# CSCI 460
# Networks and Communications

## Datalink Layer

**Humayun Kabir**
Professor, CS, Vancouver Island University, BC, Canada

# Outline

- Connectionless services
- Framing
- Error Control
- Error-Correcting Code
- Error-Detecting Code
- Flow Control
  - Stop and Wait Protocol
  - Sliding Window Protocol with Go Back N
  - Sliding Window Protocol with Selective Repeat

# Connectionless Services

**Unacknowledged** connectionless service

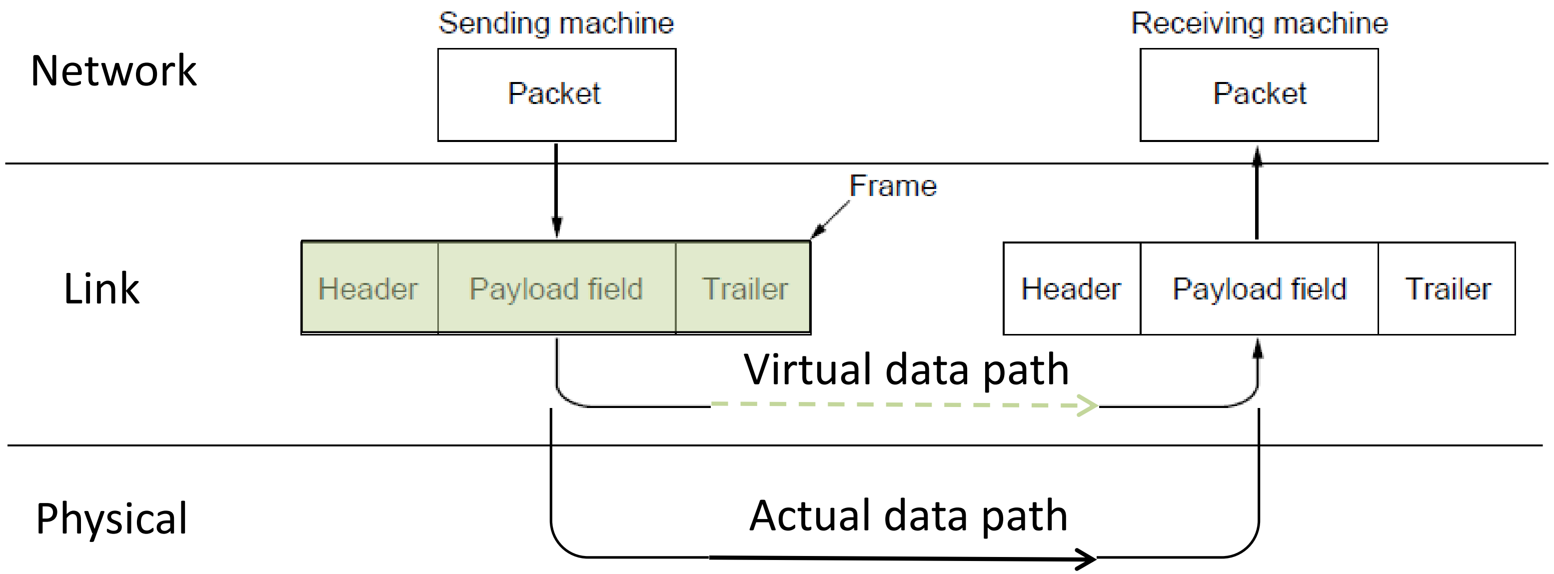
- Frame is sent with no connection and error recovery
- Ethernet is example

**Acknowledged** connectionless service

- Frame is sent with no connection but with retransmissions if needed
- Example is 802.11

# 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; reception is the opposite process



# Framing Methods

– Frames need to be delimited from each other by the sender before transmitting over the physical medium in order to facilitate the receiver to separate them upon receptions.

– Once the frames are separated from each other these delimiters have no other usage at the receiver.

– Frame delimiters are not part of data link layer protocol header.

# Framing Methods

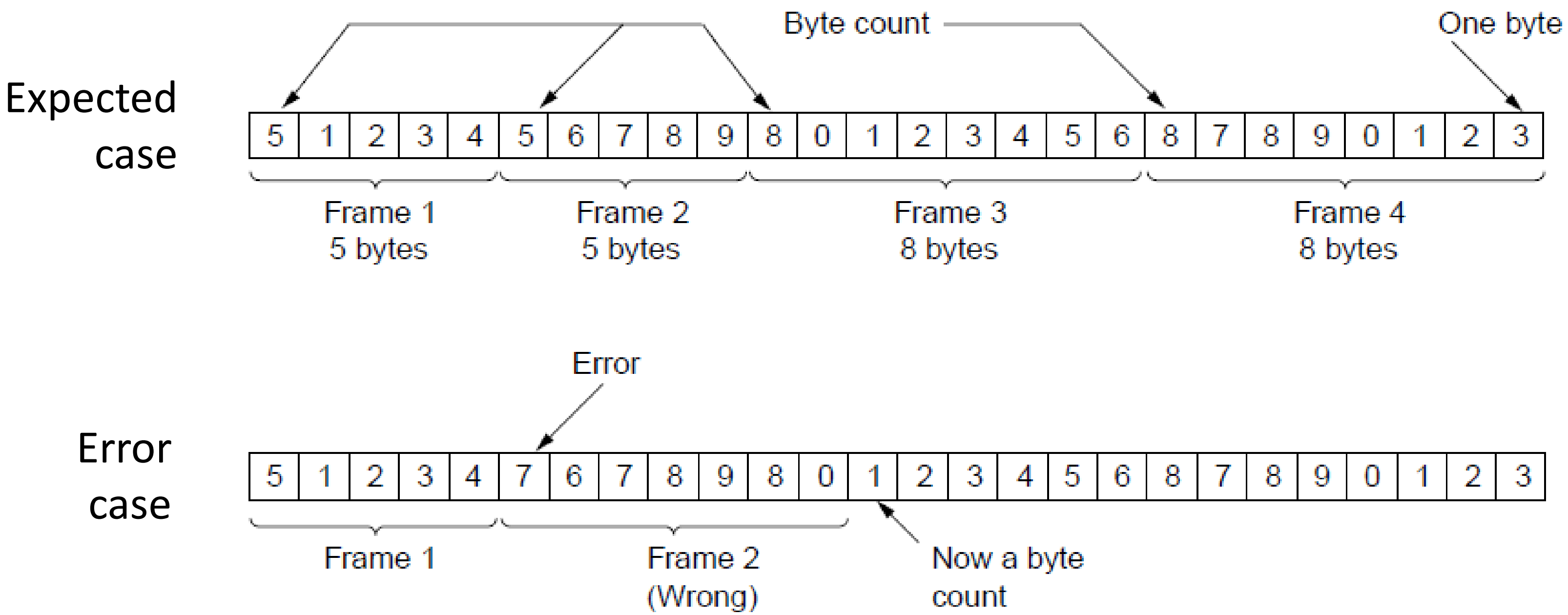
Frame delimiting methods

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

# Framing – Byte count

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

- Simple, but difficult to resynchronize after an error

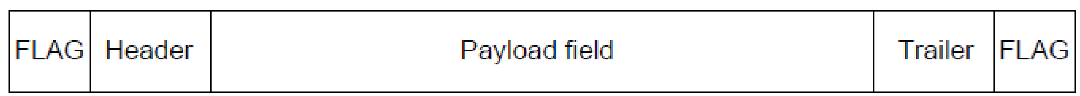


# Framing – Byte stuffing

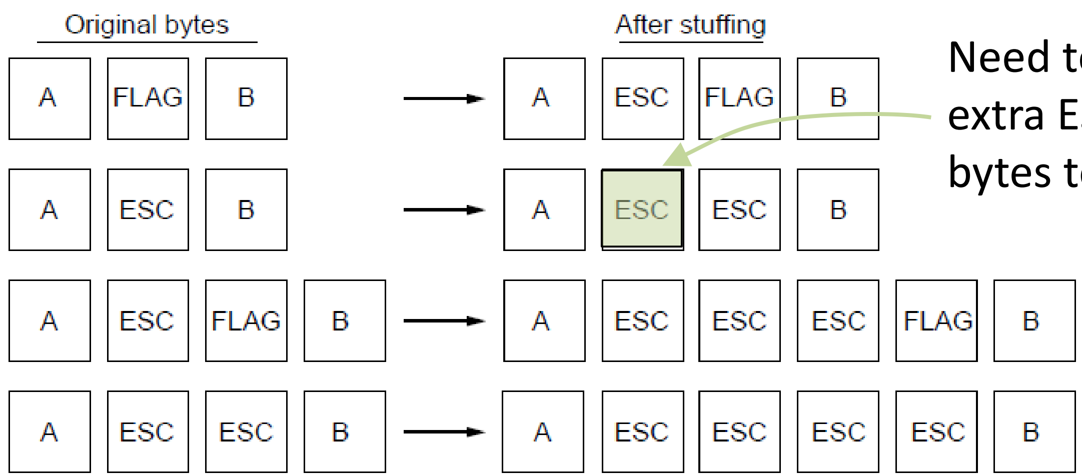
Special flag bytes delimit frames; occurrences of flags in the data must be stuffed (escaped)

- Longer, but easy to resynchronize after error

Frame format




Stuffing examples




Need to escape extra ESCAPE bytes too!

# 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 receive, a 0 after five 1s is deleted

| | |
|---|---|
| Data bits | 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 |
| Transmitted bits with stuffing | 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0 |

Stuffed bits

# Error Control

Error control repairs frames that are received in error

- Requires errors to be detected at the receiver
- Requires to acknowledge error free frames
- Typically retransmits the unacknowledged frames
- Timer protects against lost acknowledgements

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

# Error Detection and Correction

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 Bounds – Hamming distance

Code turns data of n bits into codewords of n+k bits

Hamming distance 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

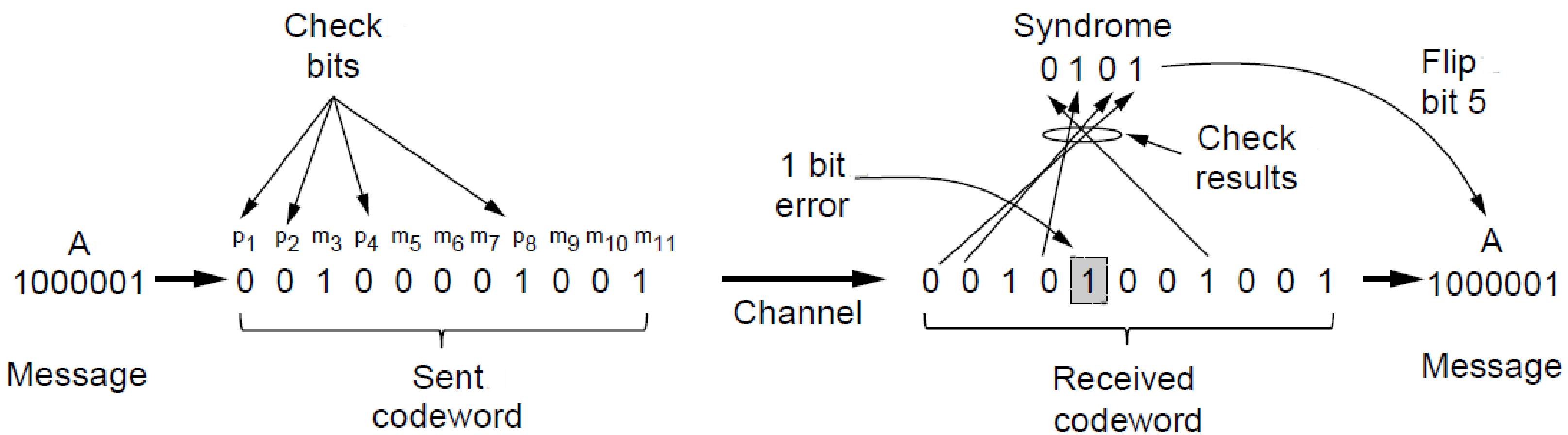
Bounds for a code with distance:

- 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 codeword
- Recomputing the parity sums (syndrome) gives the position of the error to flip, or 0 if there is no error




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

# Error Detection – Parity

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)

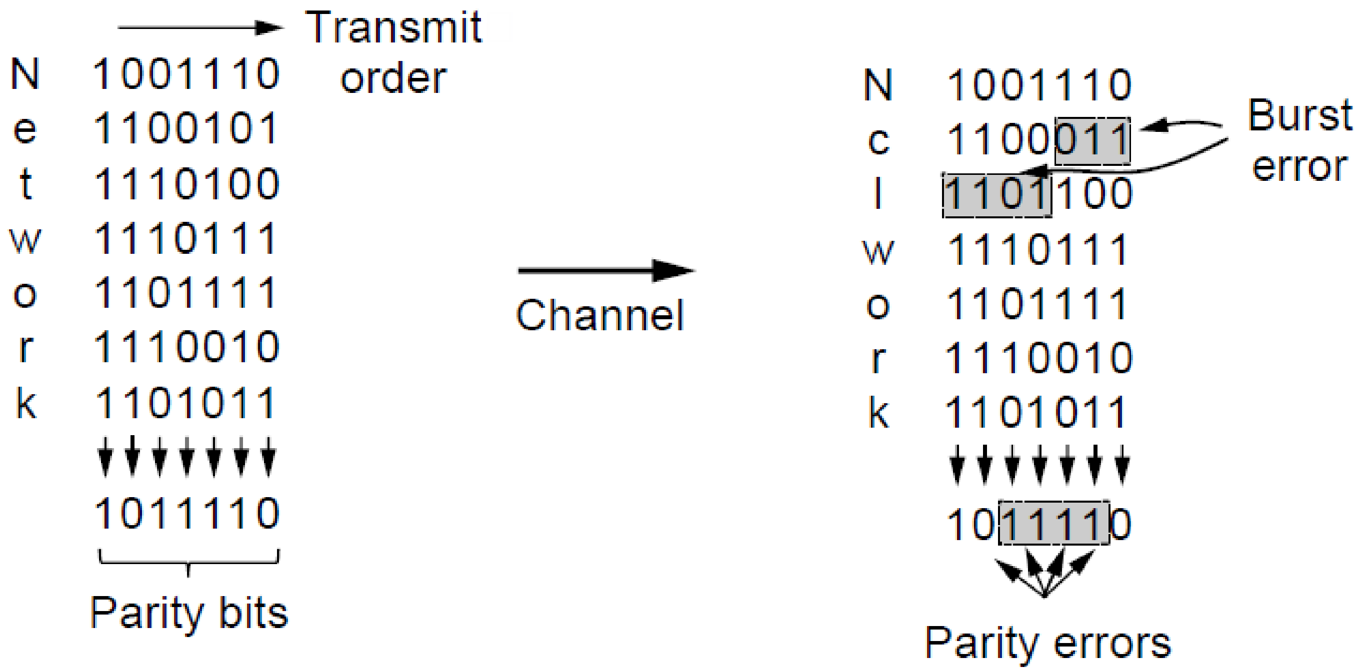
Simple way to detect an *odd* number of errors

- Ex: 1 error, 11100101; detected, sum is wrong
- Ex: 3 errors, 11011001; detected sum is wrong
- Ex: 2 errors, 11101101; *not detected*, sum is right!
- Error can also be in the parity bit itself
- Random errors are detected with probability ½

# Error Detection – Parity

<u>Interleaving</u> of N parity bits detects burst errors up to N

- Each parity sum is made over non-adjacent bits
- An even burst of up to N errors will not cause it to fail



# 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

# Error Detection – CRCs

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

Start by adding 0s to frame and try dividing

Offset by any reminder to make it evenly divisible

# Error Detection – CRCs

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

# 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

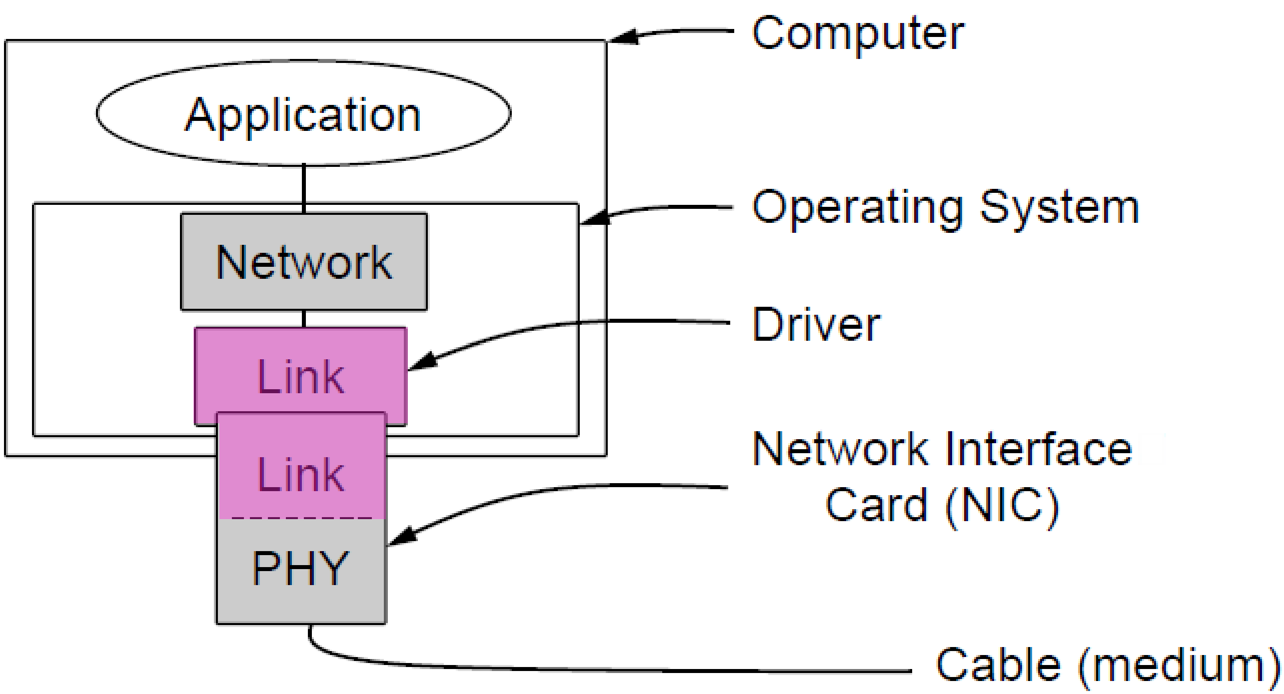
Flow control is a topic in both Link and Transport layers.

# 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

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



# Link layer environment

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

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

# 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 …

# Stop-and-Wait – Error-free channel

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

- Receiver returns a dummy frame (ack) when ready
- Only one frame out at a time – called stop-and-wait
- We added flow control!

```
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

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

## 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

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 (event == frame_arrival) {`

If it's new then take it and advance expected frame → `if (r.seq == frame_expected) { to_network_layer(&r.info); inc(frame_expected); }`

Ack current frame → `s.ack = 1 – frame_expected;`

# Sliding Window Protocols

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

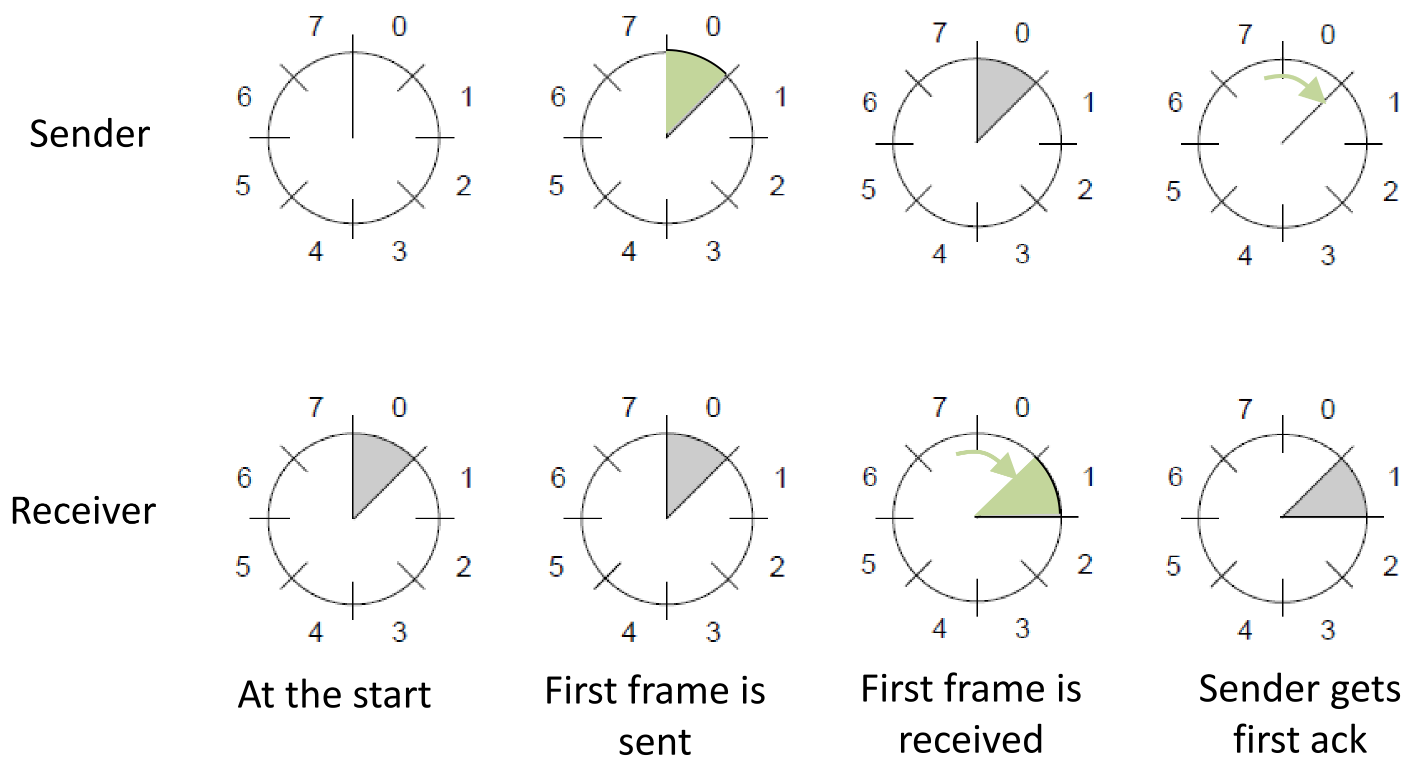
# Sliding Window concept

- 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

A sliding window advancing at the sender and receiver

- Ex: window size is 1, with a 3-bit sequence number.



# Sliding Window concept

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 $w \geq 2BD+1$ 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

- 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):

```
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);      // Prepare first frame
  s.info = buffer;
  s.seq = next_frame_to_send;
  s.ack = 1 – frame_expected;
  to_physical_layer(&s);            // Launch it, and set timer
  start_timer(s.seq);
```

. . .

Prepare first frame

Launch it, and set timer

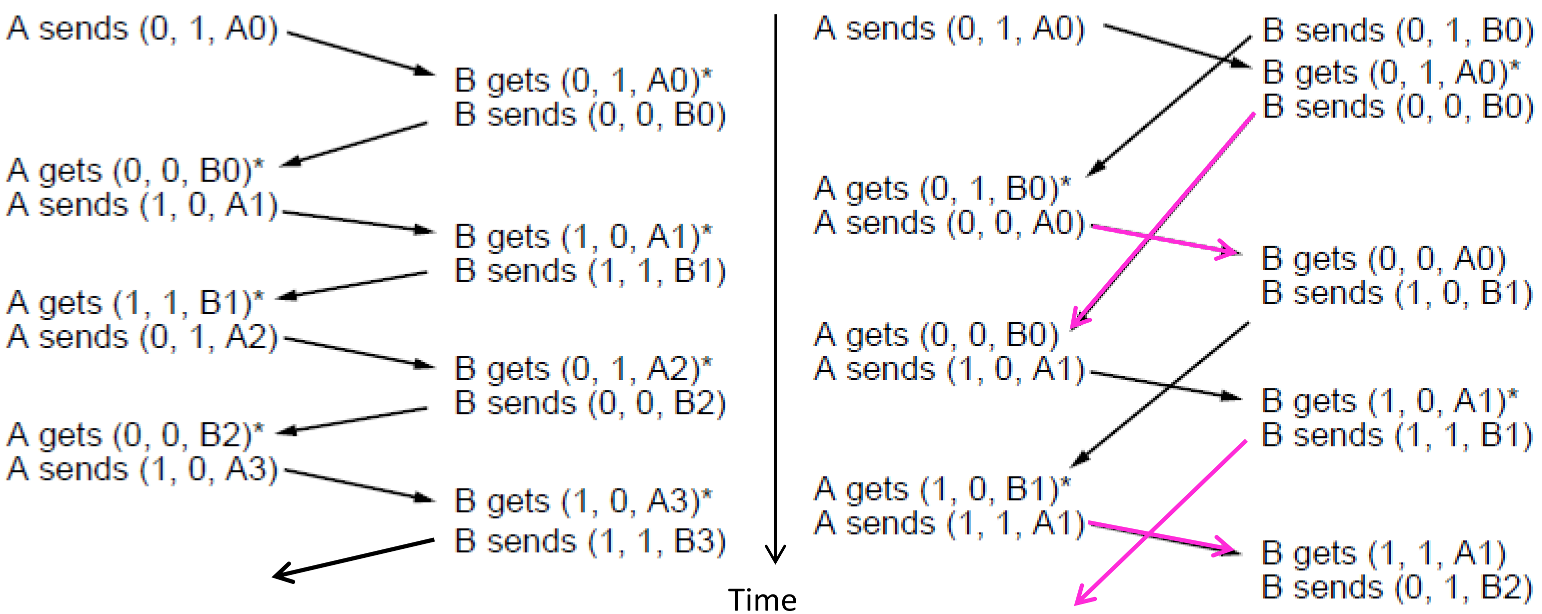
# One-Bit Sliding Window

| | |
|---|---|
| | . . . |
| Wait for frame or timeout | `wait_for_event(&event);` |
| If a frame with new data then deliver it | `if (r.seq == frame_expected) { ... }` |
| If an ack for last send then prepare for next data frame | `if (r.ack == next_frame_to_send) { ... }` |
| (Otherwise it was a timeout) | |
| Send next data frame or retransmit old one; ack the last data we received | `to_physical_layer(&s); start_timer(s.seq);` |

```
. . .
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

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



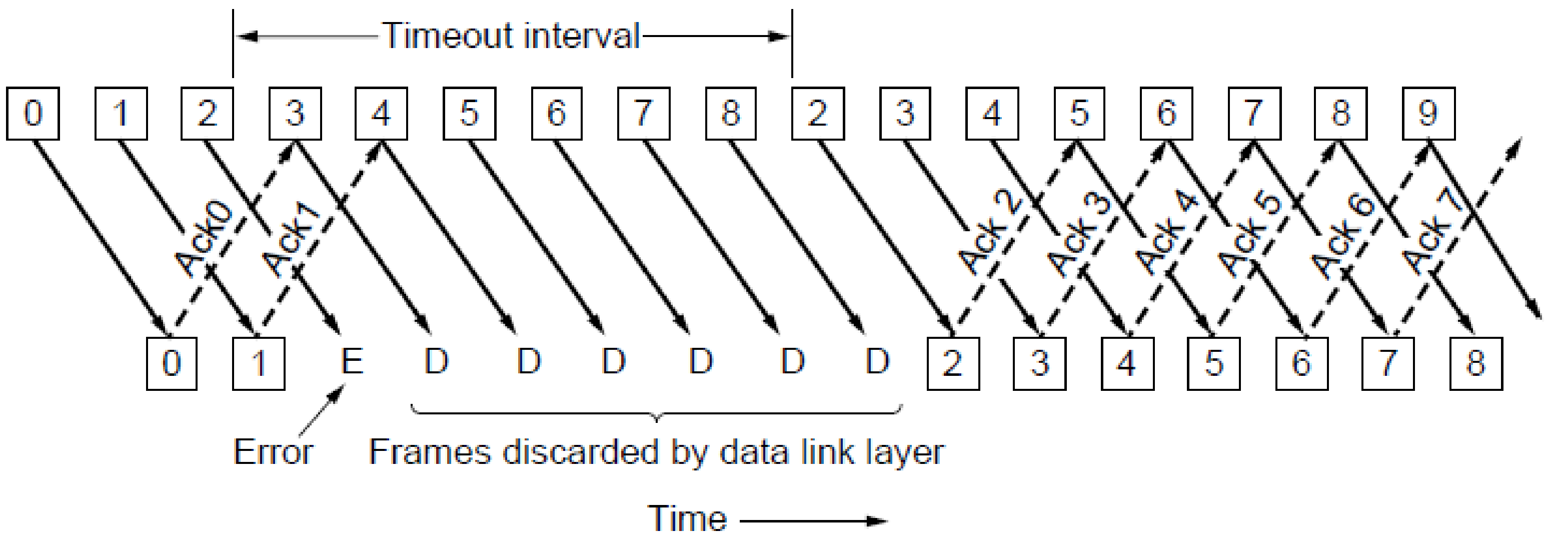

Notation is (seq, ack, frame number). Asterisk indicates frame accepted by network layer .

Normal case | Correct, but poor performance

# Go-Back-N

Receiver only accepts/acks frames that arrive in order:

- Discards frames that follow a missing/errored frame
- Sender times out and resends all outstanding frames



# Go-Back-N

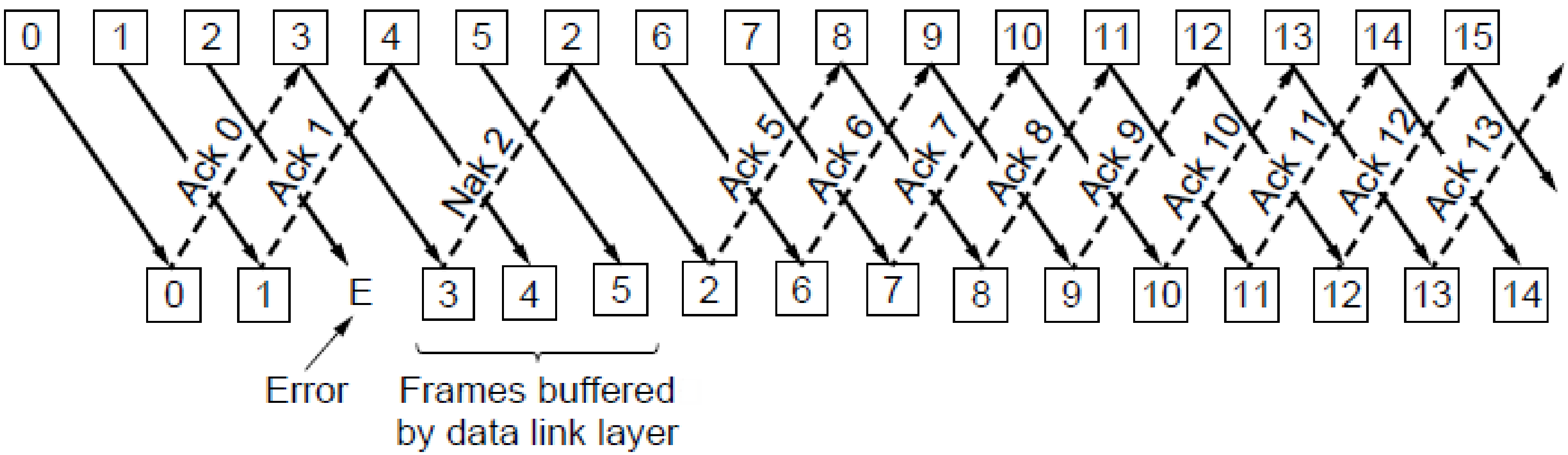
- Tradeoff made for Go-Back-N:
  - Simple strategy for receiver; needs only 1 frame
  - Wastes link bandwidth for errors with large windows; entire window is retransmitted

- Implemented as p5 (see code in book)

# Selective Repeat

Receiver accepts frames anywhere in receive window

- Cumulative ack indicates highest in-order frame
- NAK (negative ack) causes sender retransmission of a missing frame before a timeout resends window



# Selective Repeat

- 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)

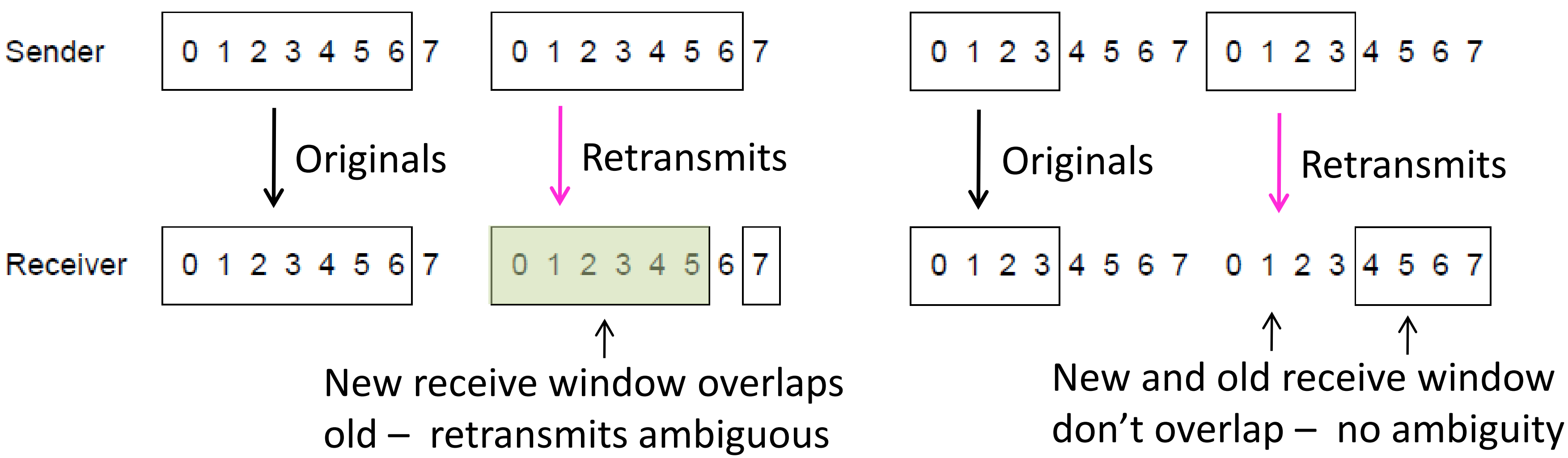
# Selective Repeat

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



# Summary

- Connectionless services
- Framing
- Error Control
- Error-Correcting Code
- Error-Detecting Code
- Flow Control
- Data Link Layer Protocols
  - Stop and Wait Protocol
  - Sliding Window Protocol with Go Back N
  - Sliding Window Protocol with Selective Repeat

# Next

## Medium Access Control Sublayer

- Channel Allocation Problem
- Multiple Access Protocols
    - Pure and Slotted ALOHA
    - Carrier Sense Multiple Access (CSMA)
    - CSMA with Collision Detection (CSMA/CD)
    - Binary Exponential Backoff Algorithm
    - CSMA with Collision Avoidance (CSMA/CA)
- Ethernet and WiFi
- Repeaters, Hubs, Bridges, and Switches