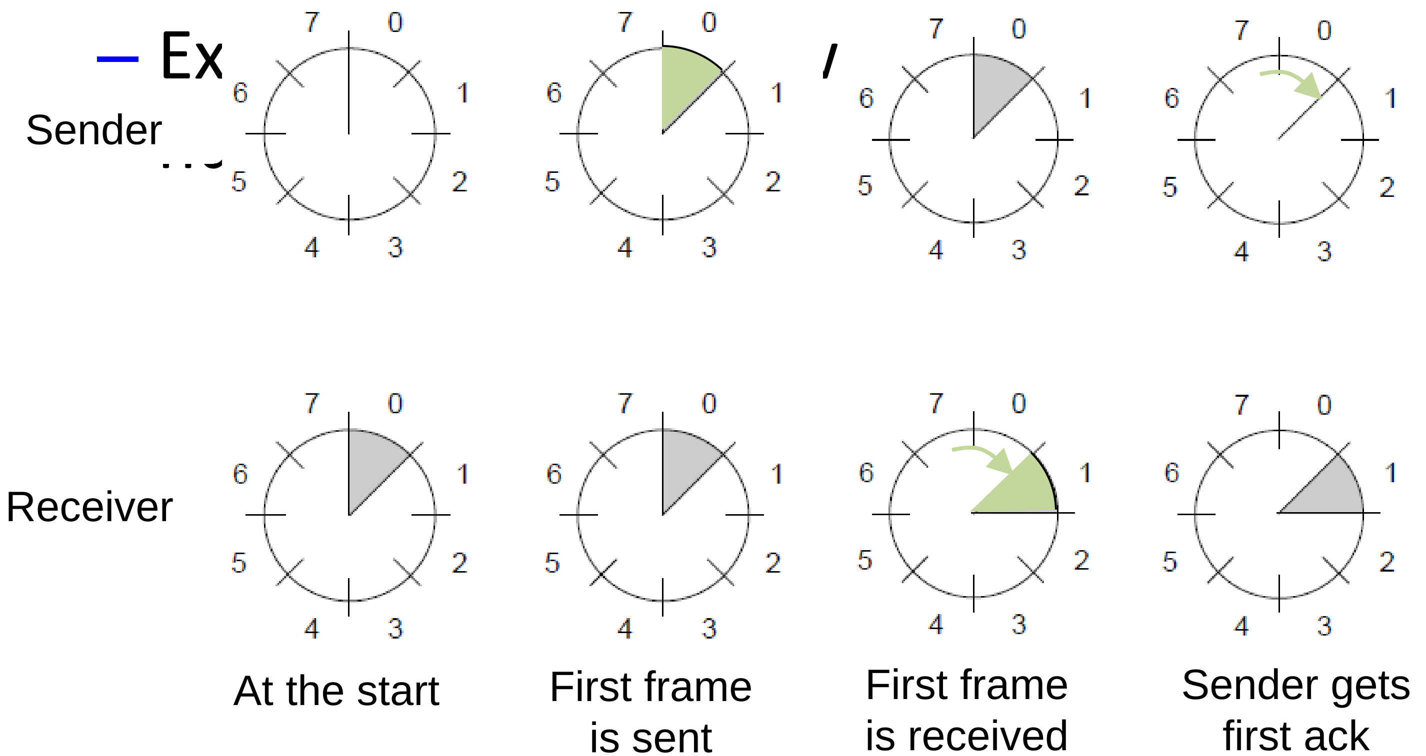
Sliding Window concept (1)

- Sender maintains window of frames it can send
  - Needs to buffer them for possible retransmission
  - Window advances with next acknowledgements
- Receiver maintains window of frames it can receive
  - Needs to keep buffer space for arrivals
  - Window advances with in-order arrivals

Sliding Window concept (2)

A sliding window advancing at the sender and receiver

– Ex



Sliding Window concept (3)

Larger windows enable pipelining for efficient link use

- Stop-and-wait (w=1) is inefficient for long links
- Best window (w) depends on bandwidth-delay (BD)
- Want w ≥ 2BD+1 to ensure high link utilization

Pipelining leads to different choices for errors/buffering

- We will consider Go-Back-N and Selective Repeat

One-Bit Sliding Window (1)

- Transfers data in both directions with stop-and-wait
  - Piggybacks acks on reverse data frames for efficiency
  - Handles transmission errors, flow control, early timers

Each node is sender and receiver (p4):

Prepare first frame

Launch it, and set timer

```
void protocol4 (void) {
  seq_nr next_frame_to_send;
  seq_nr frame_expected;
  frame r, s;
  packet buffer;
  event_type event;
  next_frame_to_send = 0;
  frame_expected = 0;
  from_network_layer(&buffer);
  s.info = buffer;
  s.seq = next_frame_to_send;
  s.ack = 1 – frame_expected;
  to_physical_layer(&s);
  start_timer(s.seq);

. . .
```

One-Bit Sliding Window (2)

. . .

Wait for frame or timeout

If a frame with new data then deliver it

If an ack for last send then prepare for next data frame

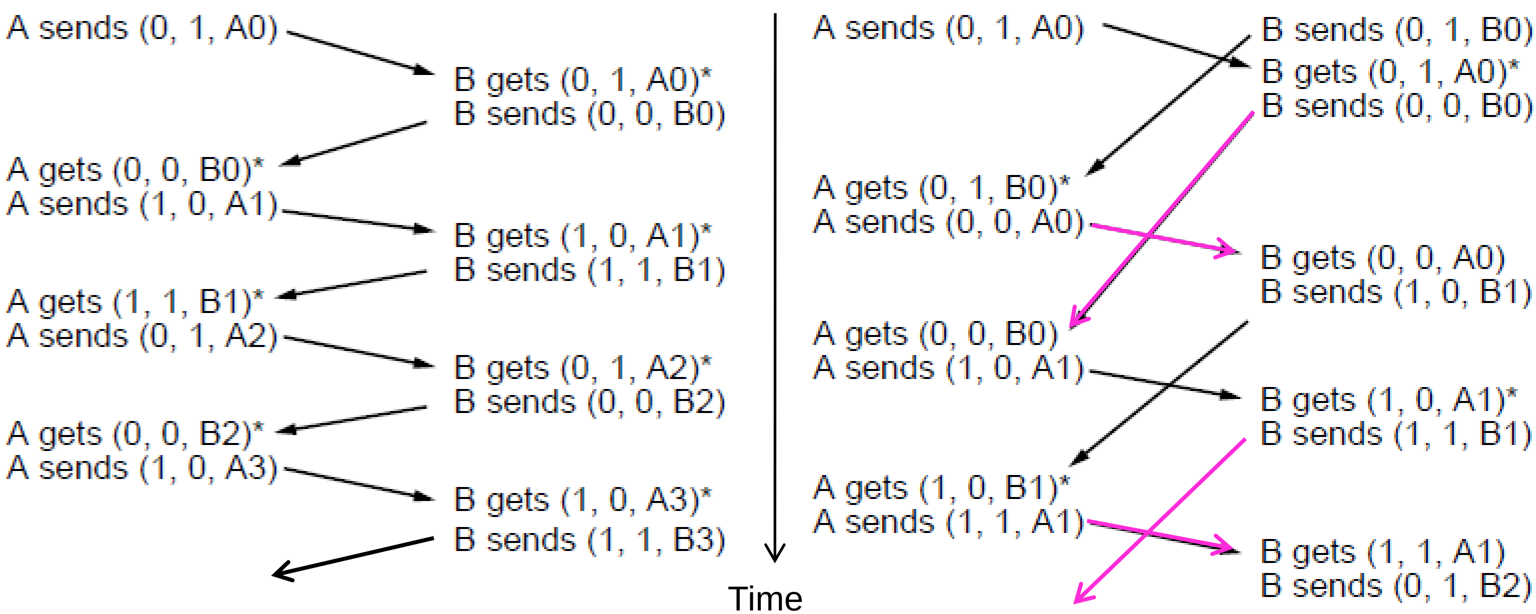
(Otherwise it was a timeout)

Send next data frame or retransmit old one; ack the last data we received

```
while (true) {
    wait_for_event(&event);
    if (event == frame_arrival) {
        from_physical_layer(&r);
        if (r.seq == frame_expected) {
            to_network_layer(&r.info);
            inc(frame_expected);
        }
        if (r.ack == next_frame_to_send) {
            stop_timer(r.ack);
            from_network_layer(&buffer);
            inc(next_frame_to_send);
        }
    }
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 – frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);
  }
}
```

One-Bit Sliding Window (3)

- Two scenarios show subtle interactions exist in p4:
  - Simultaneous start [right] causes correct but slow operation compared to normal [left] due to duplicate transmissions.



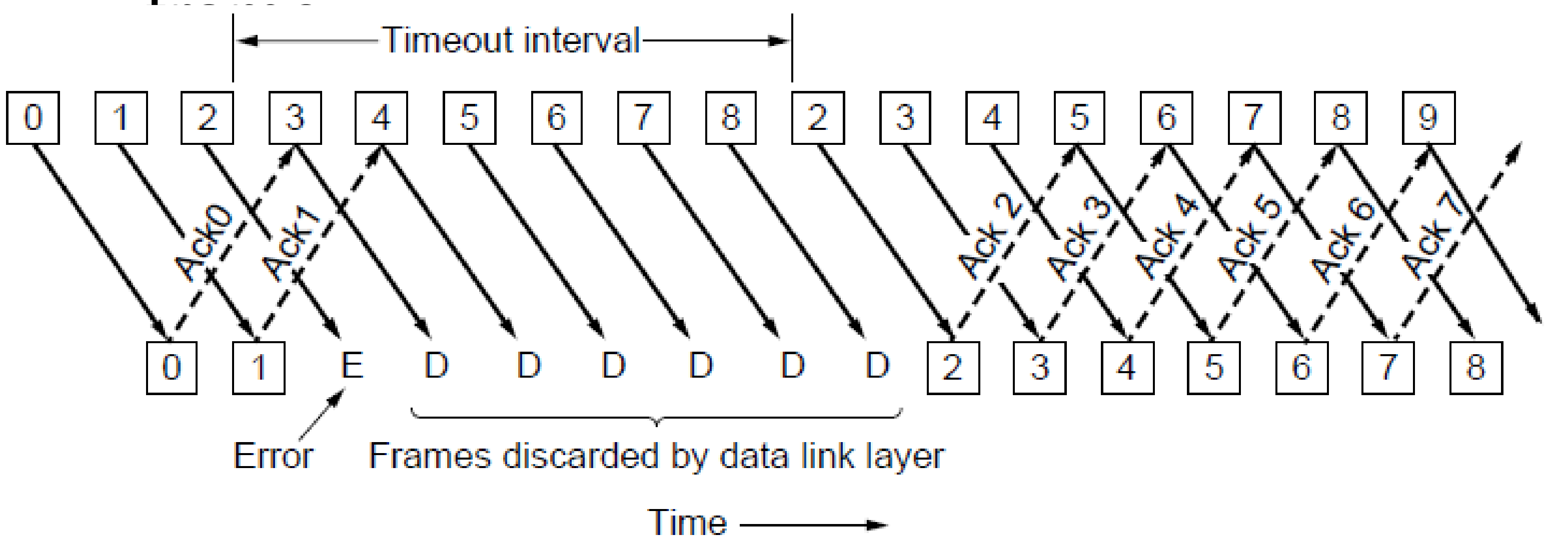
Notation is (seq, ack, frame number). Asterisk indicates frame accepted by network layer .

(a) Normal case (b) Correct, but poor performance

Go-Back-N (1)

Receiver only accepts/acks frames that arrive in order:

– Discards frames that follow a missing/errored



Go-Back-N (2)

Tradeoff made for Go-Back-N:

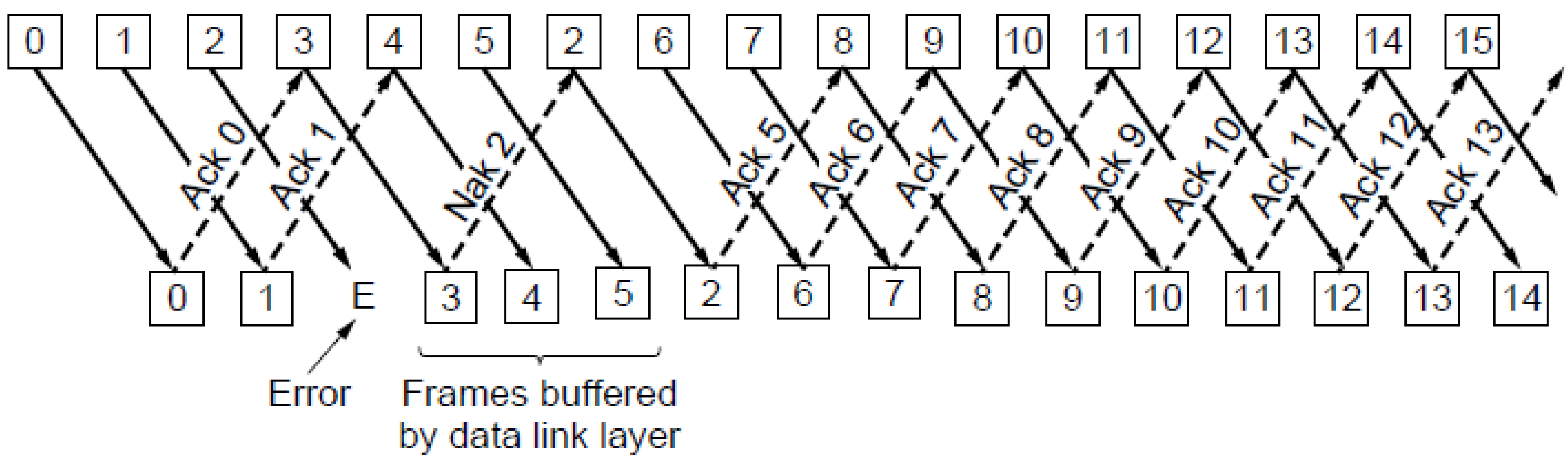
- Simple strategy for receiver; needs only 1 frame
- Wastes link bandwidth for errors with large windows; entire window is retransmitted

Implemented as p5 (see code in book)

Selective Repeat (1)

Receiver accepts frames anywhere in receive window

- Cumulative ack indicates highest in-order frame



Selective Repeat (2)

Tradeoff made for Selective Repeat:

- More complex than Go-Back-N due to buffering at receiver and multiple timers at sender
- More efficient use of link bandwidth as only lost frames are resent (with low error rates)

Implemented as p6 (see code in book)

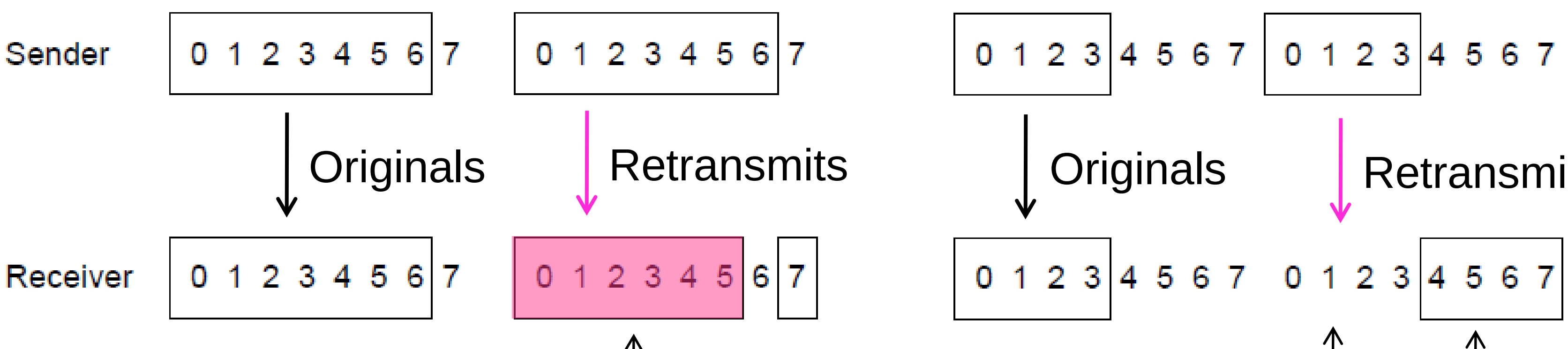
Selective Repeat (3)

For correctness, we require:

– Sequence numbers (s) at least twice the window (w)

Error case (s=8, w=7) – too few sequence numbers

Correct (s=8, w=4) – enough sequence numbers



New receive window overlaps old – retransmits ambiguous

New and old receive window don't overlap – no ambiguity

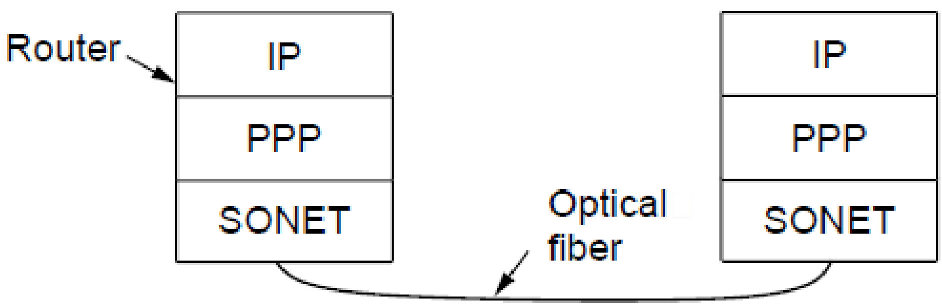
Example Data Link Protocols

- Packet over SONET »
- PPP (Point-to-Point Protocol) »
- ADSL (Asymmetric Digital Subscriber Loop) »

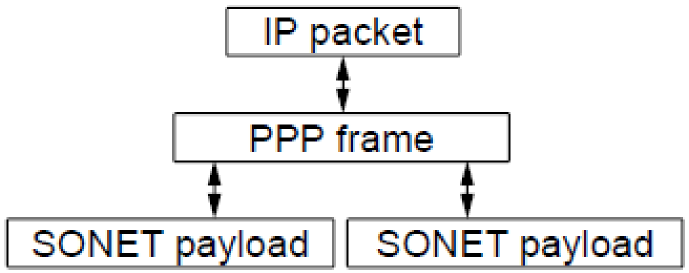
Packet over SONET

Packet over SONET is the method used to carry IP packets over SONET optical fiber links

- Uses PPP (Point-to-Point Protocol) for framing



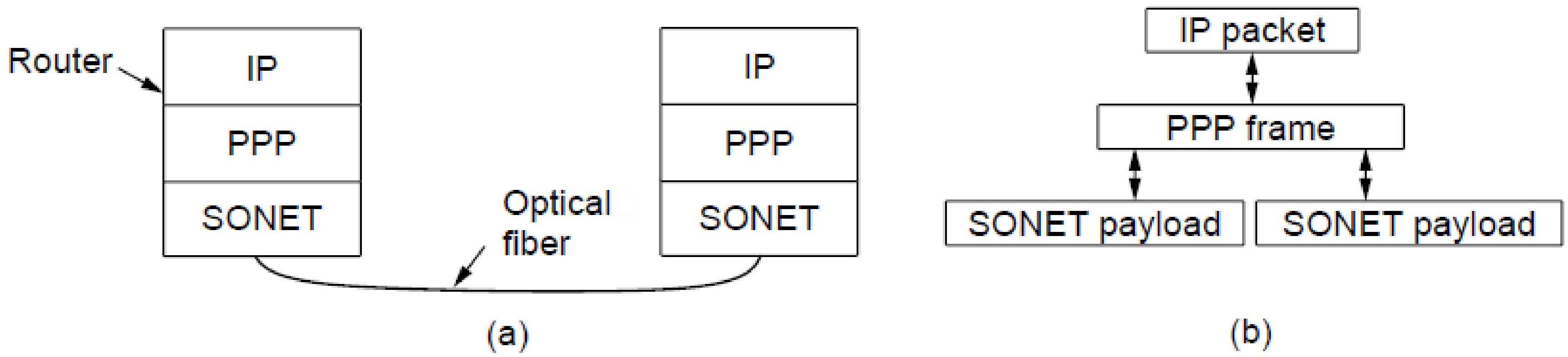
Protocol stacks



PPP frames may be split over SONET payloads

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

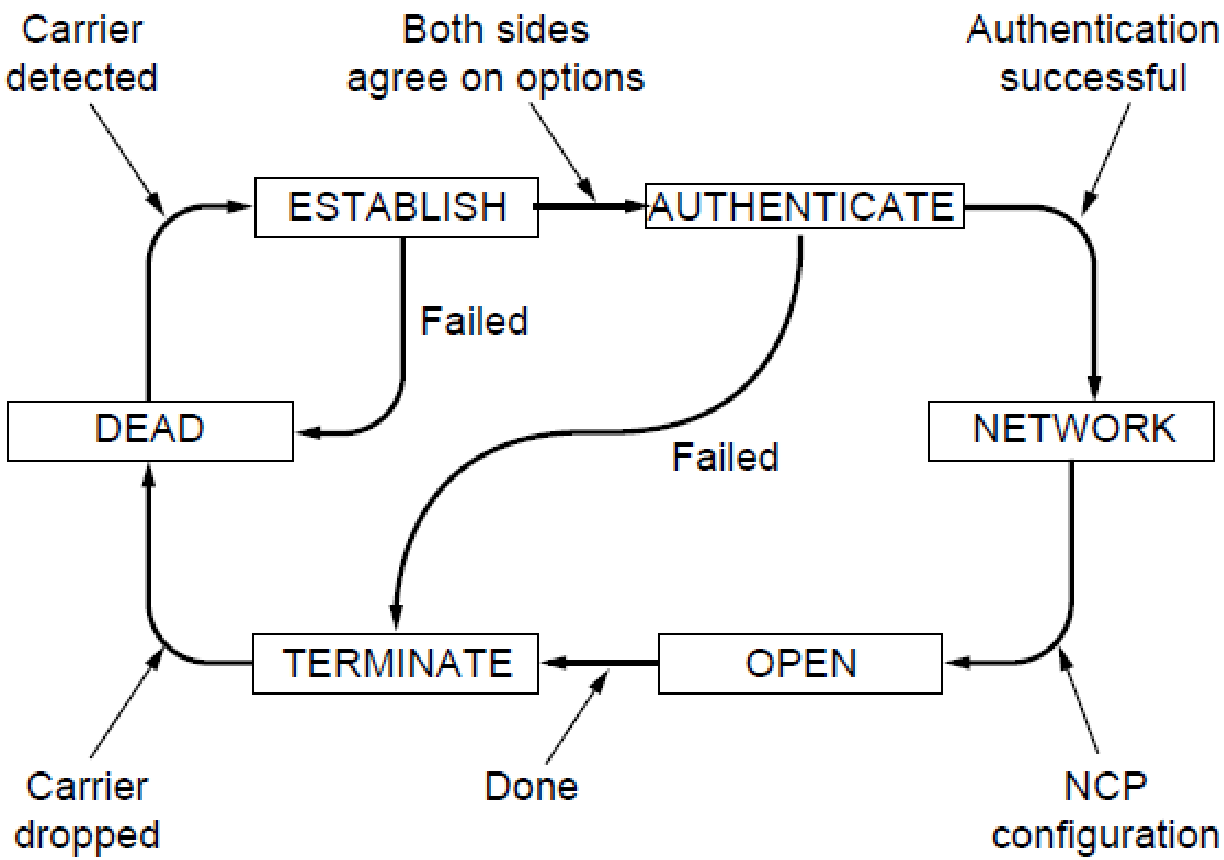
Packet over SONET (3)

The PPP full frame format for unnumbered mode operation

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

Packet over SONET (4)

State diagram for bringing a PPP link up and down



PPP (1)

PPP (Point-to-Point Protocol) is a general method for delivering packets across links

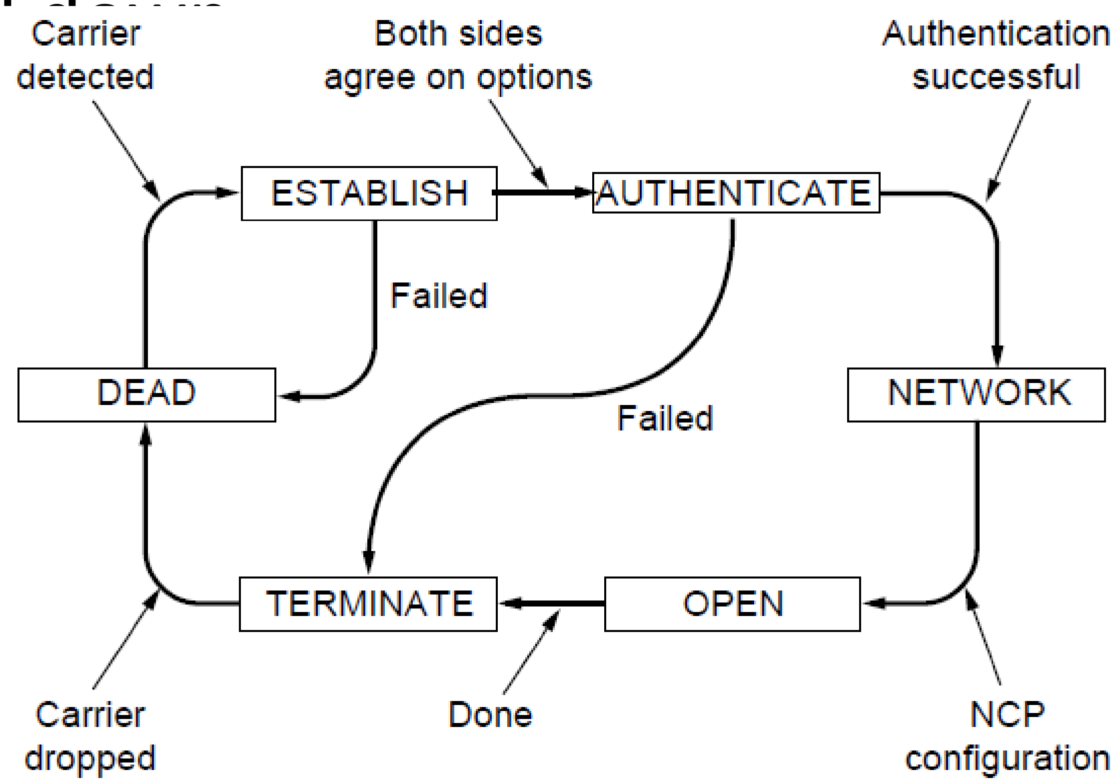
- Framing uses a flag (0x7E) and byte stuffing
- "Unnumbered mode" (connectionless unacknow-ledged service) is used to carry IP packets

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

0x21 for IPv4 IP packet

PPP (2)

A link control protocol brings the PPP link up and down



State machine for link control

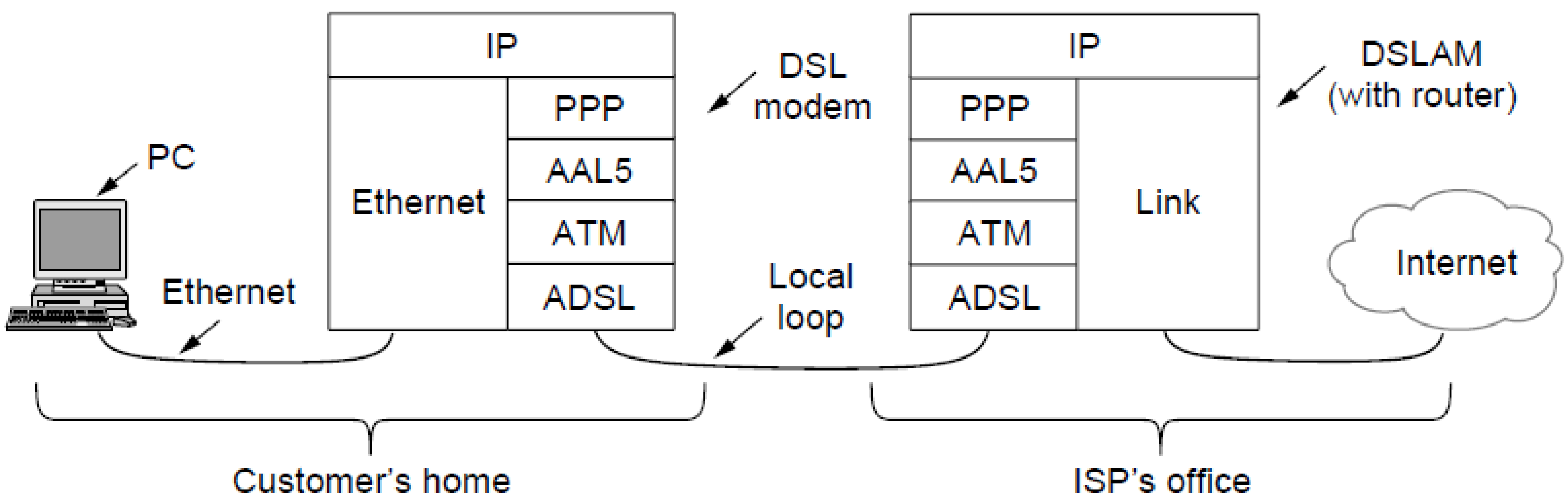
PPP – Point to Point Protocol (3)

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)

ADSL (1)

Widely used for broadband Internet over local loops

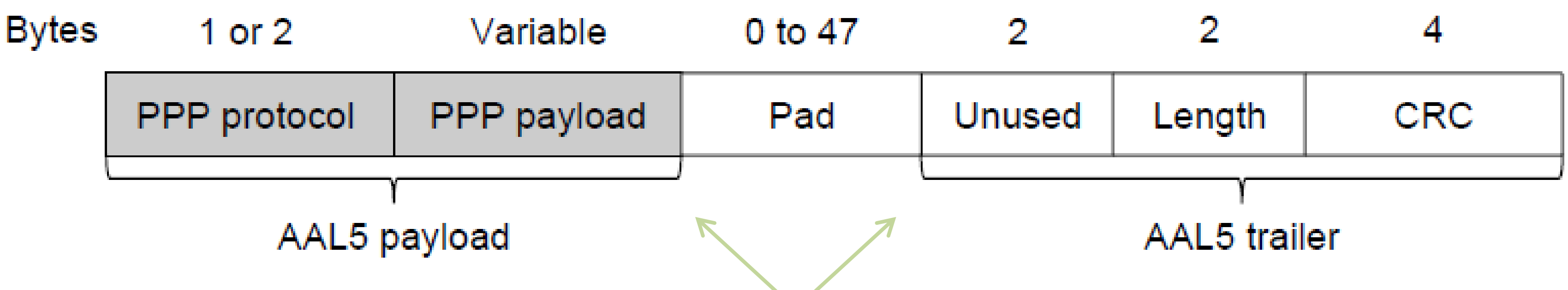
- ADSL runs from modem (customer) to DSLAM



ADSL (2)

PPP data is sent in AAL5 frames over ATM cells:

– ATM is a link layer that uses short, fixed-size cells (53 bytes); each cell has a virtual circuit identifier



AAL5 frame is divided into 48 byte pieces, each of which goes into one ATM cell with 5 header bytes

High-Level Data Link Control

Frame format for bit-oriented protocols.

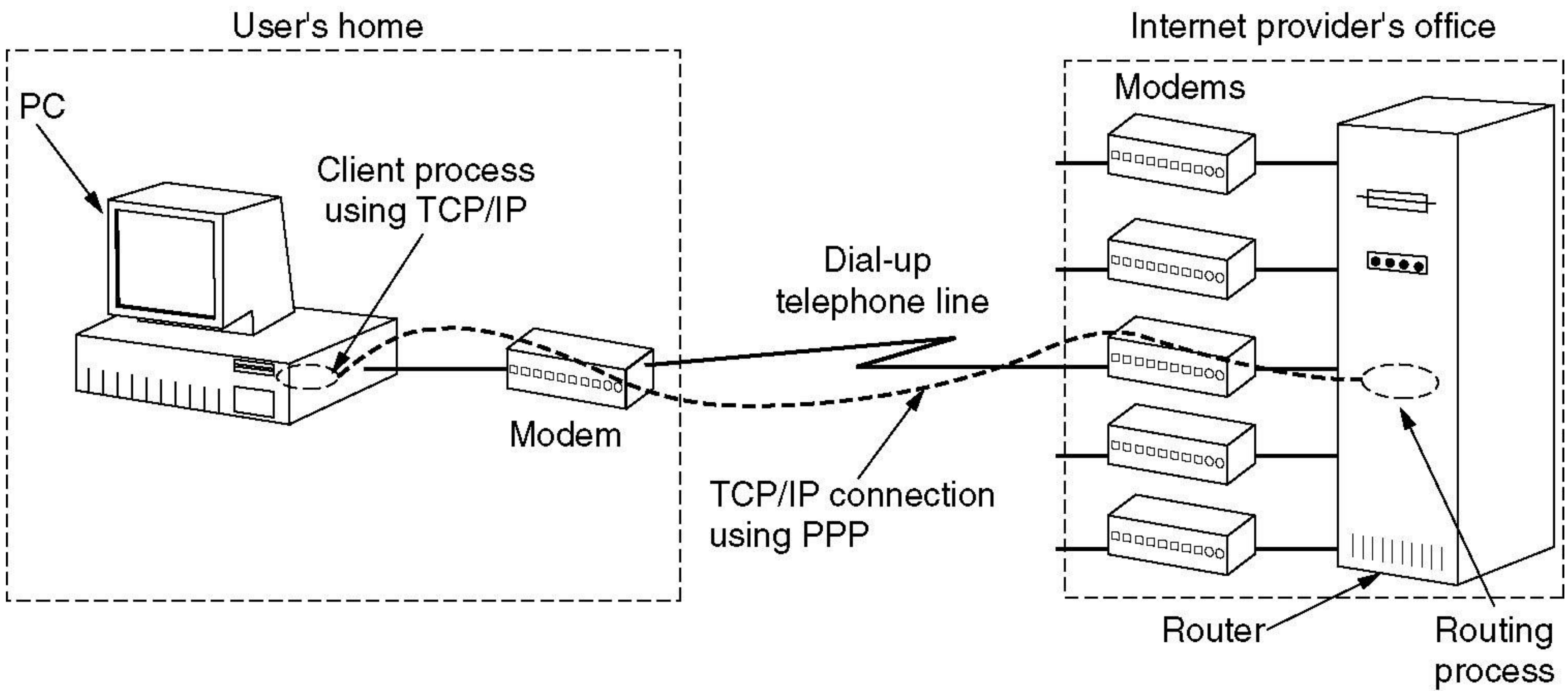
Bits	8	8	8	≥ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

High-Level Data Link Control (2)

Bits	1	3		1	3
(a)	0	Seq		P/F	Next
(b)	1	0	Type	P/F	Next
(c)	1	1	Type	P/F	Modifier

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

The Data Link Layer in the Internet



End

Chapter 3

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

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

A positive acknowledgement with retransmission protocol.

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.

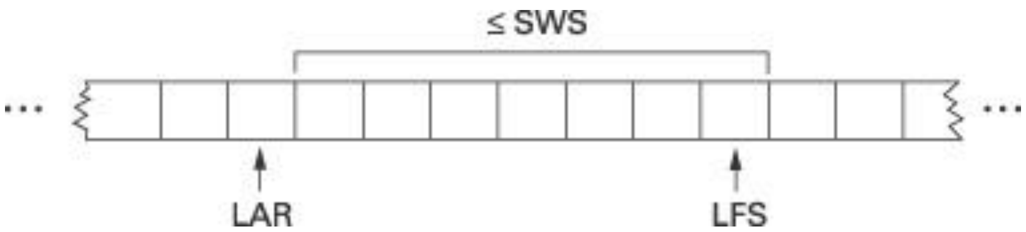
Sliding Window Protocol

- 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

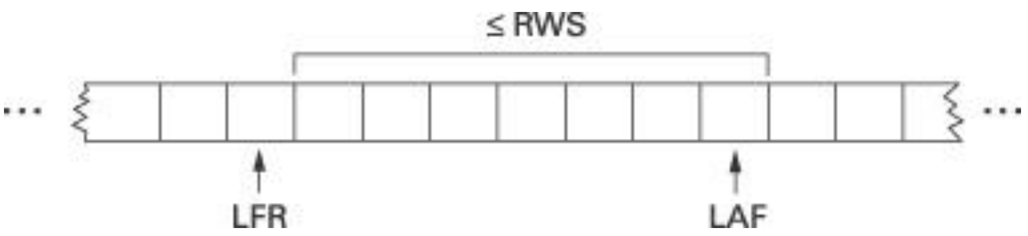
Sliding Window Protocol

- 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.

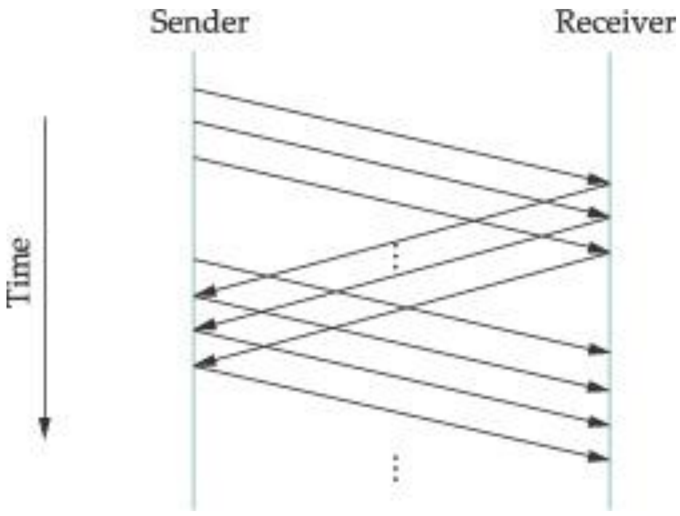
Sliding Window



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

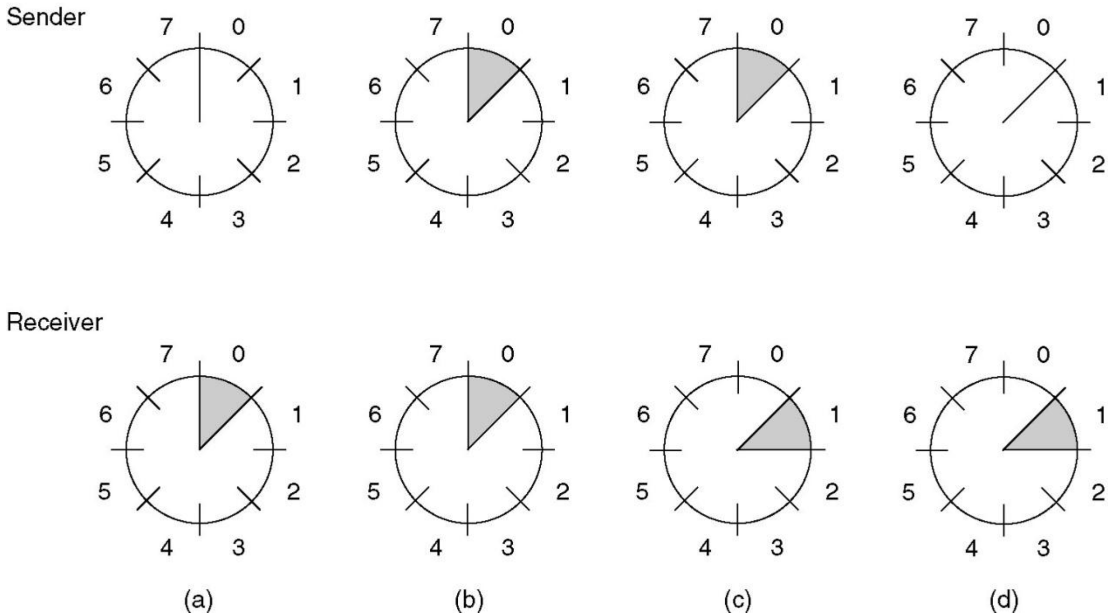
1. Reliably deliver frame across unreliable links
2. Deliver frames to higher levels in sequence
3. Flow Control

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

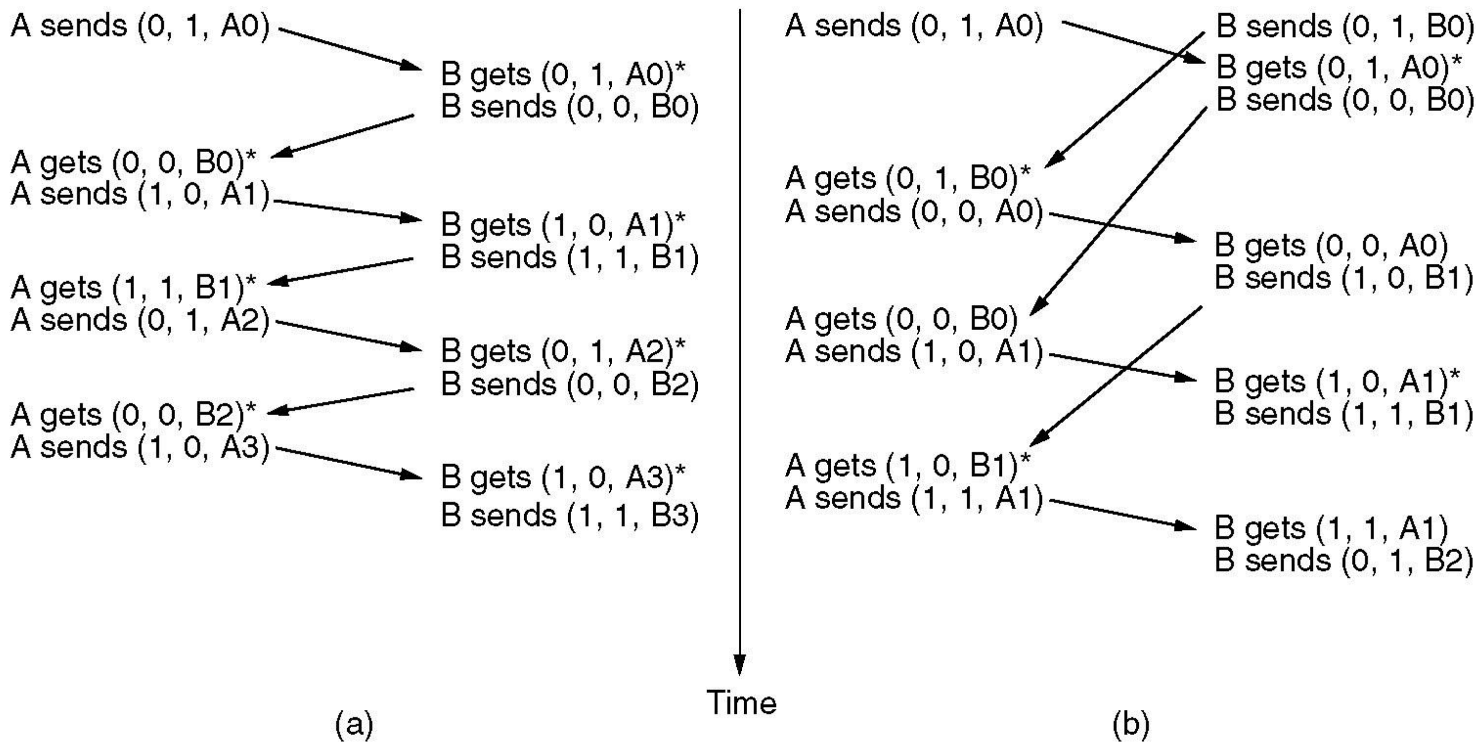
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /*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

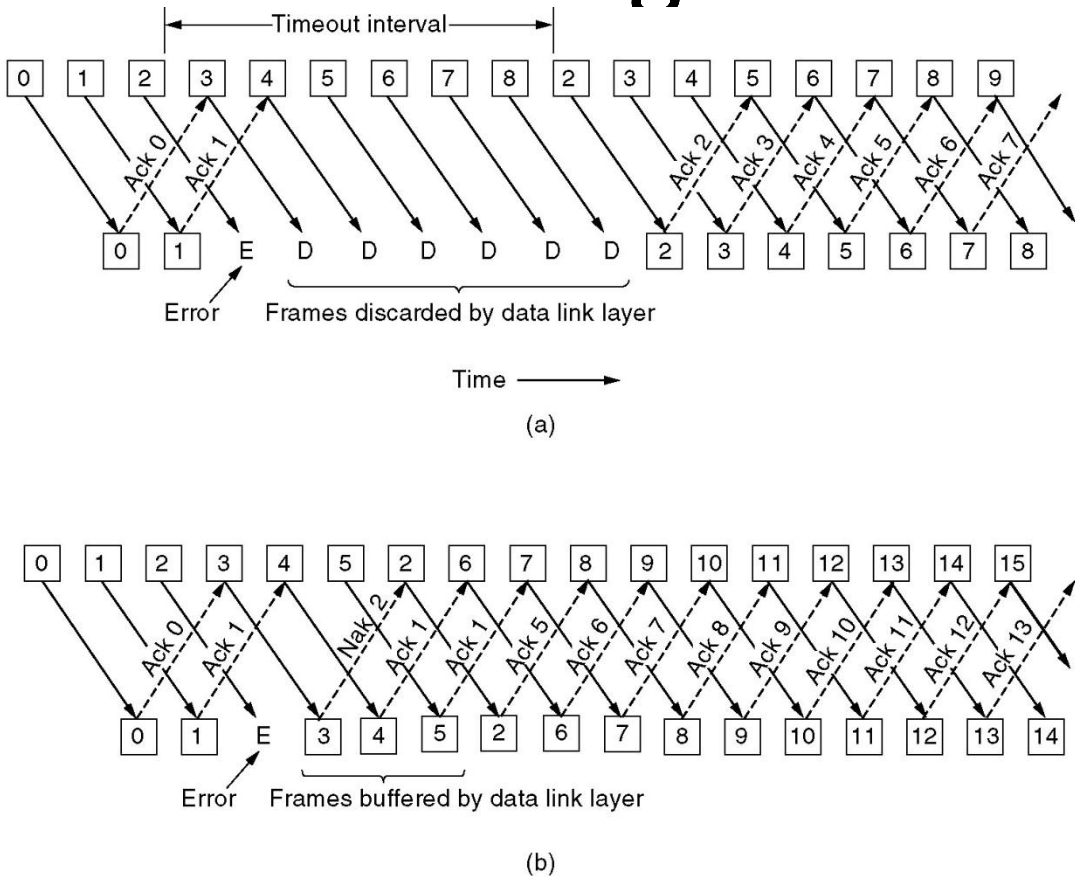
```
while (true) {
    wait_for_event(&event);                    /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {              /* a frame has arrived undamaged. */
        from_physical_layer(&r);               /* go get it */

        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

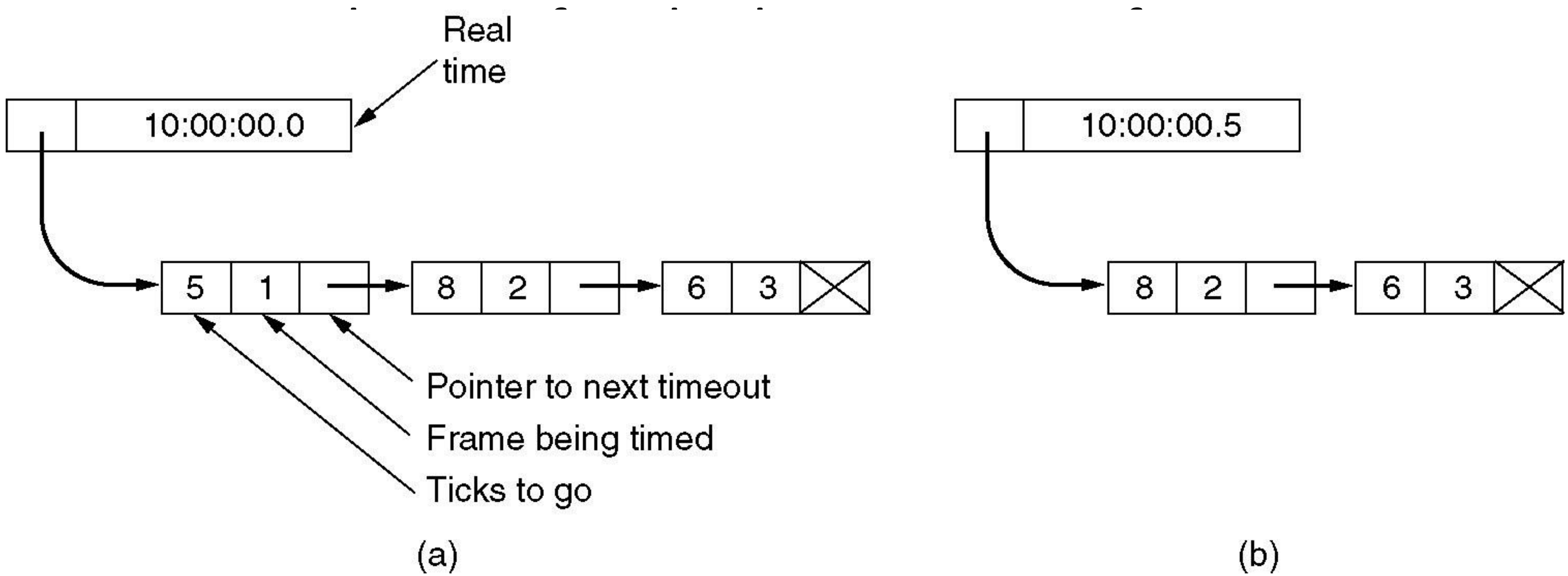
```
while (true) {
  wait_for_event(&event);              /* four possibilities: see event_type above */

  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.

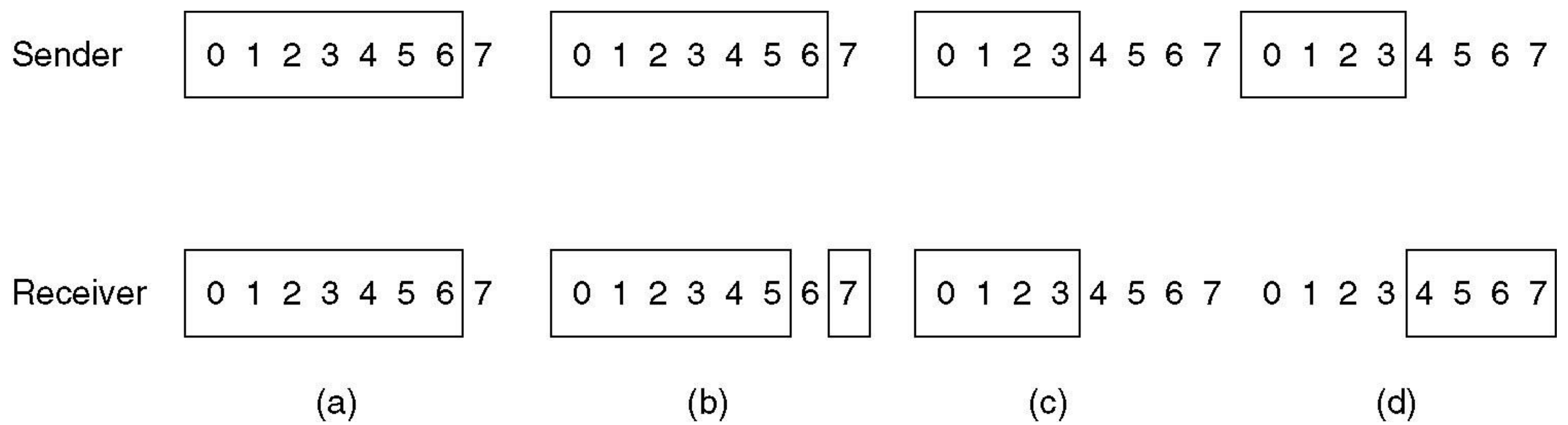
```
    if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
         send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

    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.