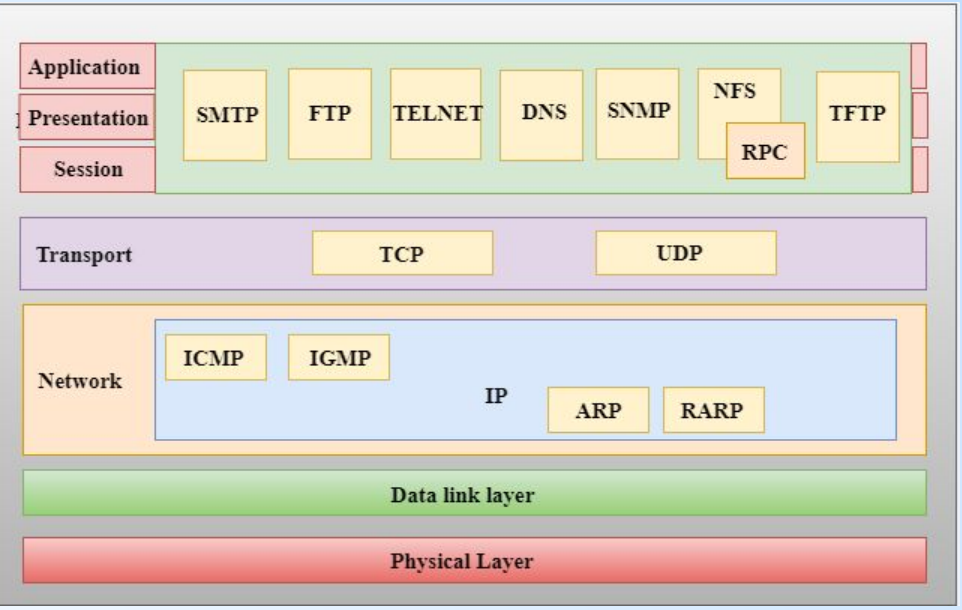
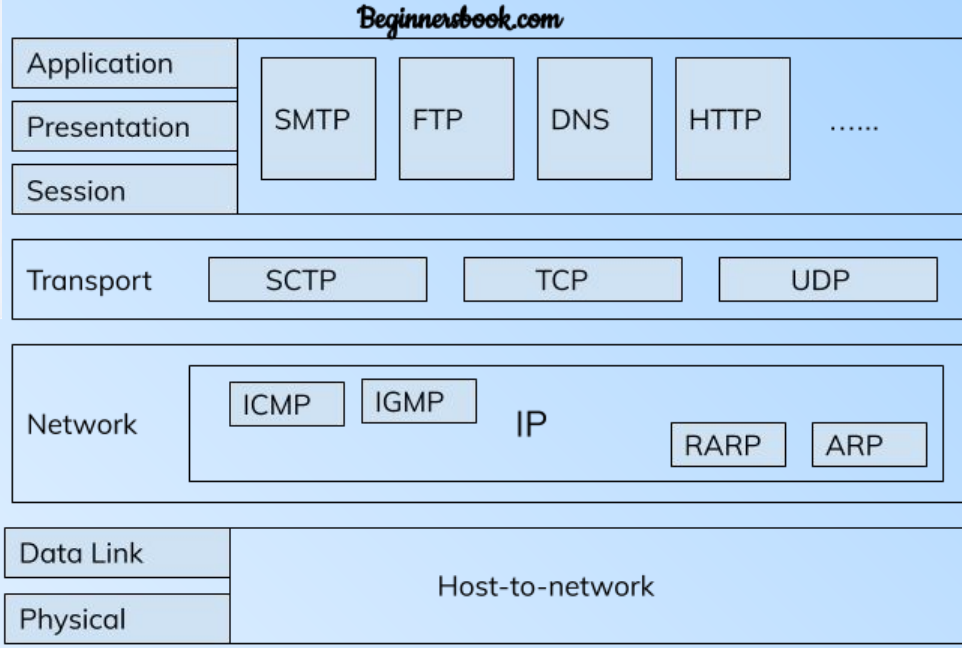
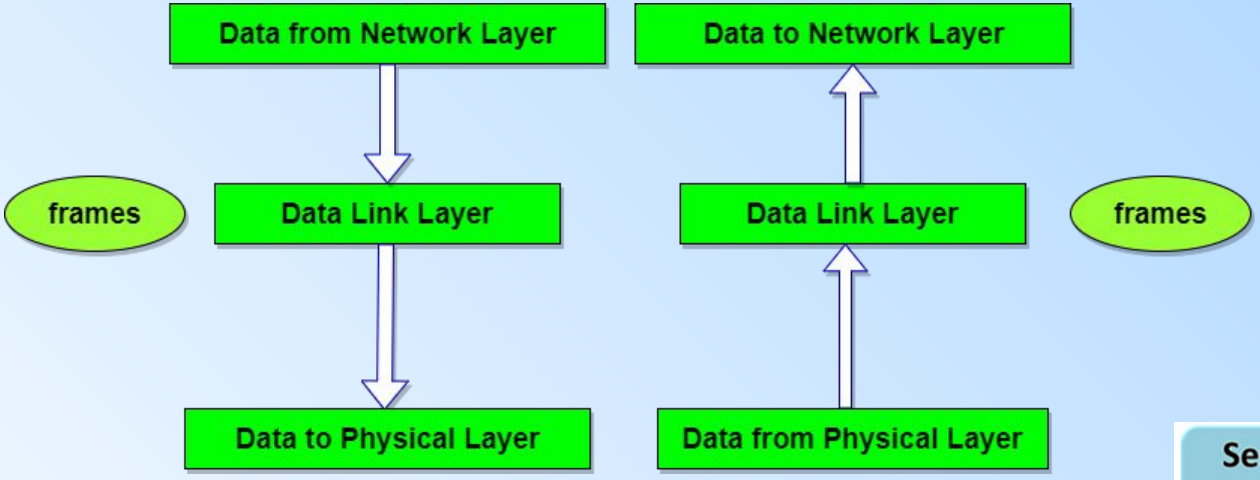
Computer Networks

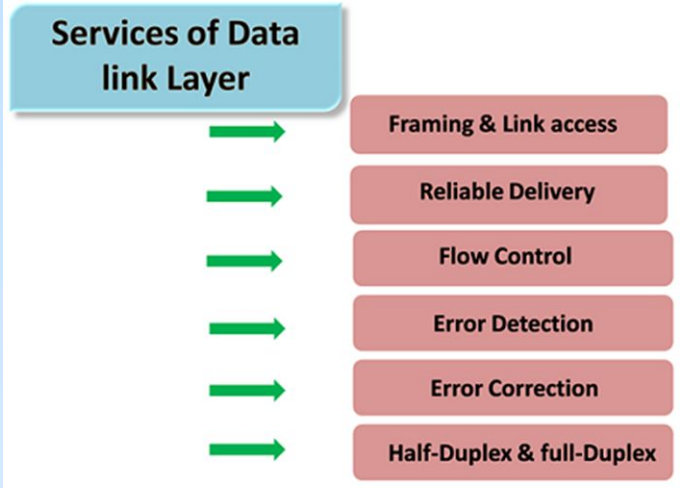
Data link layer

The Internet Layers



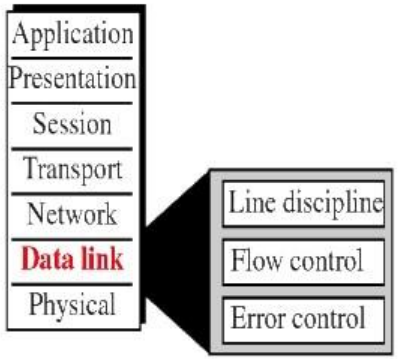



Data from Network Layer
Data to Network Layer
frames
Data Link Layer
Data Link Layer
frames
Data to Physical Layer
Data from Physical Layer


Services of Data link Layer
Framing & Link access
Reliable Delivery
Flow Control
Error Detection
Error Correction
Half-Duplex & full-Duplex

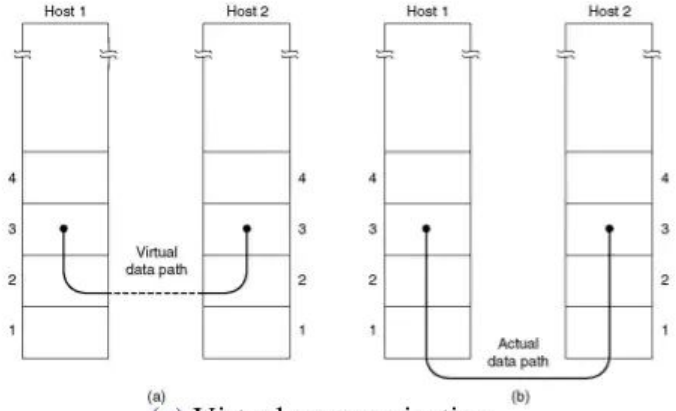
Data Link Layer Design Issues

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



Services Provided to Network Layer

The principal service is transferring data from the network layer on the source machine to the network layer on the destination machine.



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

Services Provided to Network Layer

- Unacknowledged connectionless service.
  - source machine send independent frames to the destination machine without having the destination machine acknowledge them.
- Acknowledged connectionless service.
  - When this service is offered, there are still no logical connections used, but each frame sent is individually acknowledged. In this way, the sender knows whether a frame has arrived correctly.
- Acknowledged connection-oriented service.

Services Provided to Network Layer

- WAN subnet consisting of routers connected by point-to-point leased telephone lines.
- When a frame arrives at a router, the hardware checks it for errors
- Then passes the frame to the data link layer software.
- The data link layer software checks to see if this is the frame expected,
- Gives the packet contained in the payload field to the routing software.
- The routing software then chooses the appropriate outgoing line
- Passes the packet back down to the data link layer software, which then transmits it

Design Issues in Data Link Layer

- **Data-link layer** is the second layer after the physical layer. The data link layer is responsible for maintaining the data link between two hosts or nodes.
- Before going through the design issues in the data link layer. Some of its sub-layers and their functions are as following below.
- The data link layer is divided into two sub-layers :

1. **Logical Link Control Sub-layer (LLC) –**
   Provides the logic for the data link, Thus it controls the synchronization, flow control, and error checking functions of the data link layer. Functions are –
   1. **(i)** Error Recovery.
   2. **(ii)** It performs the flow control operations.
   3. **(iii)** User addressing.

Media Access Control Sub-layer (MAC)

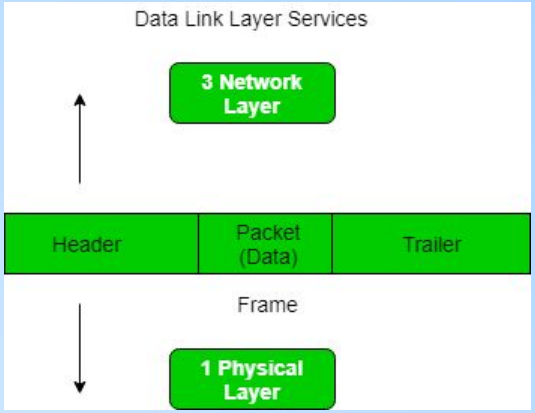
It is the second sub-layer of data-link layer. It controls the flow and multiplexing for transmission medium. Transmission of data packets is controlled by this layer. This layer is responsible for sending the data over the network interface card.

Functions are –

1. **(i)** To perform the control of access to media.
2. **(ii)** It performs the unique addressing to stations directly connected to LAN.
3. **(iii)** Detection of errors.

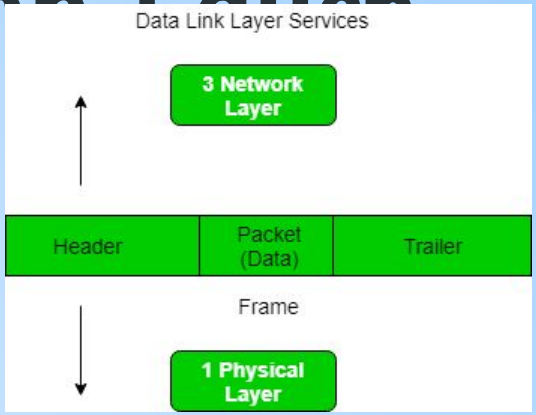
Design issues with data link layer

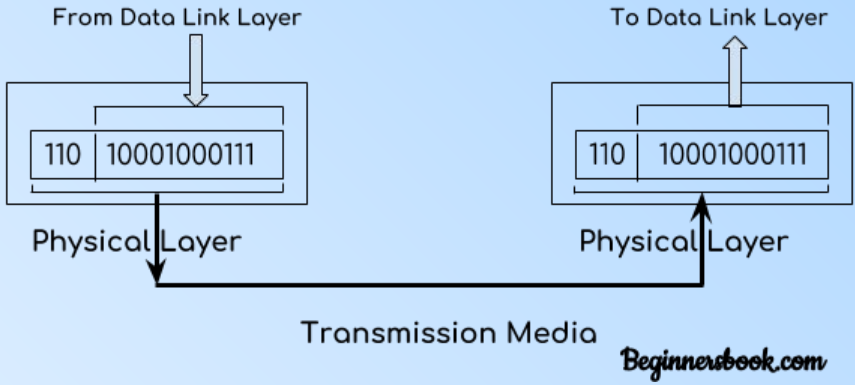
1. **Services provided to the network layer –**
   The data link layer act as a service interface to the network layer. The principle service is transferring data from network layer on sending machine to the network layer on destination machine. This transfer also takes place via DLL (Dynamic Link Library).
2. **Frame synchronization –**
   The source machine sends data in the form of blocks called frames to the destination machine. The starting and ending of each frame should be identified so that the frame can be recognized by the destination machine.
3. **Flow control –**
   Flow control is done to prevent the flow of data frame at the receiver end. The source machine must not send data frames at a rate faster than the capacity of destination machine to accept them.
4. **Error control –**
   Error control is done to prevent duplication of frames. The errors introduced during transmission from source to destination machines must be detected and corrected at the destination machine.



Framing in Data Link Layer

- Frames are the units of digital transmission, particularly in computer networks and telecommunications.
- Framing is a point-to-point connection between two computers or devices consists of a wire in which data is transmitted as a stream of bits.
- However, these bits must be framed into discernible blocks of information.
- Framing is a function of the data link layer. It provides a way for a sender to transmit a set of bits that are meaningful to the receiver.





Framing

- Data link layer must use the service provided to it by the physical layer.
- This bit stream is not guaranteed to be error free.
- Data link layer to break the bit stream up into discrete frames and compute the checksum for each frame.
- When a frame arrives at the destination, the checksum is recomputed.
- If the newly-computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it

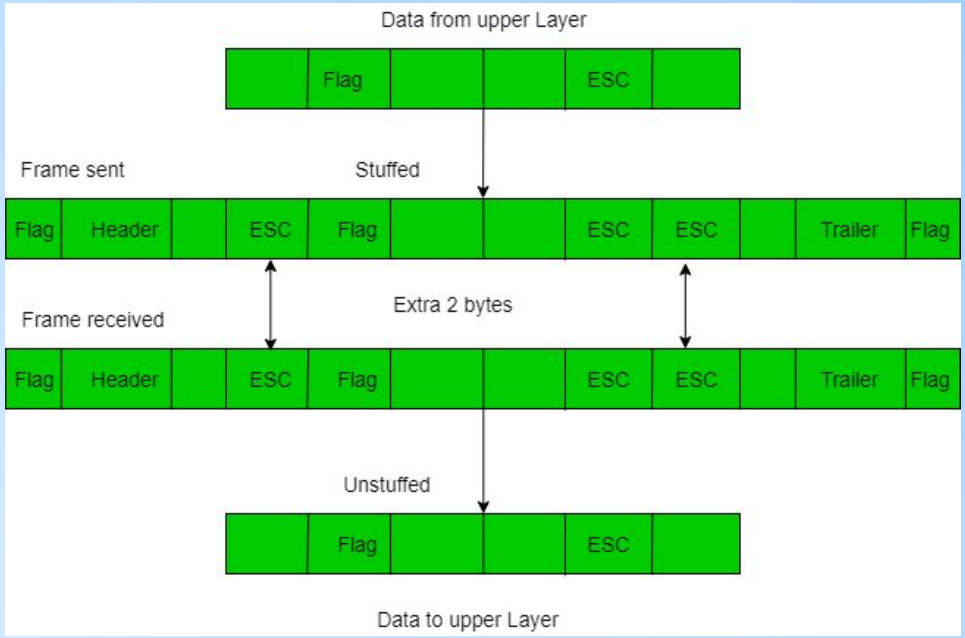
Problems in Framing

- **Detecting start of the frame:** When a frame is transmitted, every station must be able to detect it. Station detects frames by looking out for a special sequence of bits that marks the beginning of the frame i.e. SFD (Starting Frame Delimiter).
- **How does the station detect a frame:** Every station listens to link for SFD pattern through a sequential circuit. If SFD is detected, sequential circuit alerts station. Station checks destination address to accept or reject frame.
- **Detecting end of frame:** When to stop reading the frame

Types of framing

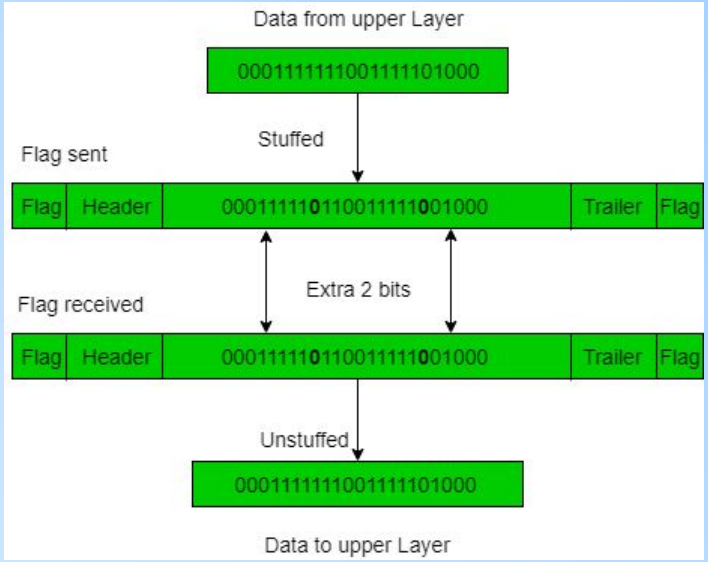
- **Fixed size –** The frame is of fixed size and there is no need to provide boundaries to the frame, the length of the frame itself acts as a delimiter.
  - **Drawback:** It suffers from internal fragmentation if the data size is less than the frame size
  - **Solution:** Padding
- **Variable size –** In this, there is a need to define the end of the frame as well as the beginning of the next frame to distinguish. This can be done in two ways

1. **Length field –** We can introduce a length field in the frame to indicate the length of the frame. Used in **Ethernet(802.3)**. The problem with this is that sometimes the length field might get corrupted.
2. **End Delimiter (ED) –** We can introduce an ED(pattern) to indicate the end of the frame. Used in **Token Ring**. The problem with this is that ED can occur in the data. This can be solved by:
    1. **Character/Byte Stuffing:** Used when frames consist of characters. If data contains ED then, a byte is stuffed into data to differentiate it from ED.
        Let ED = "$" –> if data contains '$' anywhere, it can be escaped using '\O' character.
        –> if data contains '\O$' then, use '\O\O\O$'($ is escaped using \O and \O is escaped using \O).



Disadvantage – It is very costly and obsolete method

- **Bit Stuffing:** Let ED = 01111 and if data = 01111
  –> Sender stuffs a bit to break the pattern i.e. here appends a 0 in data = 0111**0**1.
  –> Receiver receives the frame.
  –> If data contains 011101, receiver removes the 0 and reads the data.



Examples –

•If Data –> 011100011110 and ED –> 0111 then, find data after bit stuffing? –> 0111**0**000111**0**10

•If Data –> 110001001 and ED –> 1000 then, find data after bit stuffing? –> 1100**1**010**0**1**1**

Summary

Part A Functions

1) The problem
2) Where are we now?
3) The Channel Allocation Problem

Part B Dynamic Channel Allocation Technologies

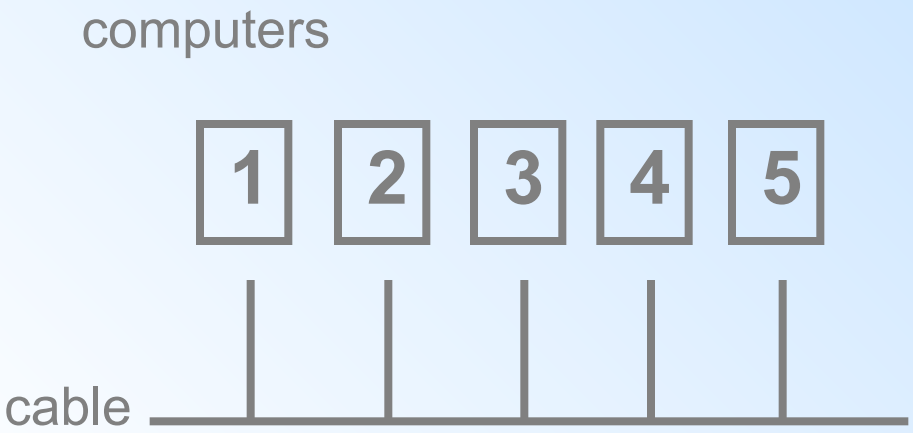
1) ALOHA Protocols
2) CSMA
3) CSMA/CD (old ETHERNET)
4) Switching (Fast ETHERNET)
5) Wireless LANS

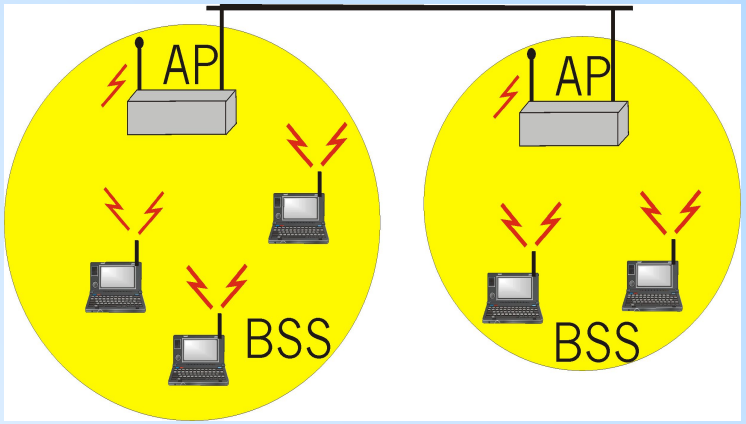
Part A: Access Control

(1)The Problem

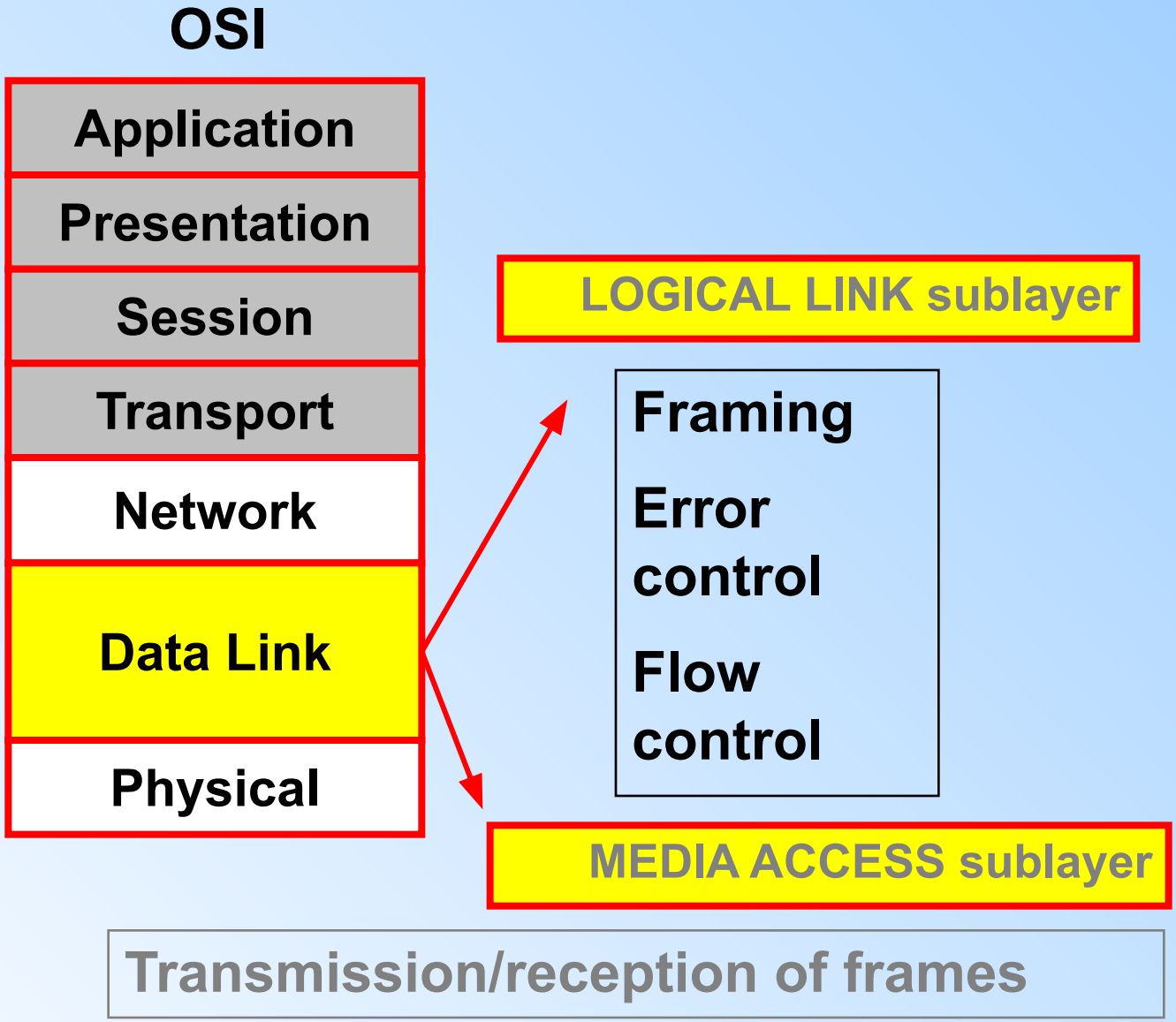
The problem: In a single channel broadcast network, when multiple stations try to send messages simultaneously, who has the right to use the channel?

A common sense:
When we take about MAC, we are faced with a broadcast network.





(2) Where are we now?



(3) The Channel Allocation Problem

- **Static**
  - FDM /TDM (Frequency/Time Division Multiplexing)
    - FDM : Radio/TV broadcasts
    - TDM : POTS (Plain Old Telephone System)
    - GSM uses both (Global System for Mobile Communications)
    - Wasteful of bandwidth and not work well for bursty traffic
- **Dynamic**
  - Pure/ Slotted ALOHA
  - Carrier Sense Multiple Access (CSMA) Protocols
  - Collision free protocols (optional contents)

Part B Dynamic Channel Allocation Technologies

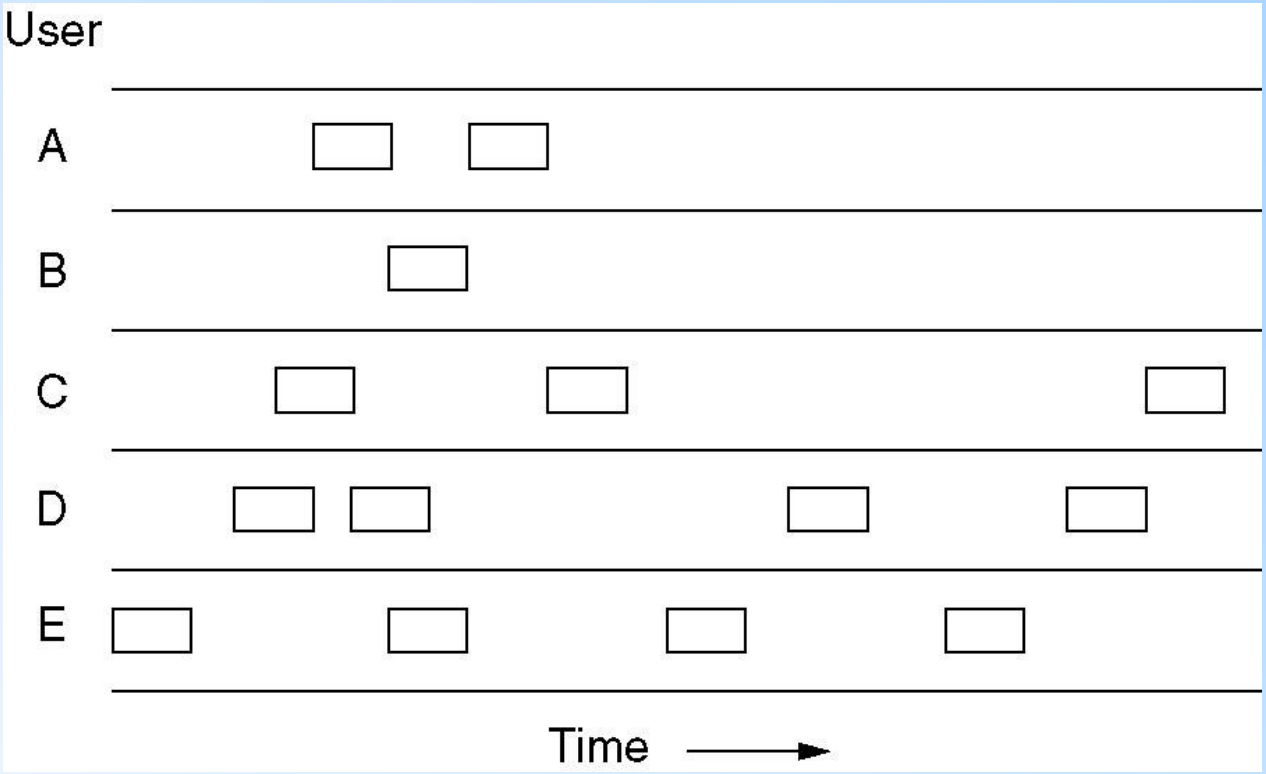
1) ALOHA Protocols
   1.1 Pure ALOHA
   1.2 Slotted ALOHA
2) CSMA
3) CSMA/CD (old ETHERNET)
4) Switching (Fast ETHERNET)
5) Wireless LANS

1) ALOHA Protocols

The core idea is extremely simple:

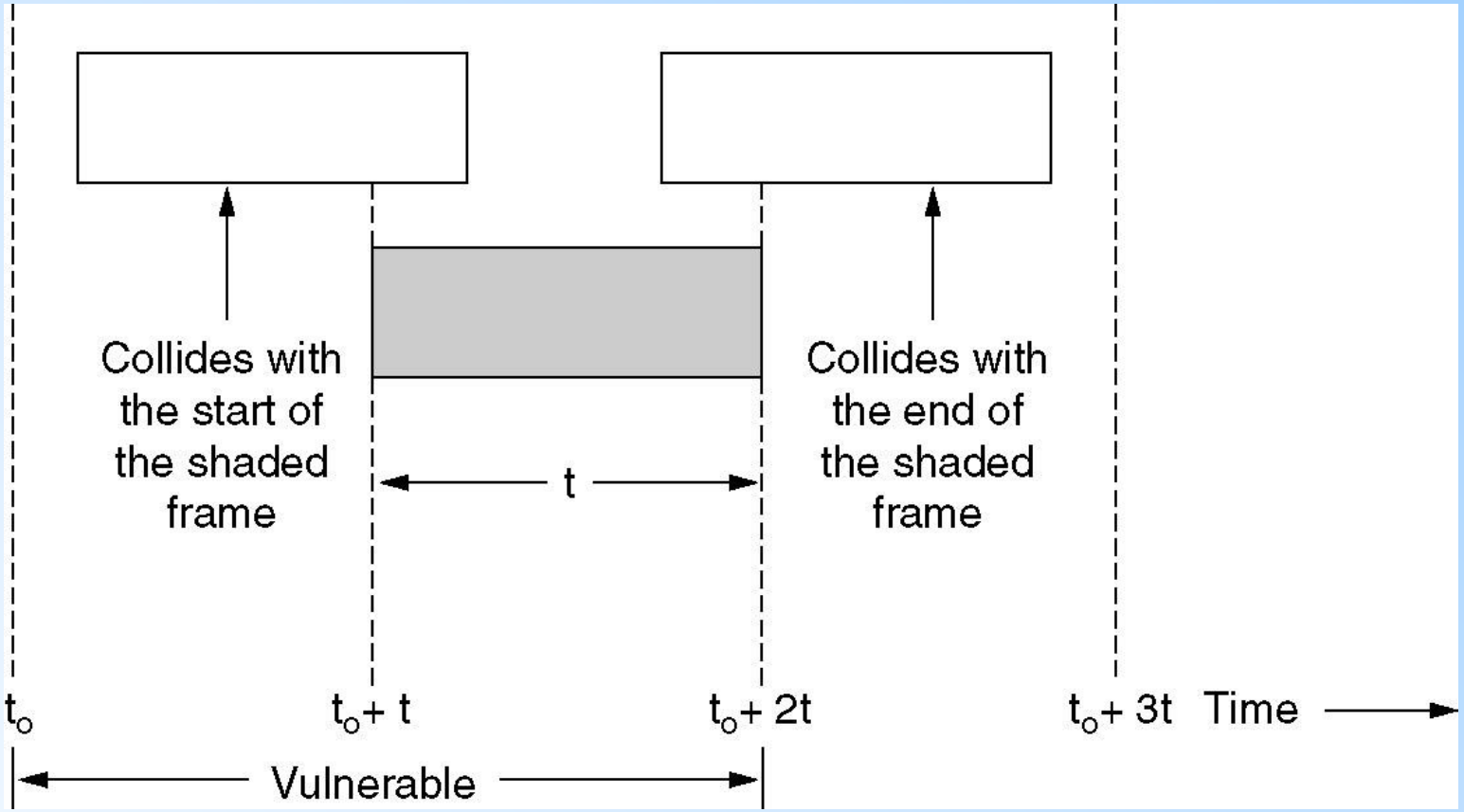
- Anyone may transmit whenever they want. (Continuous time model.)
- Detect if the transmission is successful. (So we need some way for Collision Detection (CD)).
- After a collision, wait a random amount of time and transmit the same frame again. This technique is known as *backoff*.

1.1) Pure ALOHA (1)



- In pure ALOHA, frames are transmitted at completely arbitrary times.

1.1) Pure ALOHA (2)

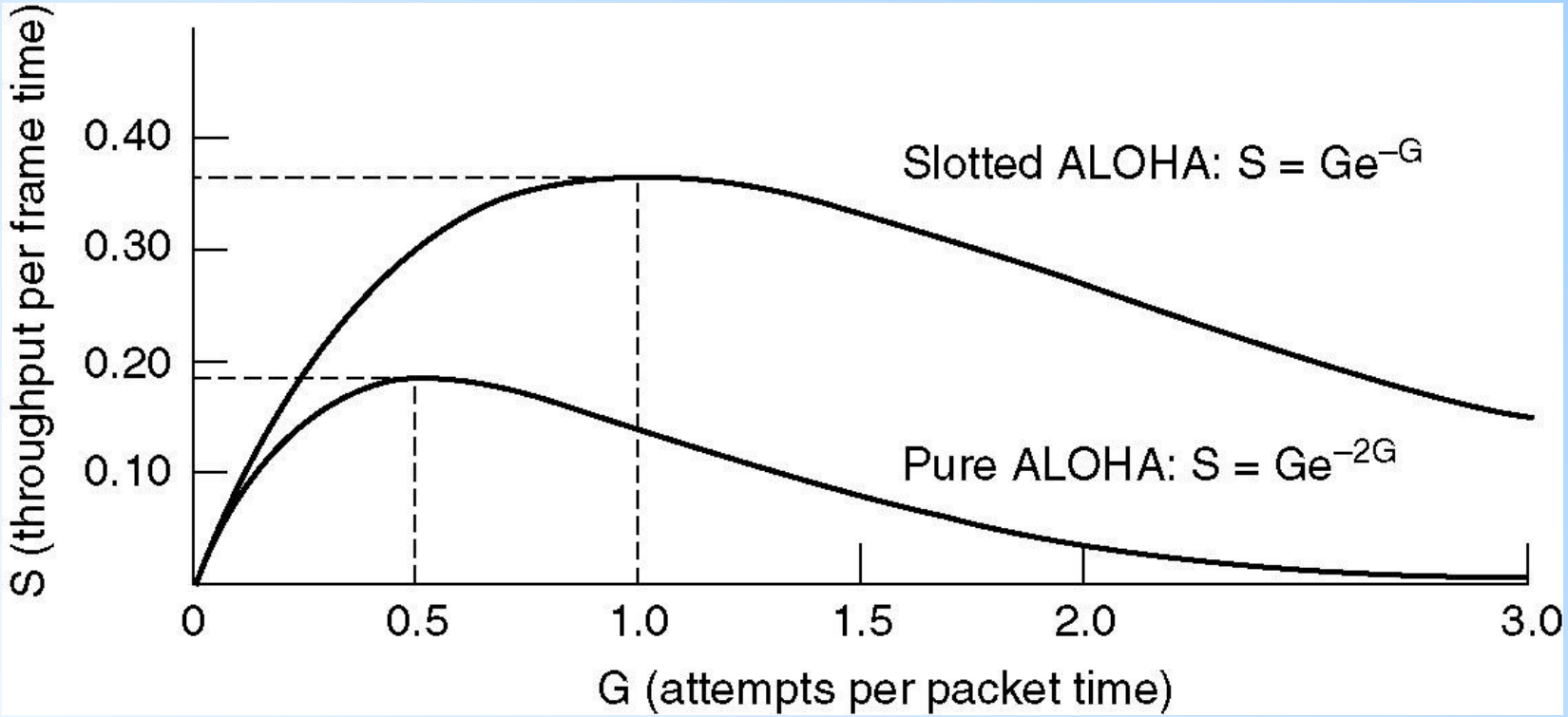


- Vulnerable period for the shaded frame.

1.2) Slotted ALOHA

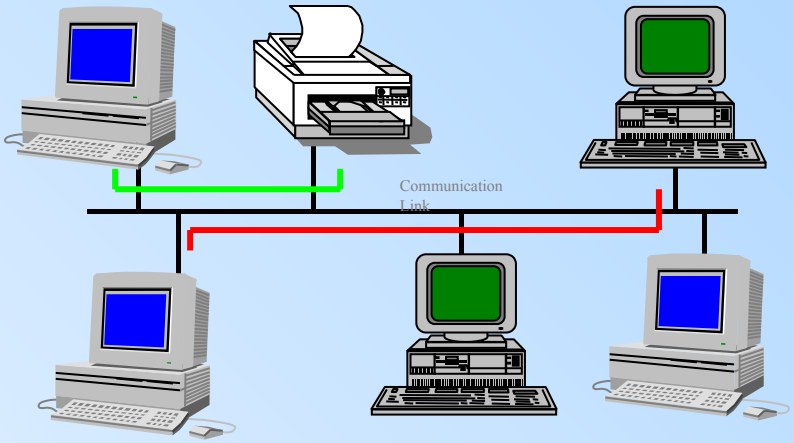
- Time is divided into slots… can only transmit at the start of slot
- Vulnerable period halved => max. eff is doubled
- Requires sync of clocks
- Still poor at hi-loads

1.2) Slotted ALOHA (2)



- Throughput versus offered traffic for ALOHA systems.

2) CSMA (Carrier Sense Multiple Access) (1)



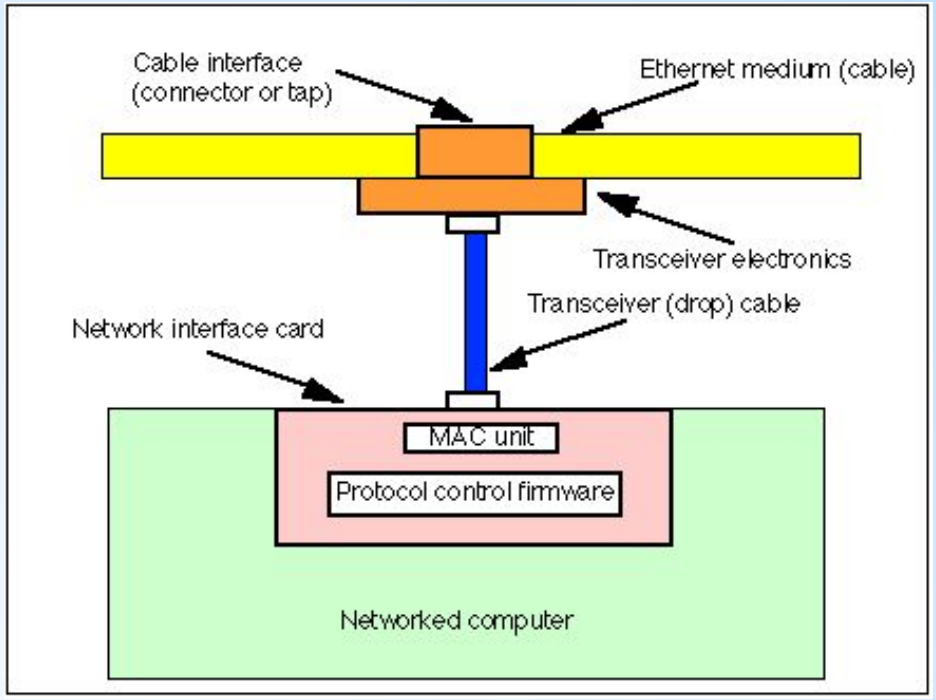
We can improve the performance of our simple network greatly if we introduce **carrier sensing (CS)**. With carrier sensing, each host listens to the data being transmitted over the cable.

- A host will only transmit its own frames when it cannot hear any data being transmitted by other hosts.
- When a frame finishes, an *interframe gap* is allowed to pass before another host starts transmitting its frame.

2) CSMA (2)

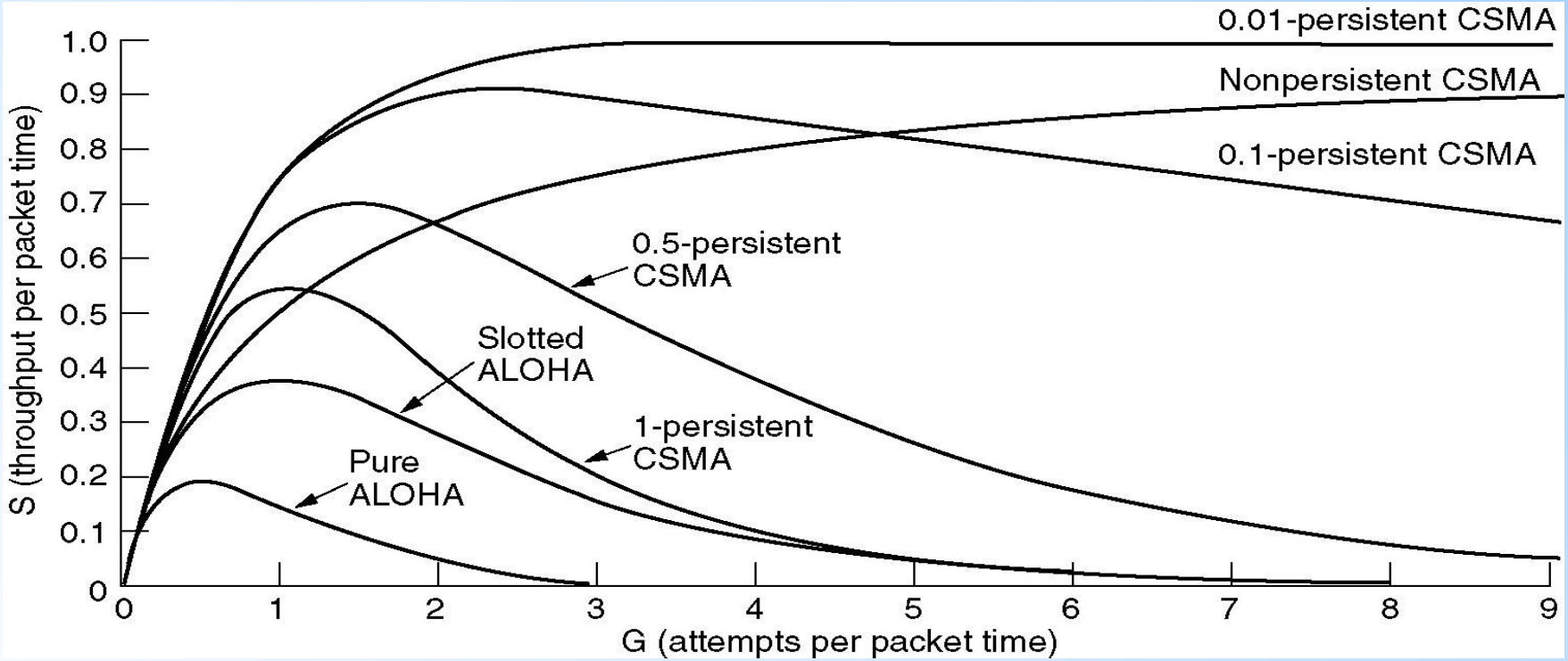
Improves performance when higher medium utilisation

When a node has data to transmit, the node ***first*** listens to the cable (using a transceiver) to see if a carrier (signal) is being transmitted by another node.

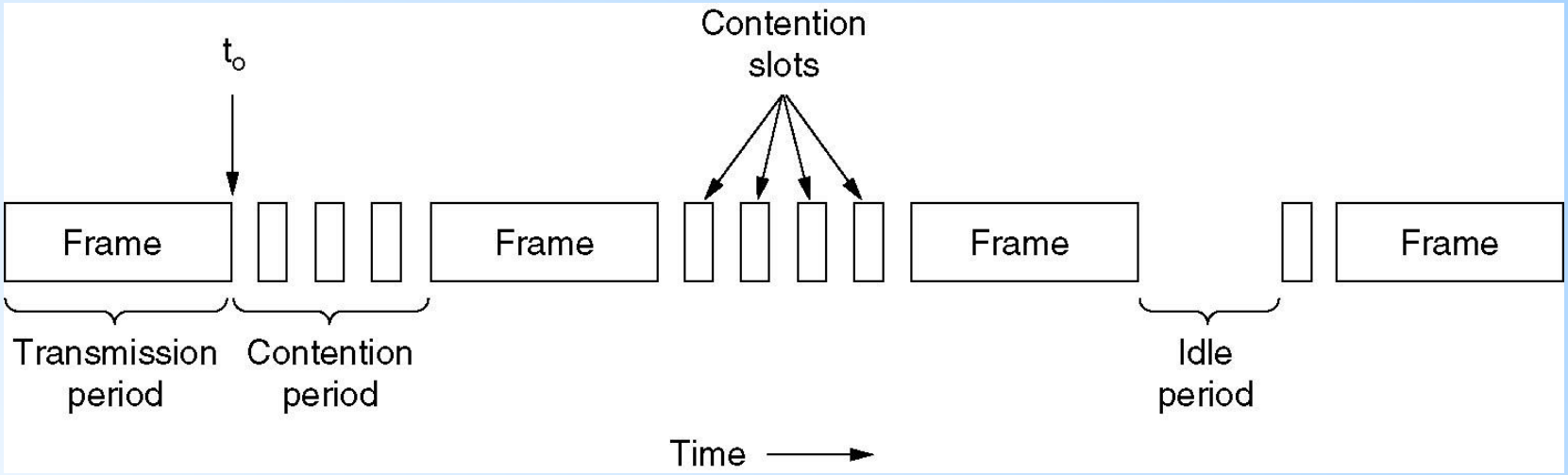


2) CSMA (3): Persistent and Nonpersistent CSMA

- Comparison of the channel utilization versus load for various random access protocols.



3) CSMA with Collision Detection (CSMA/CD) (1)



- CSMA/CD can be in one of three states: contention, transmission, or idle.

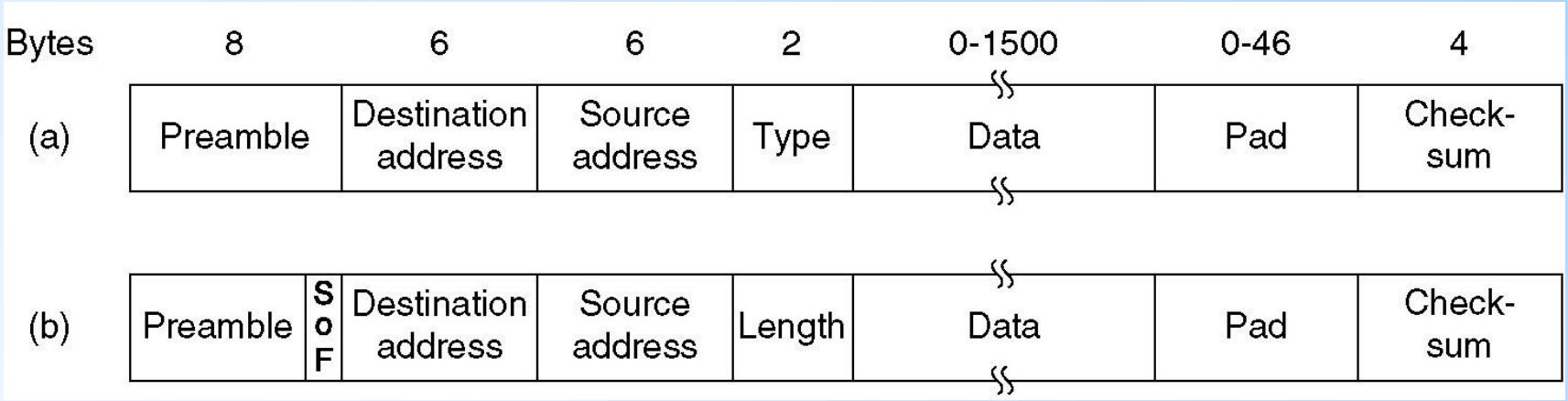
3) CSMA/CD (2): IEEE 802.3 Bus LAN

The 802.3 standard describes the operation of the MAC sub-layer in a bus LAN that uses carrier sense, multiple access with collision detection (CSMA/CD).

- Beside carrier sensing, collision detection and the binary exponential back-off algorithm, the standard also describes the format of the frames and the type of encoding used for transmitting frames.
- The minimum length of frames can be varied from network to network. This is important because, depending on the size of the network, the frames must be of a suitable minimum length.
- The standard also makes some suggestions about the type of cabling that should be used for CSMA/CD bus LANs.

The CSMA/CD Bus LAN is also widely called **Ethernet**.

3) CSMA/CD (3): Ethernet MAC Sublayer Protocol



- Frame formats. (a) DIX Ethernet,
- (b) IEEE 802.3.

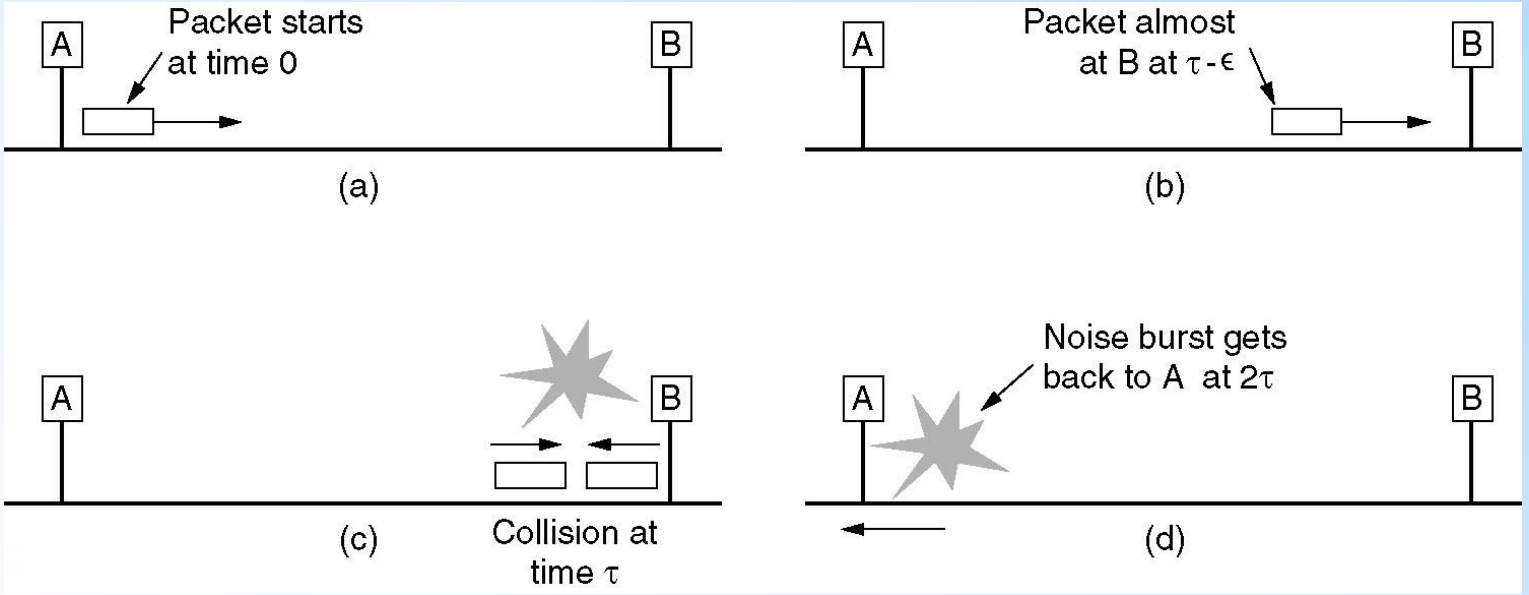
3) CSMA/CD (4): Ethernet MAC Sublayer Protocol

Every network card in the world has a unique 46-bit serial number called a **MAC address**. The IEEE allocates these numbers to network card manufacturers who encode them into the firmware of their cards.

- The destination and source address fields of the MAC frame have 48 bits set aside (the standard also allows for 16-bit addresses but these are rarely used).
- The most significant bit is set to 0 to indicate an ordinary address and 1 to indicate a group address (this is for *multicasting*, which means that frames are sent to several hosts). If all 48 bits are set to 1 then frames are *broadcast* to all the hosts.
- If the two most significant bits are both zero then the 46 least significant bits contain the MAC addresses of the source and destination hosts.

3) CSMA/CD (5): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

- When a host transmits a frame, there is a small chance that a collision will occur. The first host to detect a collision transmits a 48-bit jam sequence.
- To ensure that any hosts involved with the collision realise that the jam sequence is associate with their frame, they must still be transmitting when the jam sequence arrives. This means that the frame must be of a minimum length.
- The worse case scenario is if the two hosts are at far ends of the cable. If host A's frame is just reaching host B when it begins transmitting, host B will detect the collision first and send a jam signal back to host A.



3) CSMA/CD (6): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

- To ensure that no node may completely receive a frame before the transmitting node has finished sending it, Ethernet defines a minimum frame size (i.e. no frame may have less than 46 bytes of payload).

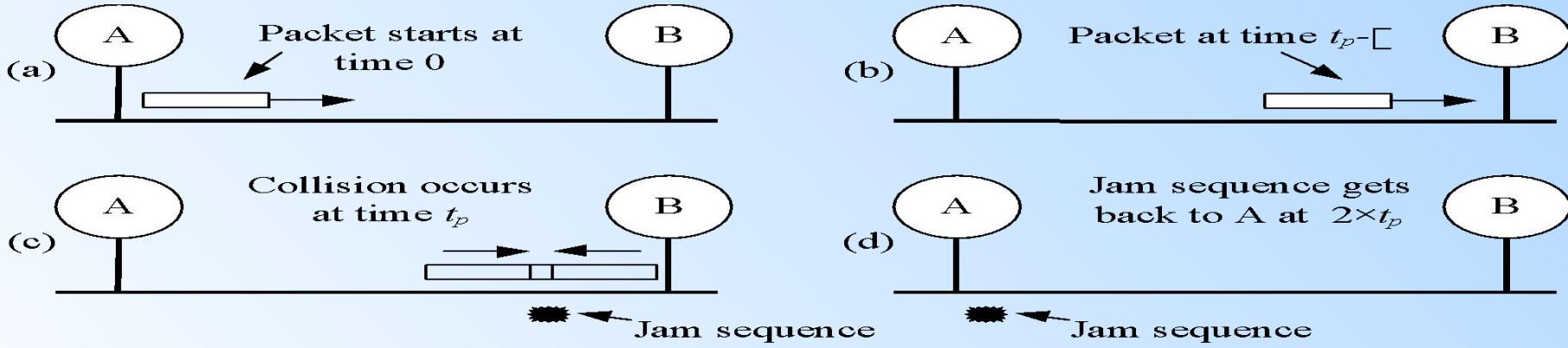
- The minimum frame size is related to the distance which the network spans, the type of media being used and the number of repeaters which the signal may have to pass through to reach the furthest part of the LAN.

- Together these define a value known as the *Ethernet Slot Time*, corresponding to 512 bit times at 10 Mbps.

3) CSMA/CD (7): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

The longest time between starting to transmit a frame and receiving the first bit of a jam sequence is twice the propagation delay from one end of the cable to the other.

- This means that a frame must have enough bits to last twice the propagation delay.
- The 802.3 CSMA/CD Bus LAN transmits data at the standard rate of r = 10Mbps.
- The speed of signal propagation is about $v = 2\times10^8$m/s.



3) CSMA/CD (8): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

Example #1: Cable = 400m, trans. speed = 10 Mbit/sec, propagation speed = 2*10^8 m/sec

$$t_{prop} = \frac{d}{V} = \frac{400}{2 \times 10^8} = 2 \times 10^{-6} = 2\mu\,\text{sec}$$

$$2 \times t_{prop} = 4\mu\,\text{sec}$$

$$R = 10Mbps$$

$$t_b = \frac{1}{R} = \frac{1}{10{,}000{,}000} = 0.1\mu\,\text{sec}$$

$$n_b = 2 \times \frac{t_p}{t_b} = \frac{4}{0.1} = 40 \ \ bits$$

A margin of error is usually added to this (often to make it a power of 2) so we might use 64 bits (8 bytes).

3) CSMA/CD (9): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

Example 2: Speed transmission is 100 Mbits/sec; frame size is 1500 bytes; the propagation speed is 3*10^8 m/sec.
Calculate the distance between the nodes such that the
time to transmit the frame = time to recognize that the collision have occurred.

$$t_{frame} = \frac{L}{R} = \frac{1500 \times 8}{100 \times 10^6} = 1.2 \times 10^{-4}$$

$$T_{round\_trip} \longrightarrow t_{frame} = 2 \times t_{prop}$$

$$t_{prop} = \frac{t_{frame}}{2} = \frac{1.2 \times 10^{-4}}{2} = 6 \times 10^{-5}$$

$$t_{prop} = \frac{d}{V}$$

$$d = t_{prop} \times V = \left(6 \times 10^{-5}\right) \times \left(3 \times 10^{8}\right) = 18 \times 10^{3} = 18 \ km$$

3) CSMA/CD (10): Ethernet MAC Sublayer Protocol (Minimum Frame Length)

The standard frame length is at least 512 bits (64 bytes) long, which is much longer than our minimum requirement of 64 bits (8 bytes).

- We only have to start worrying when the LAN reaches lengths of more than 2.5km.

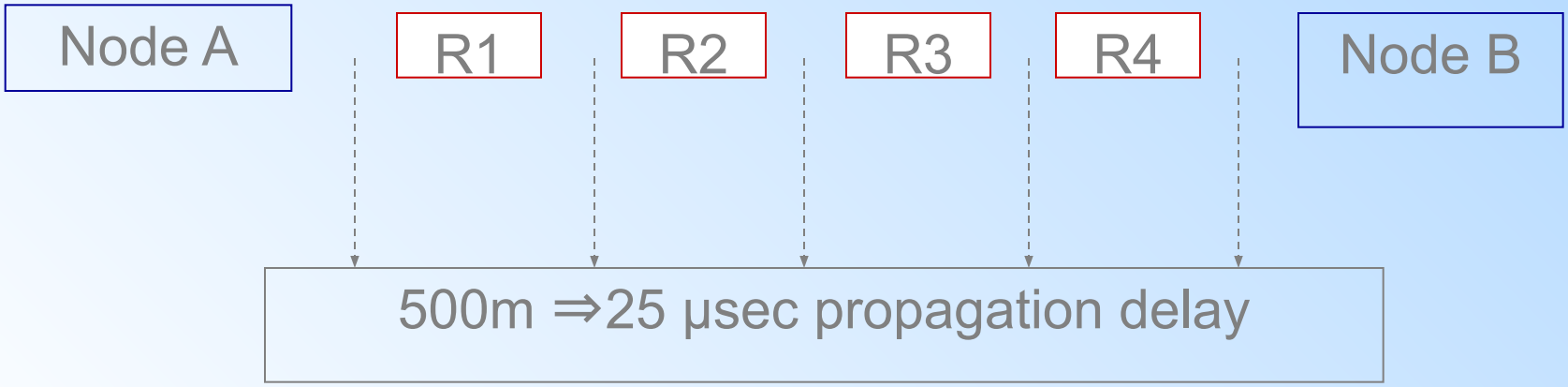
802.3 CSMA/CD bus LANs longer than 500m are usually composed of multiple segments joined by *in-line passive repeaters*, which output on one cable the signals received on another cable.

- When we work out the minimum frame length for these longer LANs, we also have to take the delays caused by the passive repeaters (about 2.5μsec each) into account as well.

3) CSMA/CD (11): Ethernet MAC Sublayer Protocol (Shortest Ethernet Frame)

Why specify a shortest frame of 64byte?

- 64 bytes sent at 10Mbps ⇒ 51.2μsec
- 500m/segment, 4 repeaters between nodes ⇒2500m ⇒25 μsec propagation delay
- The frame should be longer enough for sender to detect the collision(2x25 or about 50 μsec)



3) CSMA/CD (12): Ethernet MAC Sublayer Protocol (Non-Deterministic)

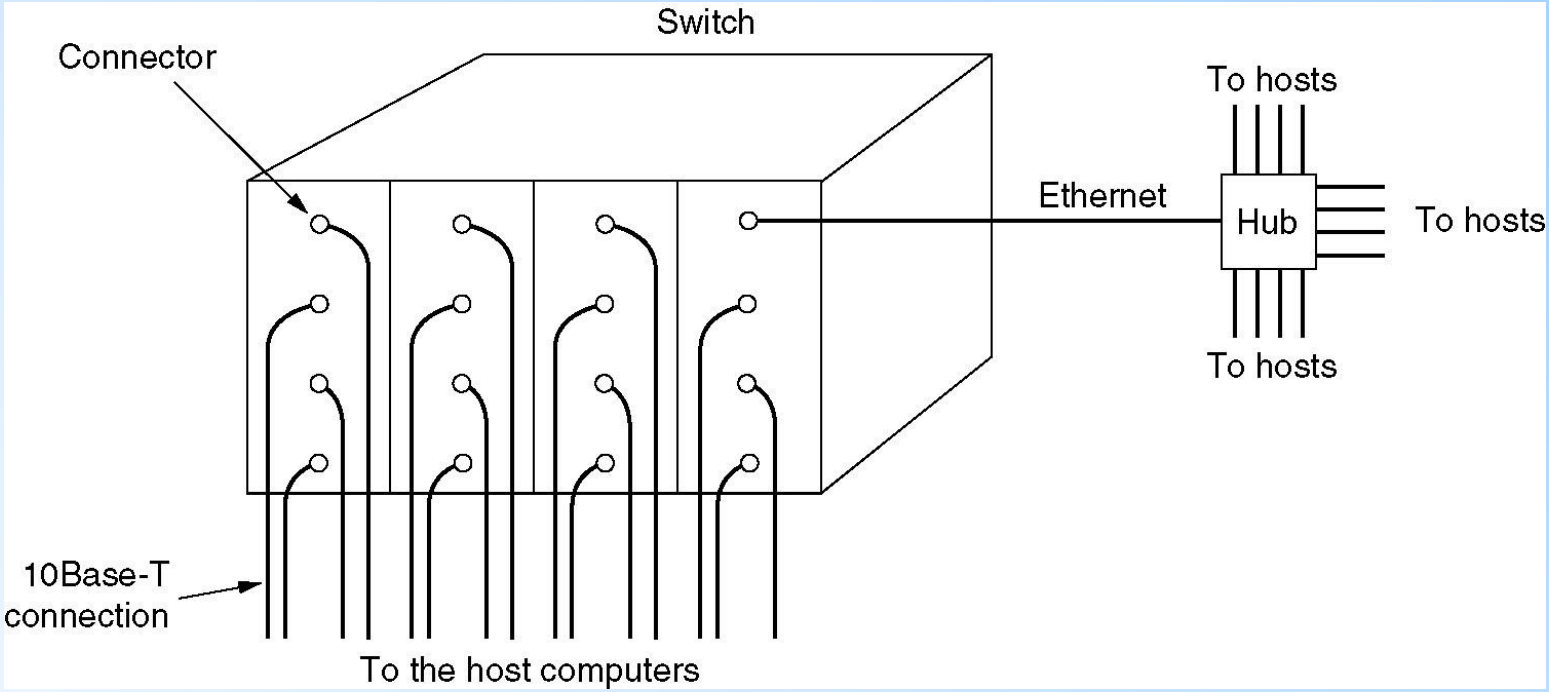
The 802.3 CSMA/CD bus LAN is said to be a **non-deterministic** network. This means that no host is guaranteed to be able to send its frame within a reasonable time (just a good probability of doing so).

- When the network is busy, the number of collisions rises dramatically and it may become very difficult for any hosts to transmit their frames.

A **real-time** computing application (such as an assembly line) will demand that data is transmitted within a specified time period.

- Since the 802.3 bus LAN cannot guarantee this, its use for real-time applications may not only be undesirable but potentially dangerous in some situations.

4) Switched Ethernet



- A simple example of switched Ethernet.

5) Wireless LANs

- 5.1 RF allocation
- 5.2 The 802.11 Protocol Stack
- 5.3 The 802.11 MAC Sublayer Protocol
- 5.4 The 802.11 Frame Structure

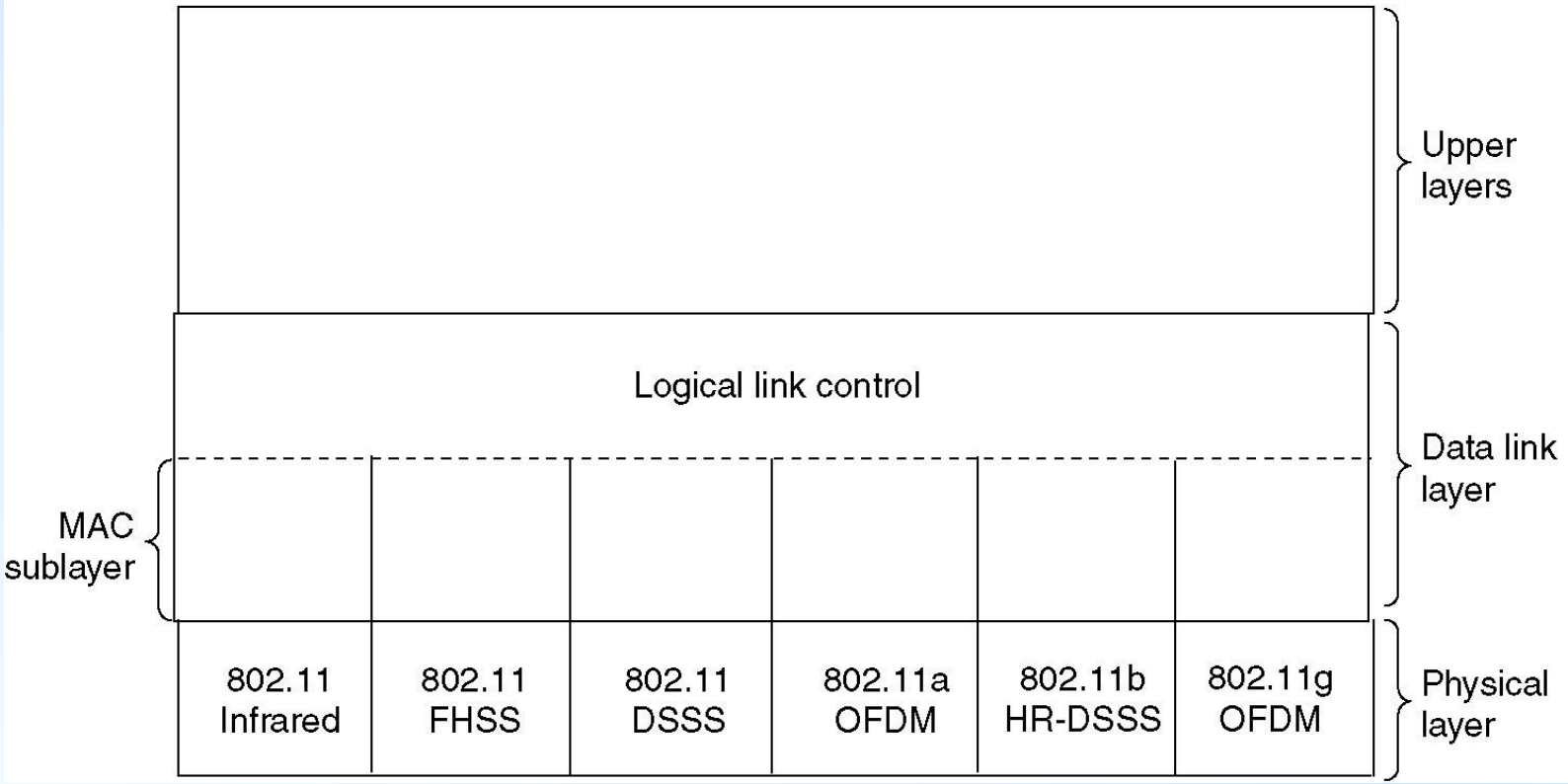
5.1) Where Does Wireless RF Live?

ISM Band: Industrial, Scientific, Medical

902-928 MHz	2400-2483.5 MHz	5725-5850 MHz
Old Wireless	802.11/802.11b	802.11a
	Bluetooth	
	Cordless Phones	
	Home RF	
	Baby Monitors	
	Microwave Ovens	

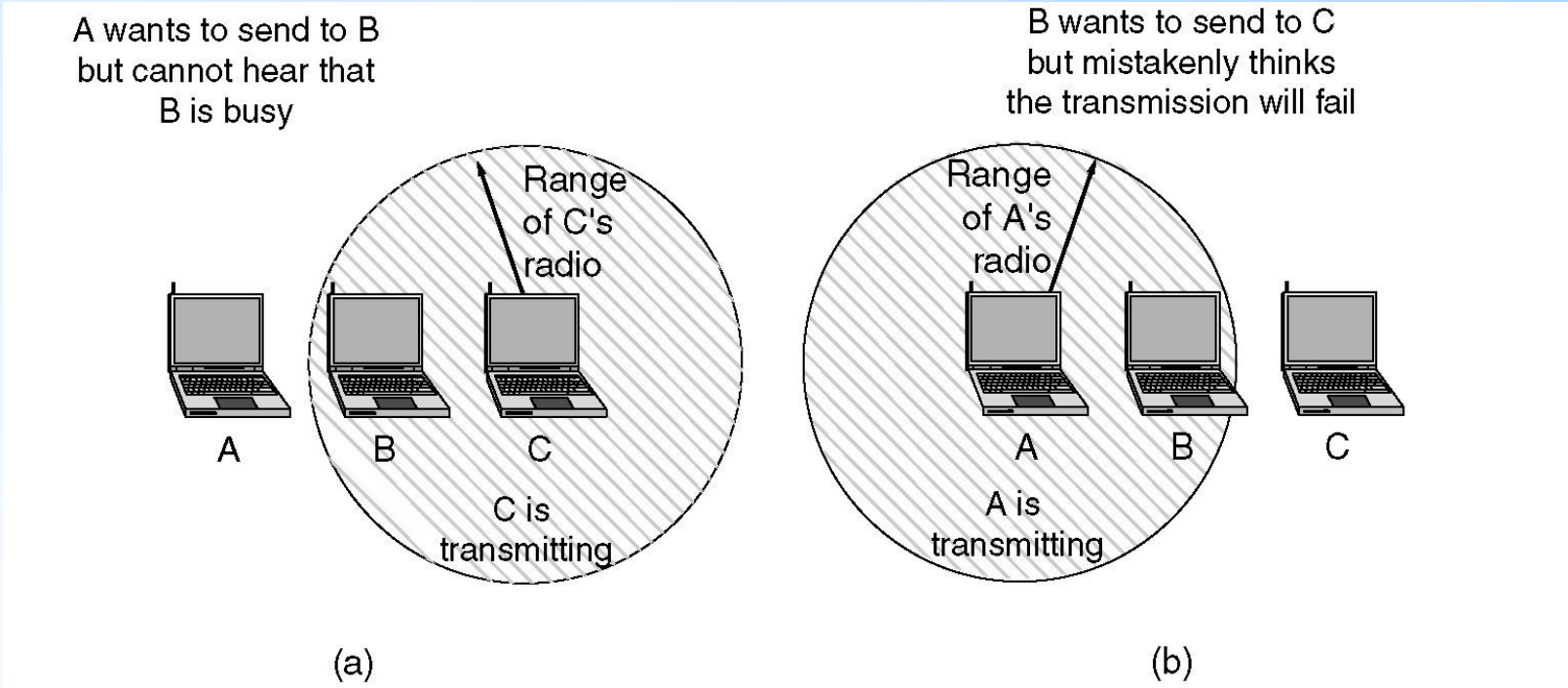
5.2) The 802.11 Protocol Stack

- Part of the 802.11 protocol stack.



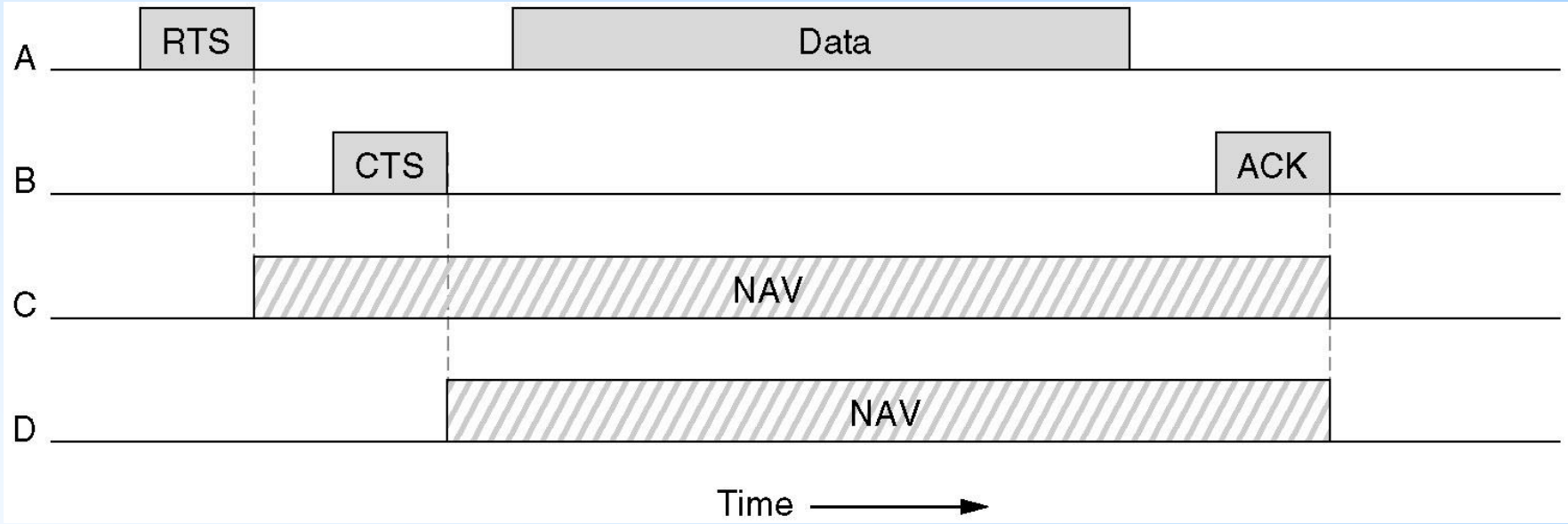
5.3) The 802.11 MAC Sublayer Protocol (1)

- (a) The hidden station problem.
- (b) The exposed station problem.



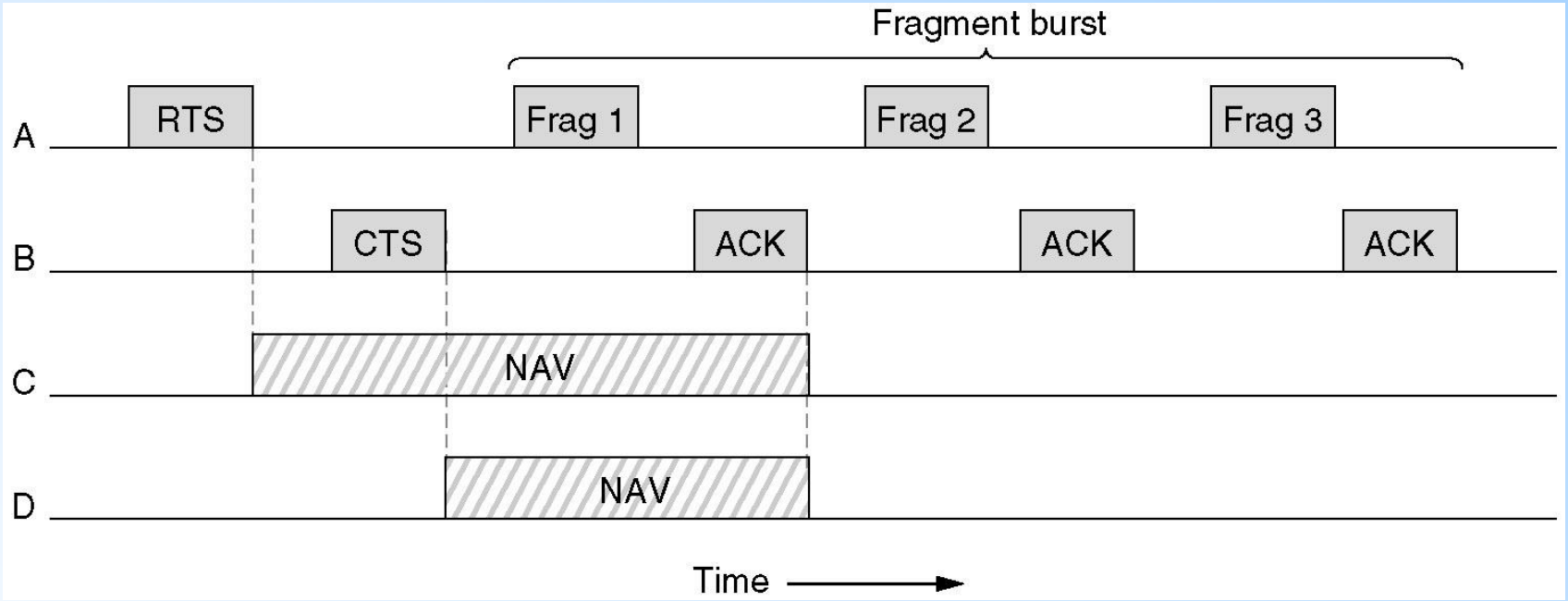
5.3) The 802.11 MAC Sublayer Protocol (2)

- The use of virtual channel sensing using CSMA/CA.



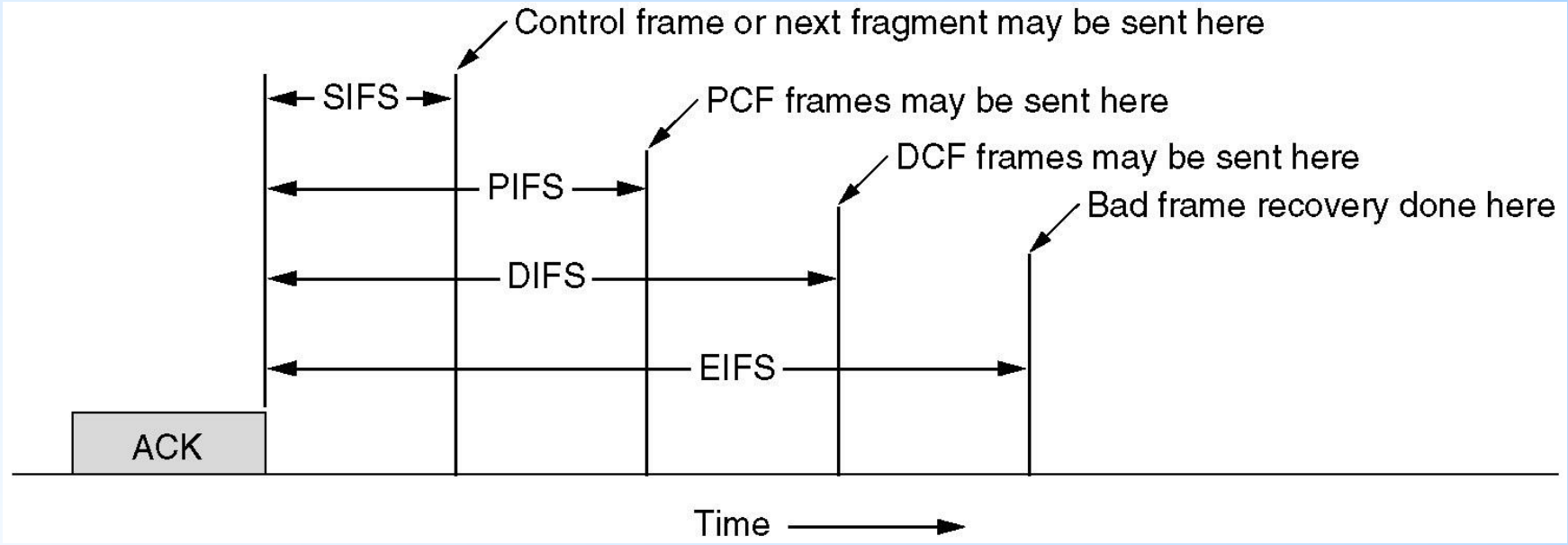
5.3) The 802.11 MAC Sublayer Protocol (3)

- A fragment burst.



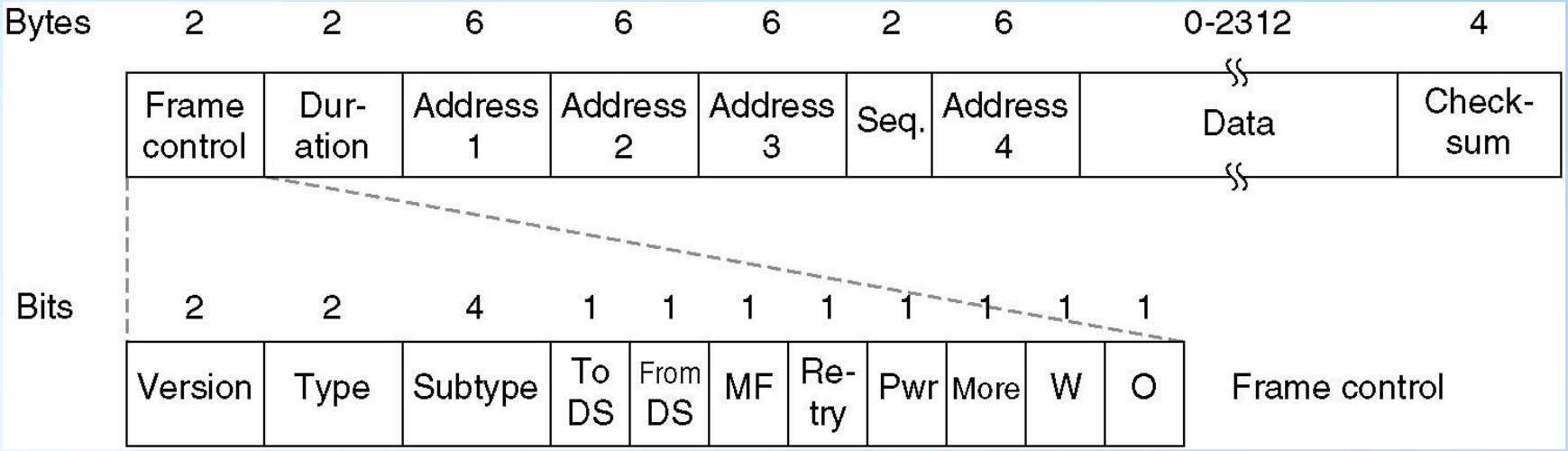
5.3) The 802.11 MAC Sublayer Protocol (4)

- Interframe spacing in 802.11.



5.4) The 802.11 Frame Structure

- The 802.11 data frame.



Overview

- Design issues
- Point-to-point links
- Local area Networks
- Data Link layer Switching
- Services
- Protocols

Design issues

- Algorithms
  protocols
  services
  for achieving reliable, efficient communication between systems connected by a "wire-like" communication channel
- Trivial?
  - Errors on communication circuits
  - Finite data rate
  - Nonzero propagation delay
  - Finite processing speed

Design issues

- Services to network layer
  - Unacknowledged connectionless service
  - Acknowledged connectionless service
  - Acknowledged connection-oriented service

- Services from physical layer
  - Unreliable bit transport

Design issues: Services

Connectionless service

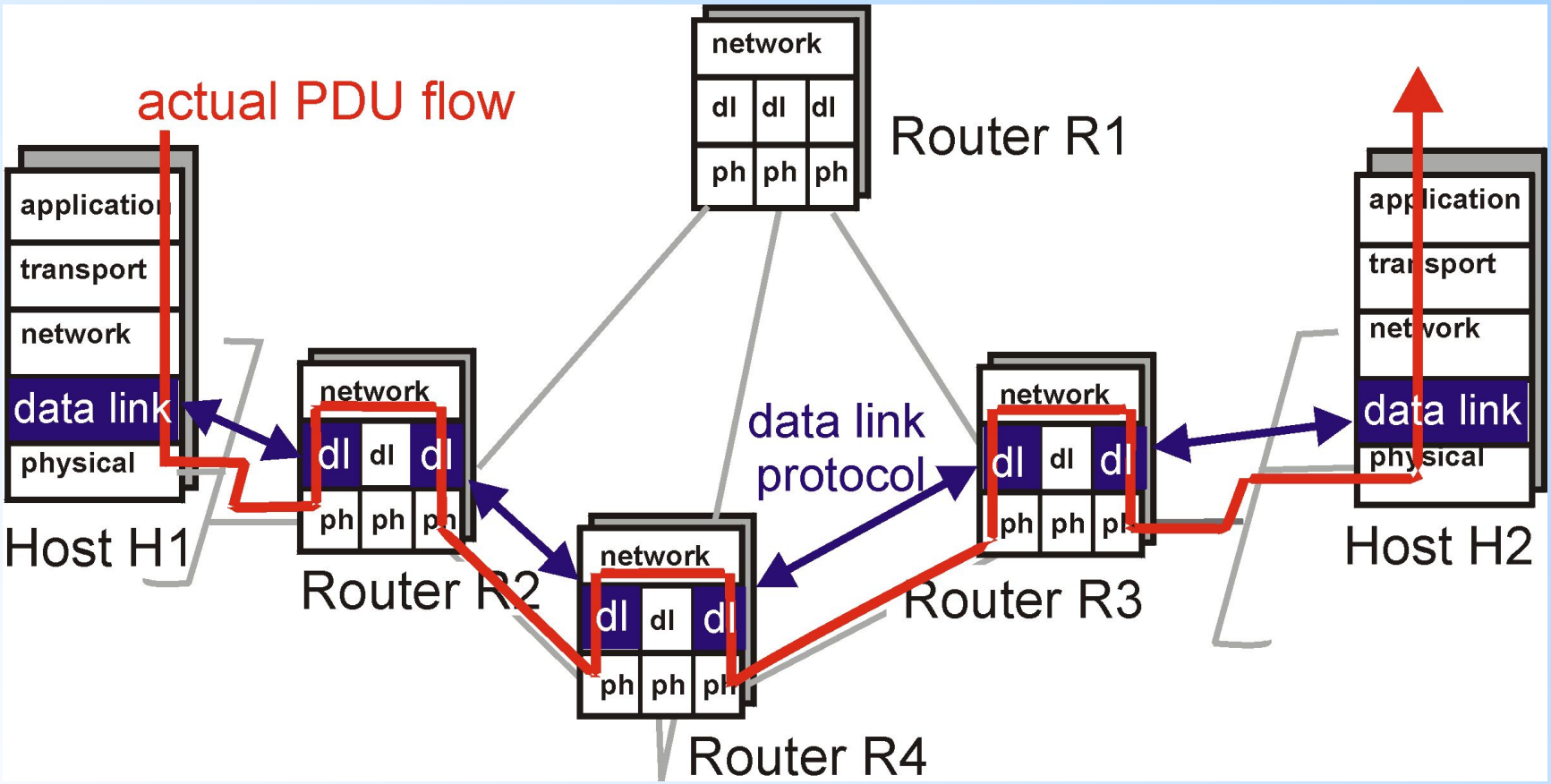
- Unacknowledged
  - Independent frames
  - Error rate should be low
  - Recovery left to higher layers
  - Used on LANs
- Acknowledged
  - No connection
  - Acknowledgement for each packet
  - Resending
  - Ack is optimisation; also transport layer can handle errors

Connection-oriented service

- Acknowledged
  - Each frame received exactly once
  - All frames received in the right order
  - 3 distinct phases:
    - Establishment of connection
    - Data transfer
    - Release of connection

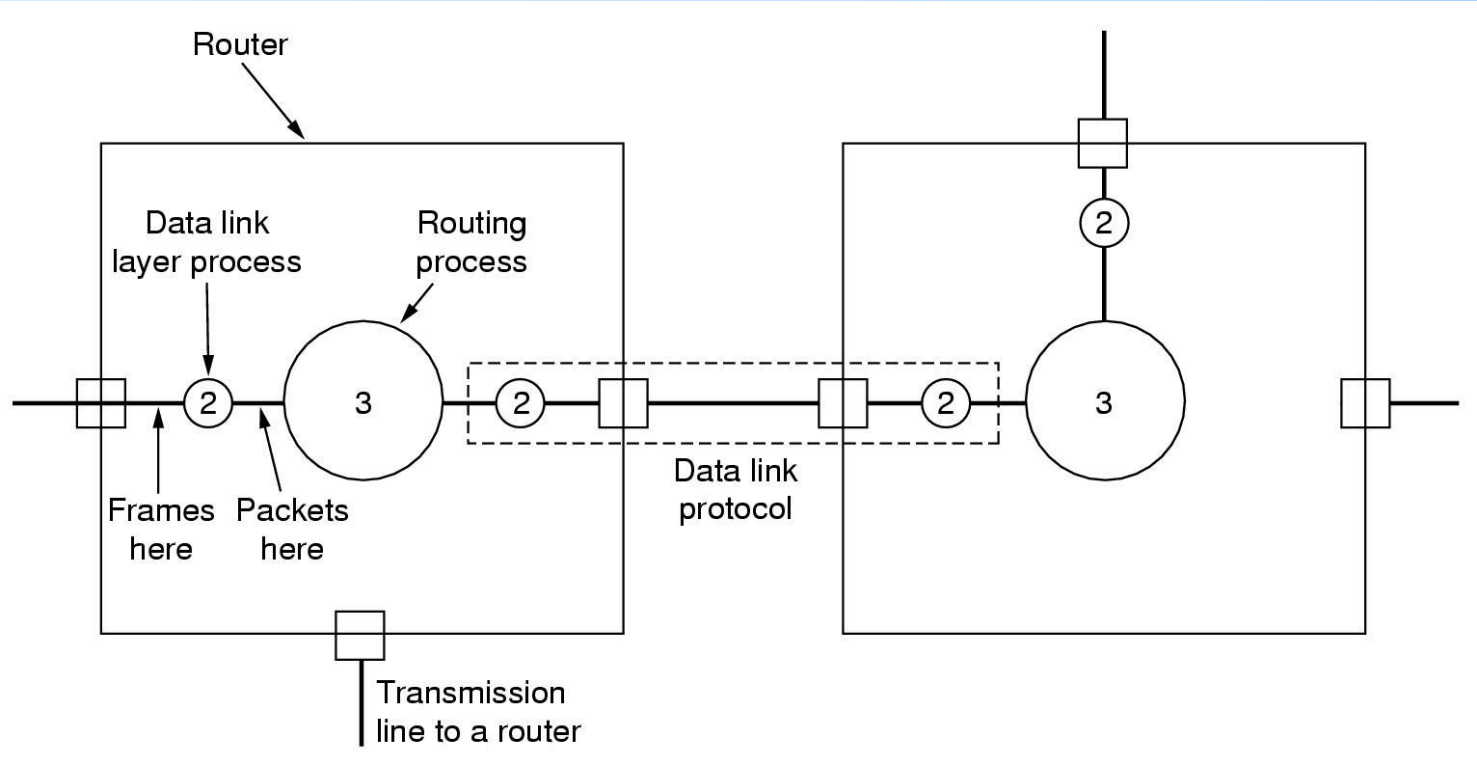
Design issues: protocols

- Position of data link protocol



Design issues: protocols

- Position of data link protocol



Design issues: protocols

- Framing
  - Break stream of bits up into discrete frames
- Error control
  - How does a sender know that all packets are correctly received
- Flow control
  - How to prevent a sender to overload the receiver with packets

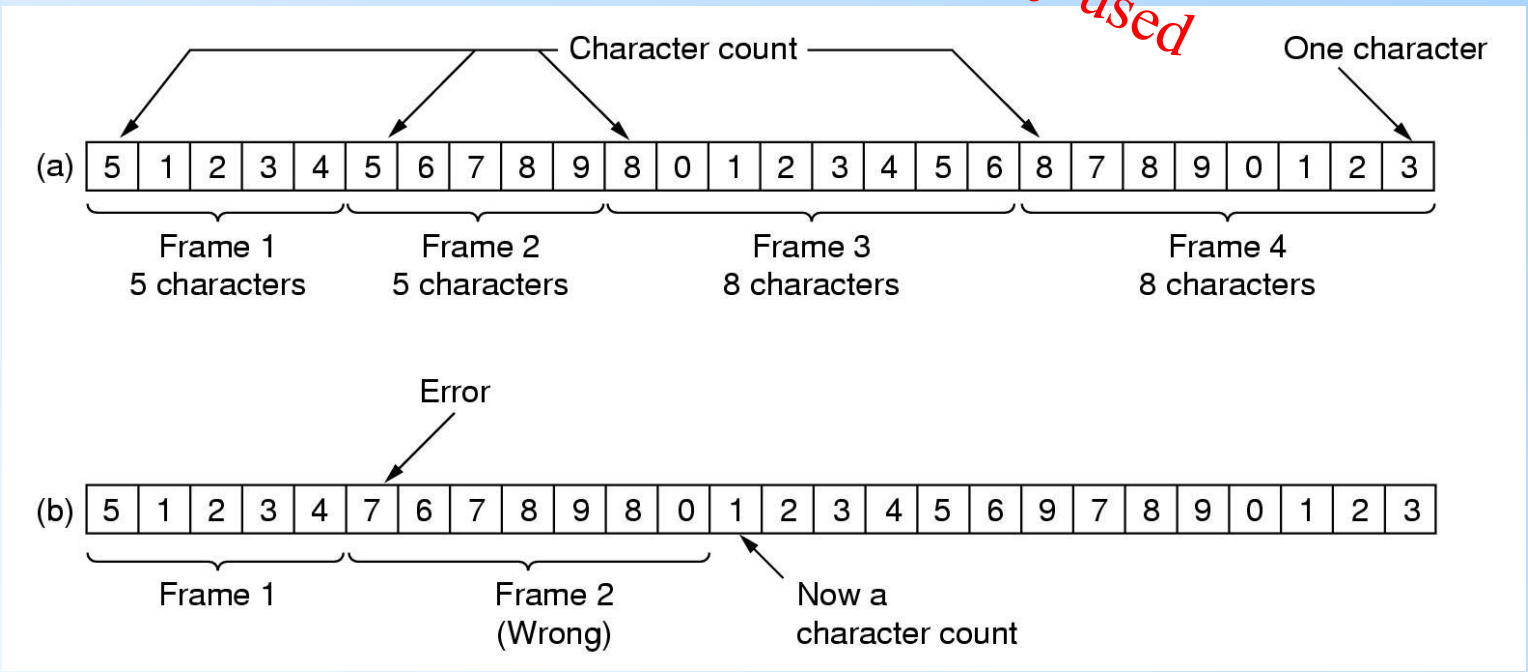
Design issues: protocols

- Framing: break stream of bits up into discrete frames
- Methods:
  - Use of time gap: unacceptable, too risky
  - Character count
  - Starting and ending characters with character stuffing
  - Starting and ending flags with bit stuffing
  - Physical layer coding violations

Design issues: protocols

- Framing: Character count
  - Frame contains length (or #characters)

Rarely used



  - Out of synchronisation if error

Design issues: protocols

- Framing: Character stuffing

DLE	STX	A	B	C	D	DLE	ETX

  - Frame starts / ends with special sequence of chars
  - Allow all chars as data in frame?
    - Stuffing: additional DLE before each DLE

A	DLE	C

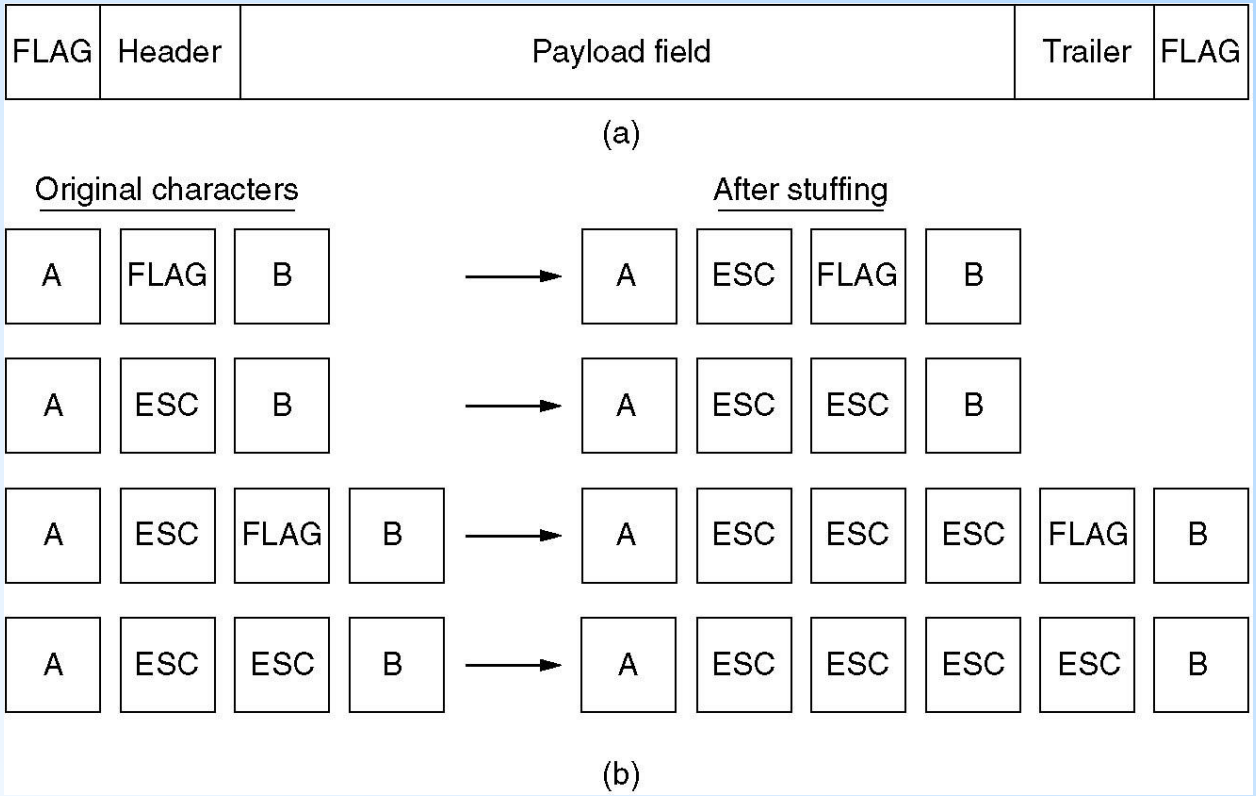
DLE	STX	A	DLE	DLE	C	DLE	ETX

  - Char set
    - DLE: Data link escape
    - STX: Start Text
    - ETX: End Text

Too closely linked to ASCII

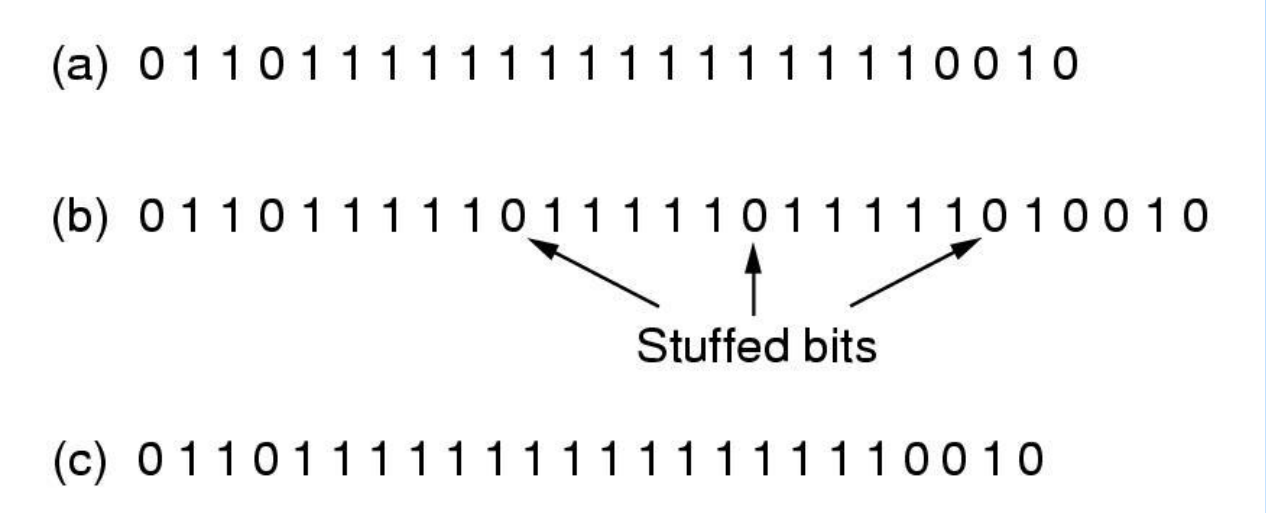
Design issues: protocols

- Framing: Character stuffing – newer protocols
  - Frame starts / ends with same char: FLAG



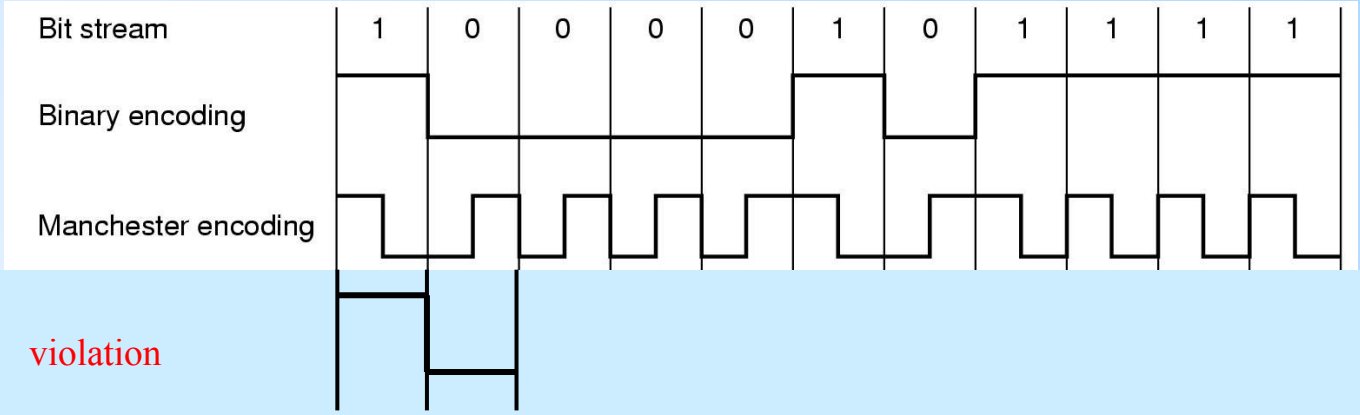
Design issues: protocols

- Framing: Bit stuffing
  - Special bit pattern for start / end of frame
    01111110
  - In data: add ‘0’ after 5 consecutive ‘1’



Design issues: protocols

- Framing: Coding violation
  - Redundancy in the encoding on medium is required
  - e.g. Manchester encoding: transition in the middle of a slot



  - Use no transition in a slot (= coding violation) as start of frame

Design issues: protocols

- Error control
  - How does a sender know that all packets are correctly received?
    - ACK packet by receiver
  - Packet lost or not recognized at receiver?
    - Timer at sender: resend packet
  - How to handle duplicates or out of order received packets?
    - Sequence number in packet and ACK packet
  - Optimisation: NACK packet
    - Inform sender that something strange happened

Design issues: protocols

- Flow control
  - How to prevent a sender to overload the receiver with packets
    - Receiver gives permission to send more packets
  - Mechanisms:
    - Implicit: ACK packet implies permission
    - Explicit: communicate window size

Part 2 - Topics

- Sliding Window Protocols
- Go Back N Sliding Window Protocol
- Selective Repeat Sliding Window Protocol

Data Frame Transmission

- Unidirectional assumption in previous elementary protocols

  $\Rightarrow$ Not general
- Full-duplex - approach 1
  - Two separate communication channels
    - Forward channel for data
    - Reverse channel for acknowledgement

      $\Rightarrow$ Problems: 1. reverse channel bandwidth wasted

      2. cost

Data Frame Transmission

- Full-duplex - approach 2
  - Same circuit for both directions
  - Data and acknowledgement are intermixed
  - How do we tell acknowledgement from data?

    "*kind*" field telling data or acknowledgement
  - Can it be improved?
- Approach 3
  - Attaching acknowledgement to outgoing data frames

    ⇒ Piggybacking

Piggybacking

- Temporarily delaying transmission of outgoing acknowledgement so that they can be hooked onto the next outgoing data frame
- Advantage: higher channel bandwidth utilization
- Complication:
  - How long to wait for a packet to piggyback?
  - If longer than sender timeout period then

    sender retransmit

    ⇒ Purpose of acknowledgement is lost

Piggybacking

- Solution for timing complexion
  - If a new packet arrives quickly
    ⇒ Piggybacking
  - If no new packet arrives after a receiver ack timeout
    ⇒ Sending a separate acknowledgement frame

Sliding Window Protocol

- Each outbound frame contains an *n*-bit sequence number
  - Range: 0 - MAX_SEQ (MAX_SEQ = 2^n - 1)
  - For stop-and-wait, *n* = __. Why?
- At any instance of time
  - Sender maintains a set of sequence numbers of frames *permitted to send*
    - These frames fall within *sending window*
  - Receiver maintains a set of sequence numbers of frames *permitted to accept*
    - These frames fall within *receiving window*

Sliding Window Protocol

- Lower limit, upper limit, and size of two windows *need not be the same*
- Fixed or variable size
- Requirements
  - Packets delivered to the receiver's network layer must be in the same order that they were passed to the data link layer on the sending machine
  - Frames must be delivered by the physical communication channel in the order in which they were sent

Sending Window

- Contains frames can be sent or have been sent but not yet acknowledged – *outstanding* frames
- When a packet arrives from network layer
  - Next highest sequence number assigned
  - Upper edge of window advanced by 1
- When an acknowledgement arrives
  - Lower edge of window advanced by 1

Sending Window

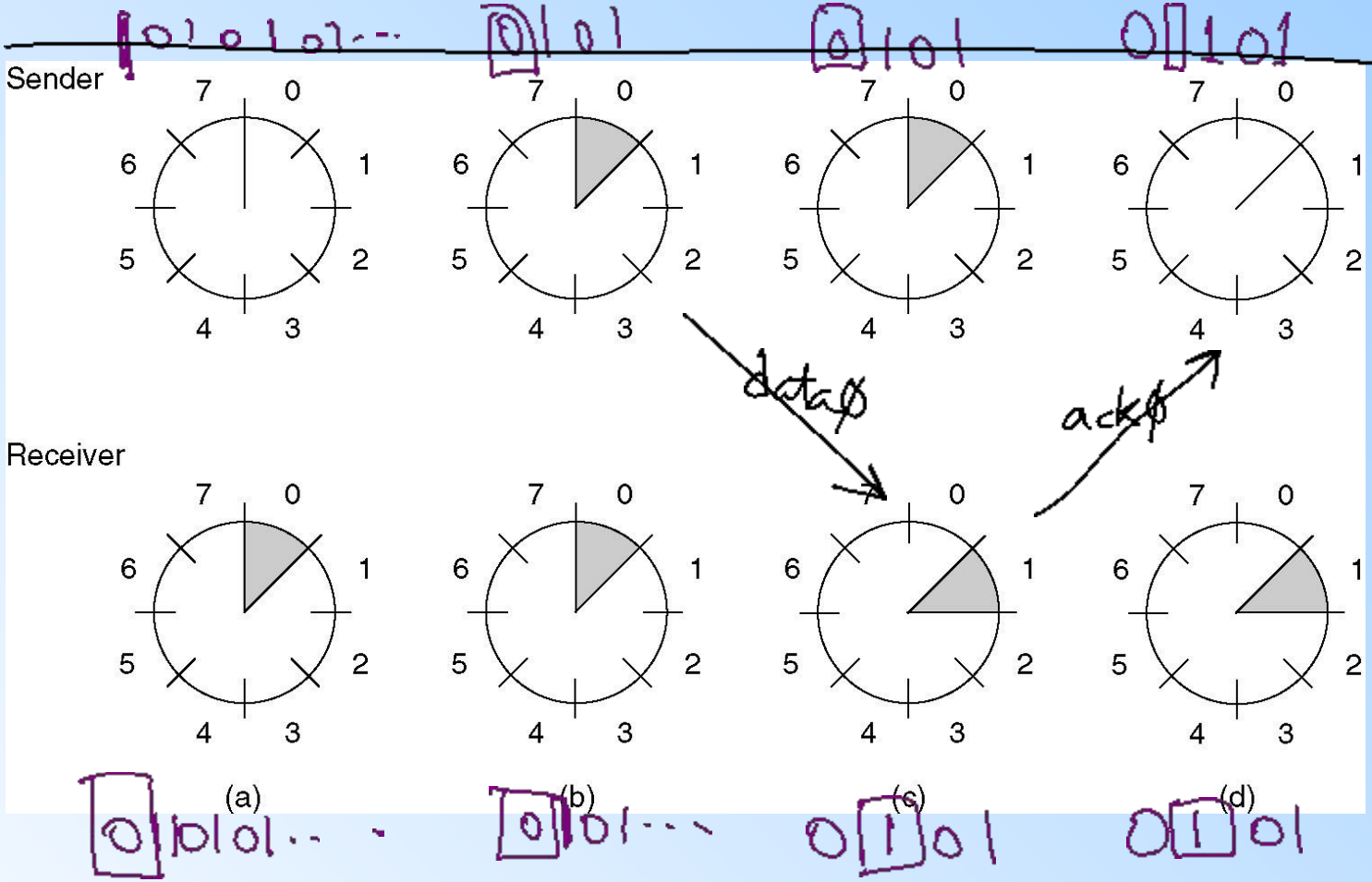
- If the maximum window size is n, n buffers is needed to hold unacknowledged frames
- Window full (maximum window size reached)
  $\Rightarrow$ shut off network layer

Receiving Window

- Contains frames may be accepted
- Frame outside the window discarded
- When a frame's sequence number equals to lower edge
  - Passed to the network layer
  - Acknowledgement generated
  - Window rotated by 1

Receiving Window

- Contains frames may be accepted
- Always remains at initial size (different from sending window)
- Size
  - =1 means frames only accepted in order
  - >1 not so
- Again, the order of packets fed to the receiver's network layer must be the same as the order packets sent by the sender's network layer



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.

Sliding Window Protocol

- We are going to study three bidirectional *sliding window protocols* (max sending window size, receiving window size)
  - One-bit sliding window protocol (1, 1)
  - Go back N (>1, 1)
  - Selective repeat (>1, >1)
- Differ in efficiency, complexity, and buffer requirements

PtP: window protocols

- One bit sliding window protocol
  - Assumptions:
    - Data in both directions: piggybacking of ACK
      - No separate ACK packets
    - Noisy channel
    - Long stream of data in both directions
  - Protocol: Tanenbaum: fig. 3.14

One Bit Sliding Window Protocol

- Sending window size = receiving window size = 1
- Stop-and-wait
- Refer to algorithm in Fig 3-16
- Acknowledgement =

  Sequence number of last frame received w/o error*
- Problem of sender and receiver send simultaneously

*: some protocols define the acknowledgement to be the sequence number expected to receive

1-bit sliding window protocol

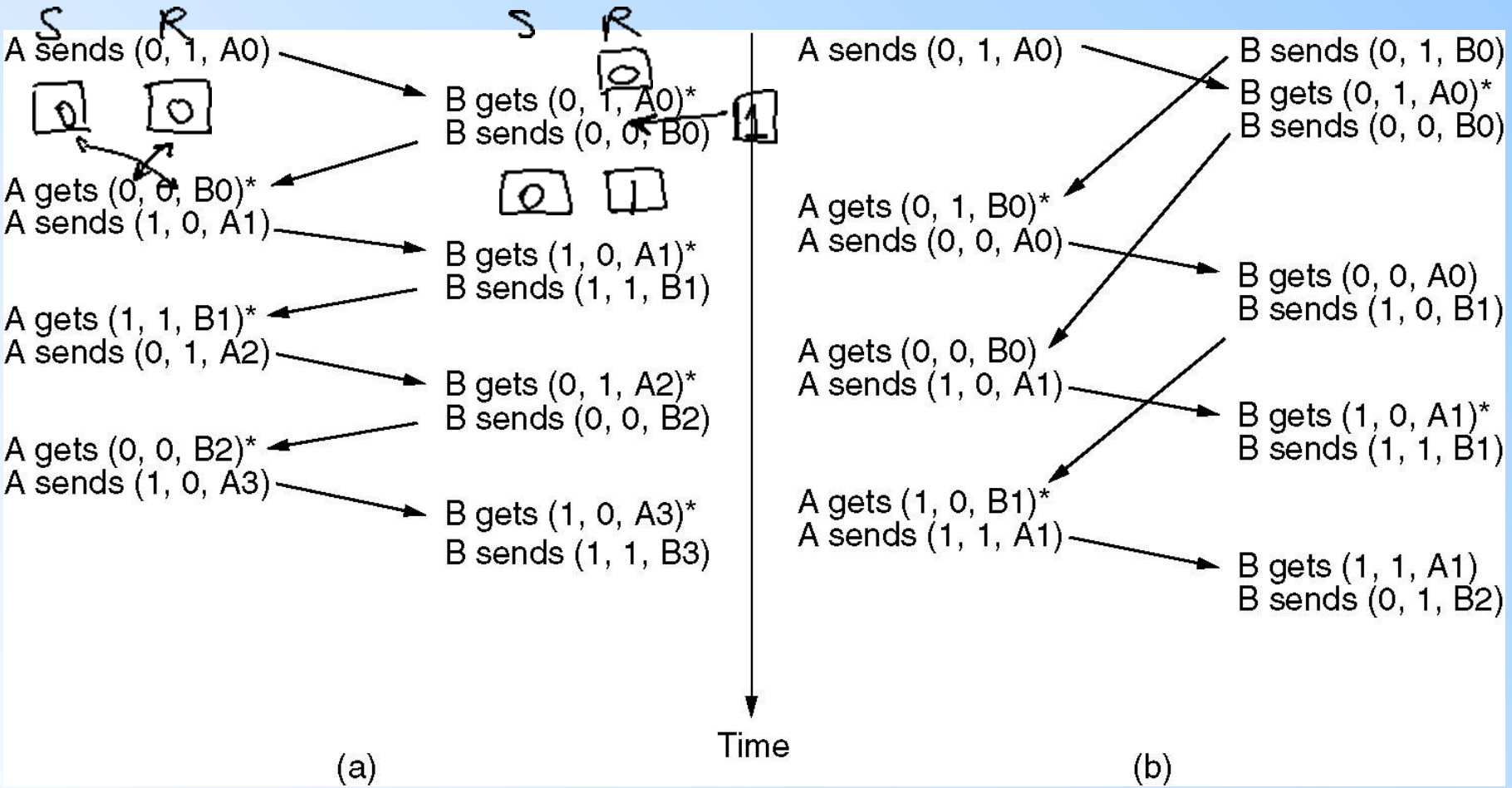
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /*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

1-bit sliding window protocol

```
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 */
}
}
```

Case 1: normal case

Case 7: simultaneous start

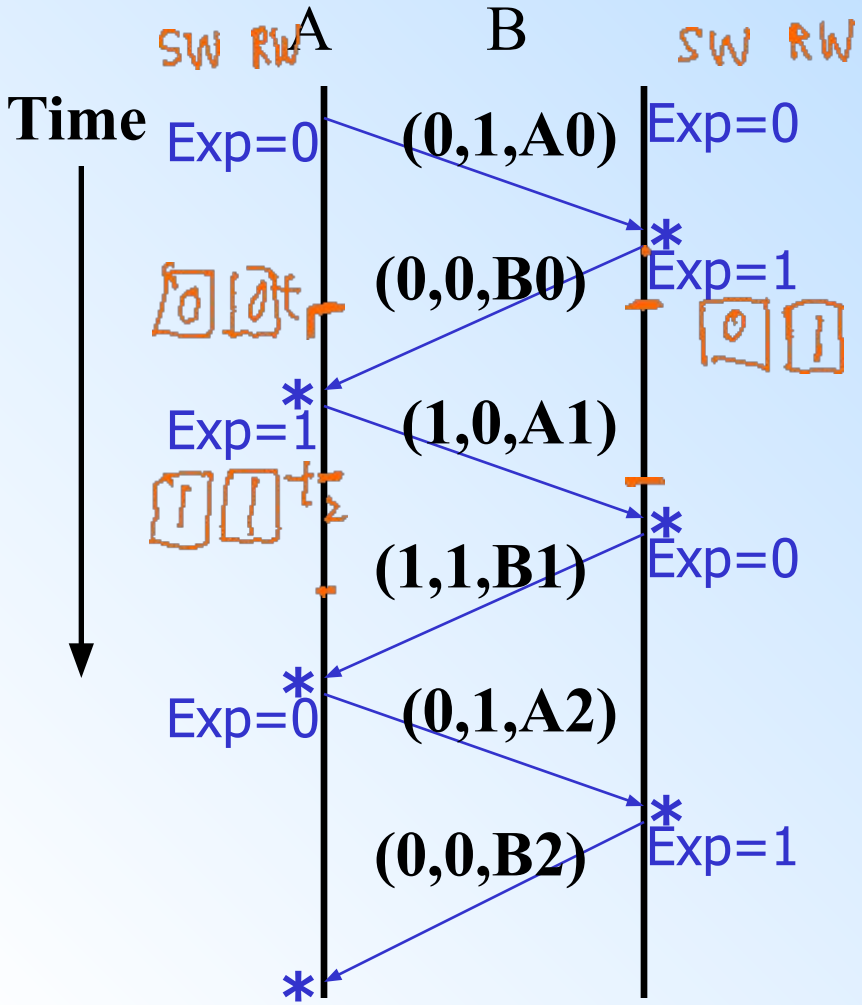


(a) Case 1: Normal case. (b) Case 7: Abnormal case. The notation is **(seq, ack, packet number)**. An asterisk indicates where a network layer accepts a packet.

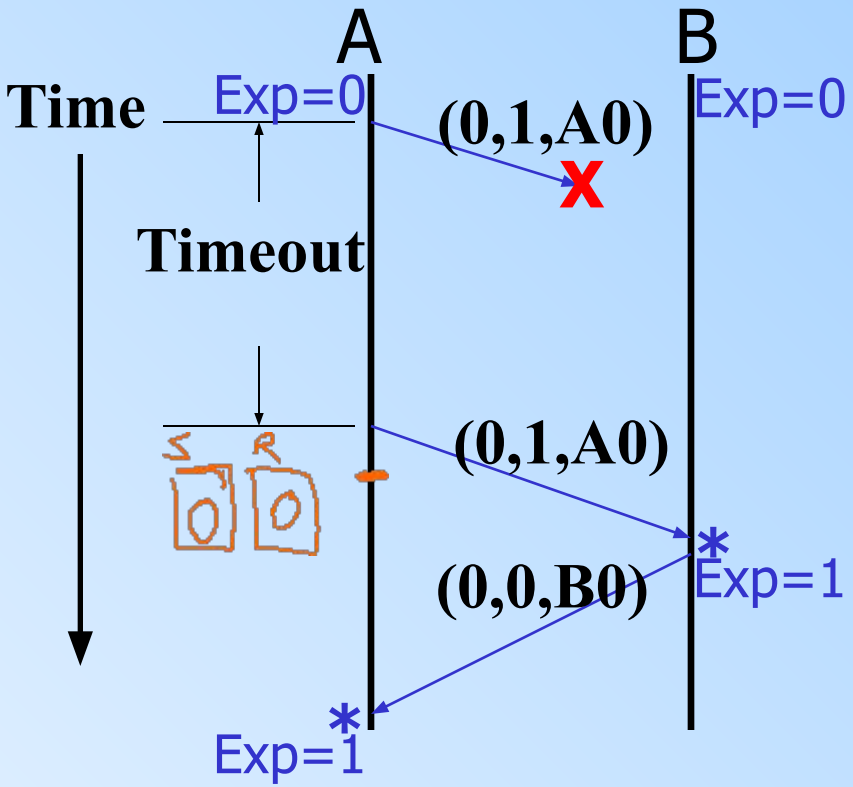
Try to draw the sending windows and receiving windows for A and B!

One Bit Sliding Window Protocol

- Case 1: no error



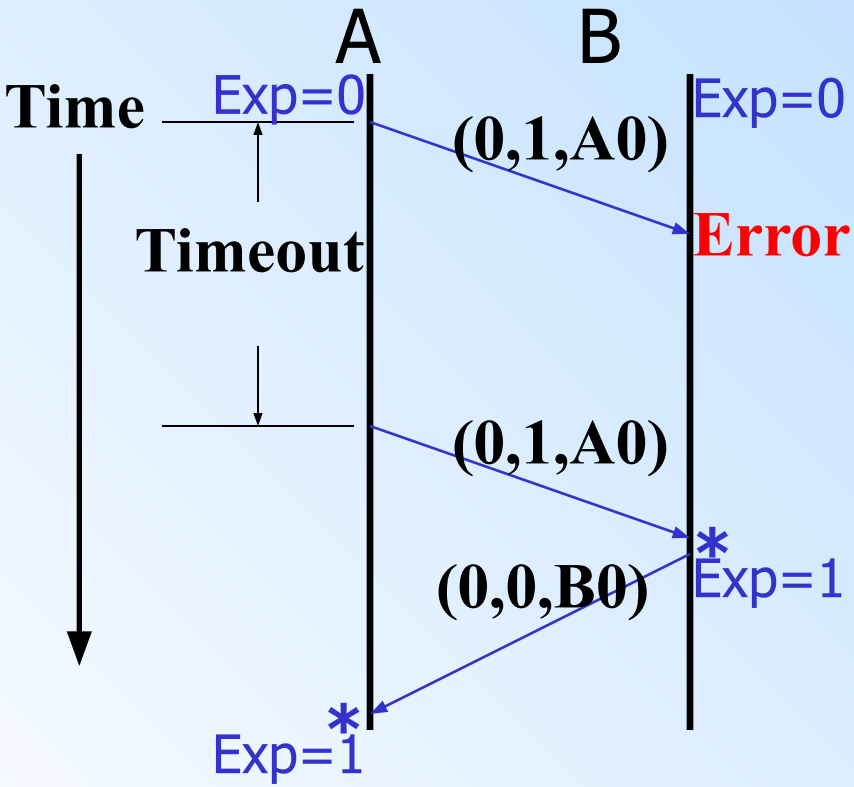
- Case 2: data lost

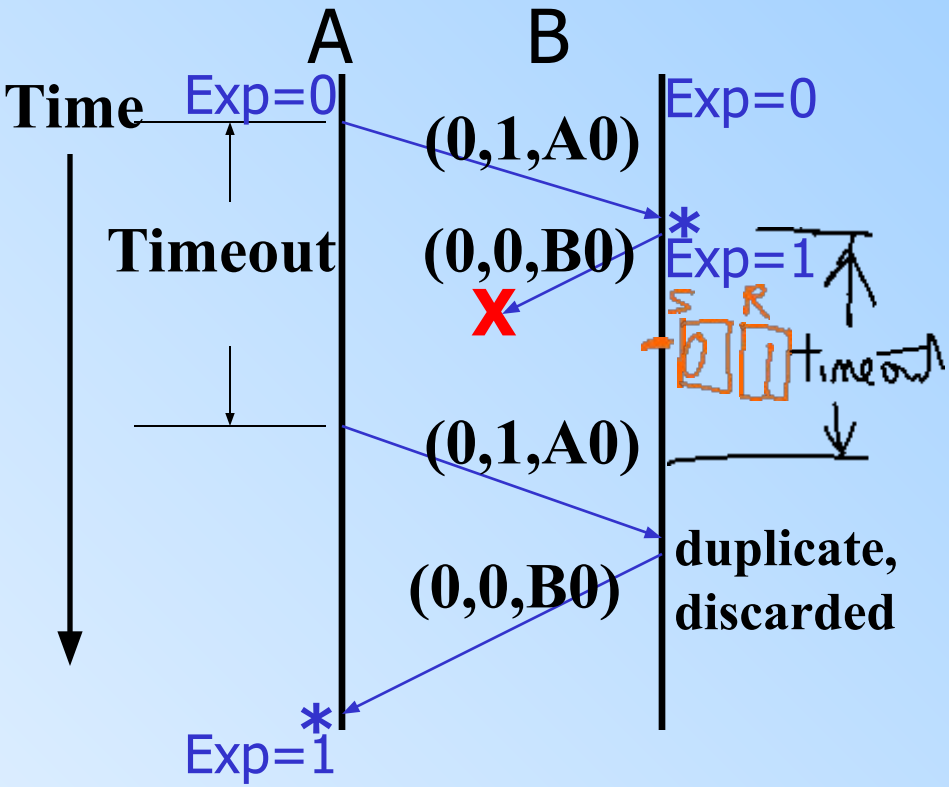


Try to draw the sending windows and receiving windows for A and B!

One Bit Sliding Window Protocol

- Case 3: data error
- Case 4: ack. lost

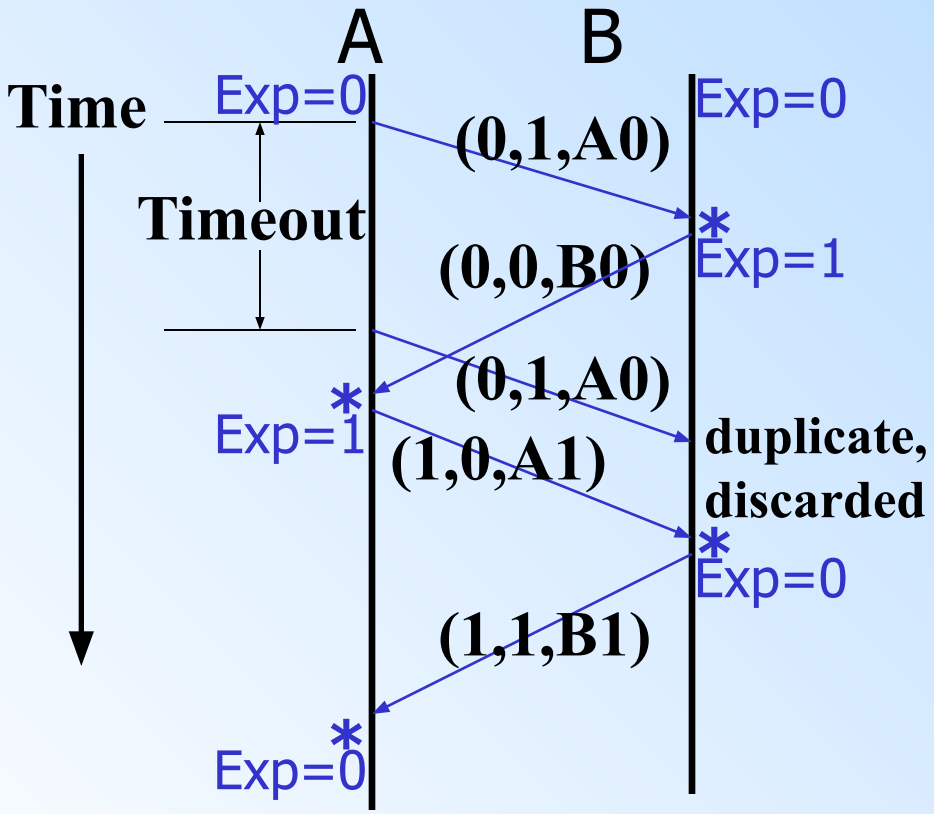




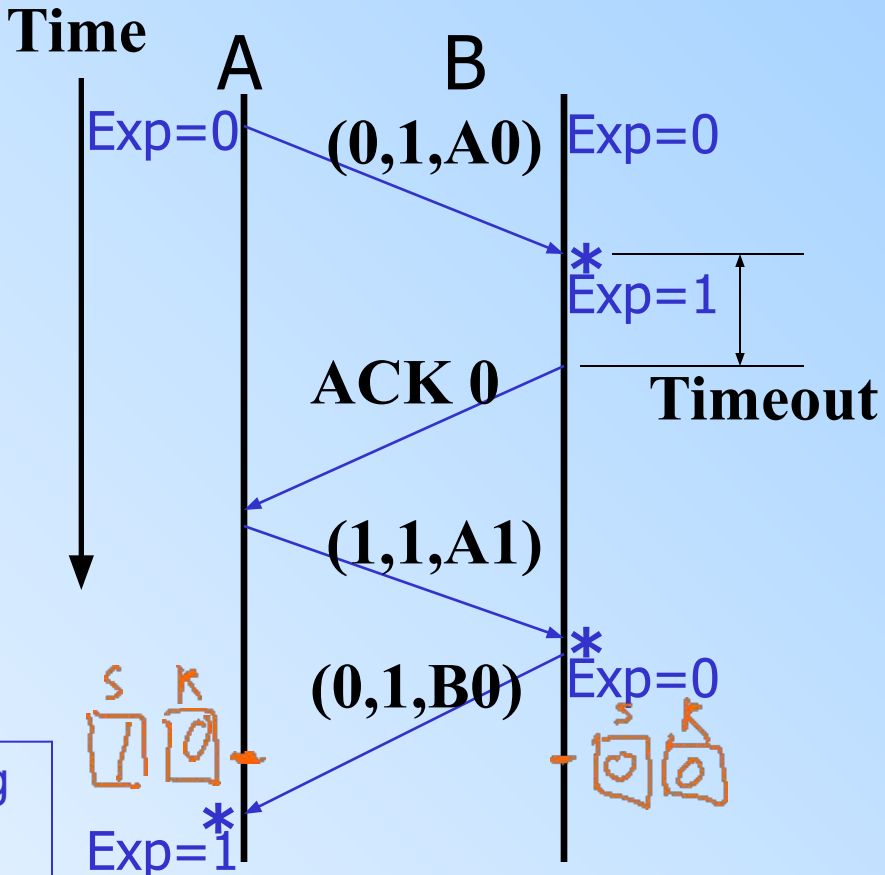
Try to draw the sending windows and receiving windows for A and B!

One Bit Sliding Window Protocol

- Case 5: early timeout



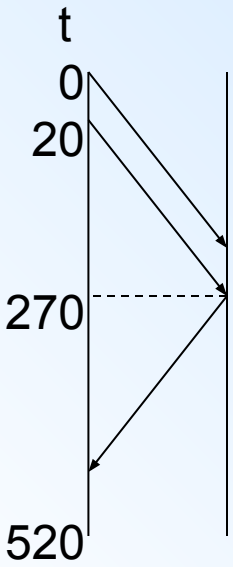
- Case 6: outgoing frame timeout



Try to draw the sending windows and receiving windows for A and B!

Performance of Stop-and-Wait Protocol

- Assumption of previous protocols:
  - Transmission time is negligible
  - False, when transmission time is long
- Example - satellite communication
  - channel capacity: 50 kbps, frame size: 1kb
    round-trip propagation delay: 500 msec
  - Time:t=0 start to send 1st bit in frame
    - t=20 msec frame sent completely
    - t=270 msec frame arrives
    - t=520 msec best case of ack. received
  - Sender blocked 500/520 = 96% of time
    - Bandwidth utilization 20/520 = 4%



Performance of Stop-and-Wait Protocol

- If channel capacity = b, frame size = L, and round-trip propagation delay = R, then bandwidth utilization = _____
- Conclusion:
  - Long transit time + high bandwidth + short frame length ⇒ disaster

Performance of Stop-and-Wait Protocol

- Solution: Pipelining
  - Allowing *w* frames sent before blocking
- In our example, for 100% utilization
  - *w* = __, max window size = __
  - sequence number = __ bits
- Problem: errors
- Solutions
  - Go back *n* protocol (GNP)
  - Selective repeat protocol (SRP)

  Acknowledge *n* means frames *n*,*n*-1,*n*-2,… are acknowledged (i.e., received correctly)

PtP: window protocols

- Go back n protocol
  - Assumptions:
    - Data in both directions: piggybacking of ACK
      - No separate ACK packets
    - Noisy channel
    - Limited stream of data from network layer
  - Protocol: Tanenbaum: fig. 3.17

Go back n protocol

```
/* 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 */
}
```

Go back n protocol

```
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 */
```

Go back n protocol

```
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 */
          }
```

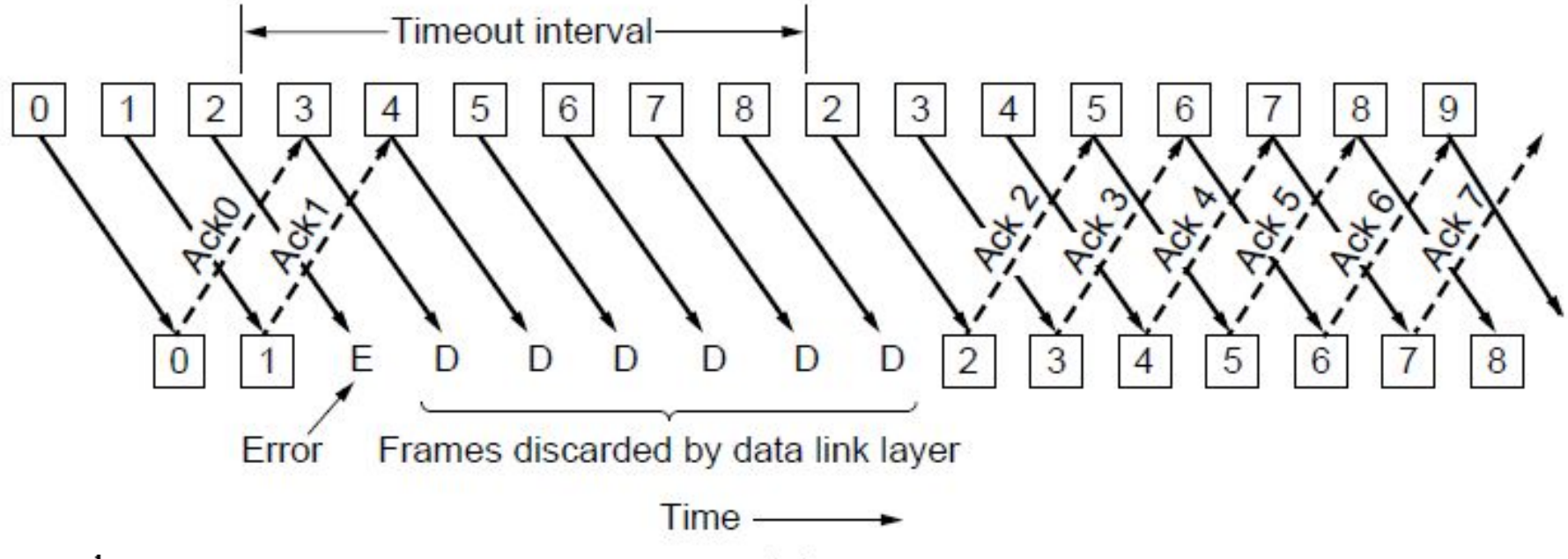
Go back n protocol

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

Go Back *n* Protocol

- Receiver discards all subsequent frames following an error one, and send no acknowledgement for those discarded
- Receiving window size = 1 (i.e., frames must be accepted in the order they were sent)
- Sending window might get full
  - If so, re-transmitting unacknowledged frames
- Wasting a lot of bandwidth if error rate is high

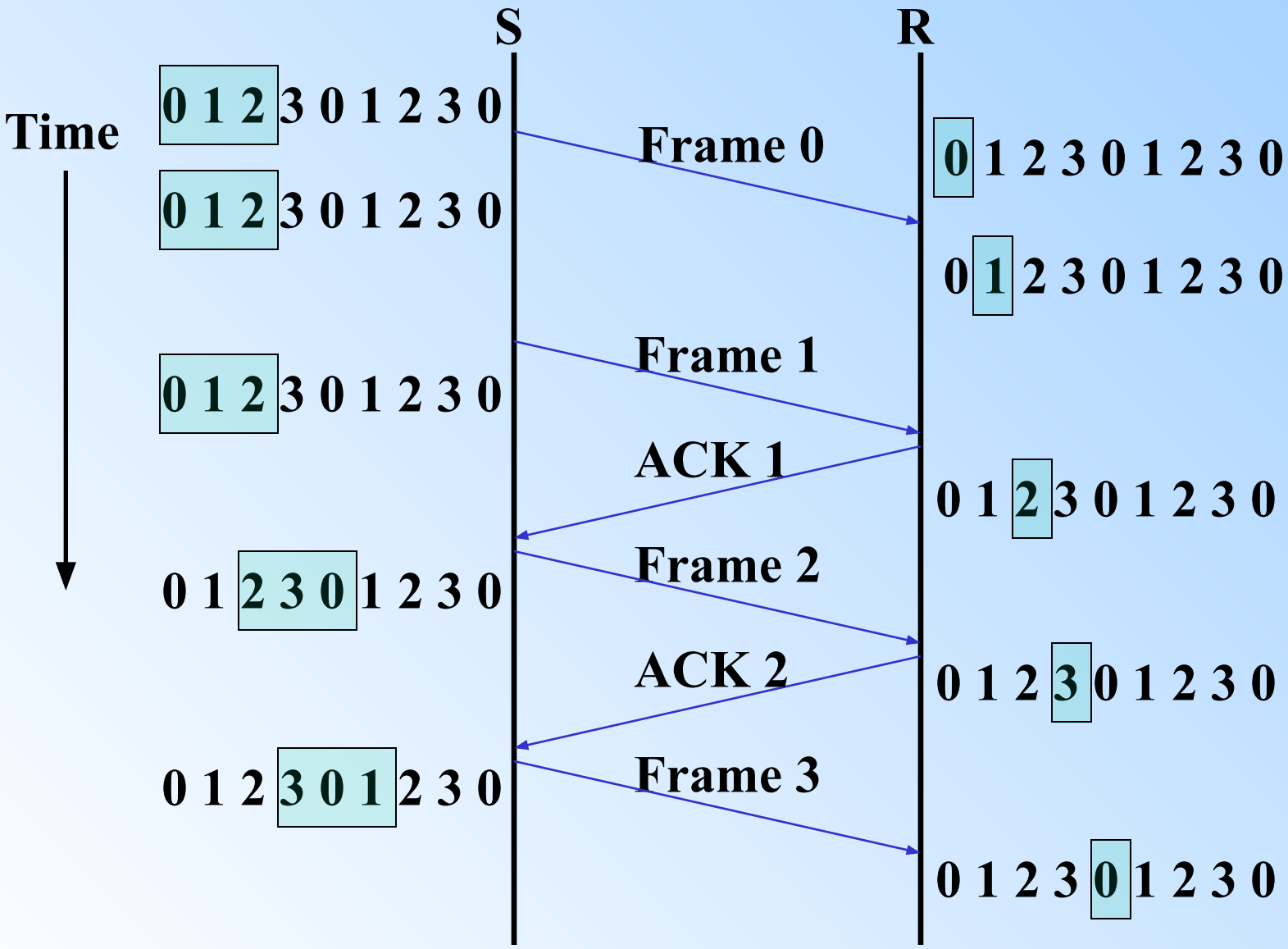
Go Back *n* Protocol



R.W.1

0 1 2 3

Go Back *n* Protocol



Go Back *n* Protocol

- What is the maximum sending window size?
- Maximum sending window size of = MAX_SEQ, not MAX_SEQ+1
  - With *n*-bit sequence number, MAX_SEQ = 2^n – 1, maximum sending window size = 2^n - 1
  - e.g., for 3-bit window, MAX_SEQ = 7, so window size = 7 although max. size could be 8
- Why?

Go Back *n* Protocol - Window Size

- Suppose 3-bit window is used and max sending window size = MAX_SEQ+1 = 8
  - Sender sends frames 0 through 7
  - Piggybacked ack 7 comes back
  - Sender sends anther 8 frames w/ sequence numbers 0 through 7
  - Another piggybacked ack 7 comes back
  - Q: Did all second 8-frames arrive successfully or did all of them get lost?
  - Ack 7 for both cases ⇒ Ambiguous

    ⇒ Max. window size = 7

Go Back *n* Protocol Implementation

- Sender has to buffer unacknowledged frames
- Acknowledge *n* means frames *n*,*n*-1,*n*-2, ... are acknowledged (i.e., received correctly) and those buffers can be released
- One timer for each *outstanding* frame in sending window

PtP: window protocols

- Go back n protocol
  - Maximum size of sender window?
    - Sequence numbers
      - 0 .. MAXSEQ
    - Size? MAXSEQ ? or MAXSEQ + 1?

PtP: window protocols

- Go back n protocol: Maximum size = MAXSEQ + 1?
  - Sender sends $F_0 \ldots F_7$ or $F_0^0 \ldots F_7^7$
  - Receiver sends Ack_7

    and expects F_0 or F_0^8

Case 1: Ack_7 received

- sender transmits $F_0^8 \ldots F_7^{15}$

  or $F_0 \ldots F_7$

Case 2: Ack_7 lost

- sender transmits $F_0^0 \ldots F_7^7$

  or $F_0 \ldots F_7$

No distinction between 2 cases

PtP: window protocols

- Go back n protocol: Maximum size = MAXSEQ?
  - Sender sends $F_0 \ldots F_6$ or $F_0^0 \ldots F_6^6$
  - Receiver sends Ack_6

    and expects F_7 or F_7^7

Case 1: Ack_6 received

- sender transmits F_7^7 F_0^8

  or F_7 F_0

Case 2: Ack_6 lost

- sender transmits F_0^0 F_1^1

  or F_0 F_1

Distinction between 2 cases !!

PtP: window protocols

- Selective repeat protocol
  - Assumptions:
    - Data in both directions: piggybacking of ACK
      - Separate ACK packets (requires ACK timer)
    - Noisy channel
    - Limited stream of data from network layer
    - NACK packets: receiver detected problem
  - Protocol: Tanenbaum: fig. 3.19

Selective repeat protocol

```
/* Protocol 6 (nonsequential receive) accepts frames out of order, but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7                                   /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                              /* no nak has been sent yet */
seq_nr oldest_frame = MAX- SEQ + 1;                 /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) ll ((c < a) && (a <= b)) ll ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                          /* scratch variable */

  s.kind = fk;                                      /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                                 /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                    /* one nak per frame, please */
  to_physical_layer(&s);                            /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                                 /* no need for separate ack frame */
}
```

Selective repeat protocol

```
void protocol6(void)
{
  seq_nr ack_expected;                          /* lower edge of sender's window */
  seq_nr next_frame_to_send;                    /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                        /* lower edge of receiver's window */
  seq_nr too_far;                               /* upper edge of receiver's window + 1 */
  int i;                                        /* index into buffer pool */
  frame r;                                      /* scratch variable */
  packet out_buf[NR_BUFS];                      /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];                       /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];                     /* inbound bit map */
  seq_nr nbuffered;                             /* how many output buffers currently used */
  event_type event;

  enable_network_layer();                       /* initialize */
  ack_expected = 0;                             /* next ack expected on the inbound stream */
  next_frame_to_send = 0;                       /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                                /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
```

Selective repeat protocol

```
while (true) {
  wait_for_event(&event);                          /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                      /* accept, save, and transmit a new frame */
         nbuffered = nbuffered + 1;                /* expand the window */
         from_network_layer(&out  buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
         send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
         inc(next_frame_to_send);                  /* advance upper window edge */
         break;

    case frame_arrival:                            /* a data or control frame has arrived */
         from_physical_layer(&r);                  /* fetch incoming frame from physical layer */
         if (r.kind == data) {
              /* An undamaged frame has arrived. */
              if ((r.seq != frame_expected) && no_nak)
                 send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
              if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)) {
                   /* Frames may be accepted in any order. */
                   arrived[r.seq % NR_BUFS] = true;      /* mark buffer as full */
                   in_buf[r.seq % NR_BUFS] = r.info;     /* insert data into buffer */
                   while (arrived[frame_expected % NR_BUFS]) {
                        /* Pass frames and advance window. */
                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc(frame_expected);     /* advance lower edge of receiver's window */
                        inc(too_far);            /* advance upper edge of receiver's window */
                        start_ack_timer();       /* to see if a separate ack is needed */
                   }
              }
         }
```

Selective repeat protocol

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

            while (between(ack_expected, r.ack, next_frame_to_send)) {
                  nbuffered = nbuffered    1;            /* handle piggybacked ack */
                  stop_timer(ack_expected % NR_BUFS);     /* frame arrived intact */
                  inc(ack_expected);                      /* advance lower edge of sender's window */
            }
            break;
      case cksum_err:
            if (no_nak) send_frame(nak, 0, frame_expected, out_buf);/* damaged frame */
            break;
      case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf);/* we timed out */
            break;
      case ack_timeout:
            send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
   }
   if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
 }
}
```

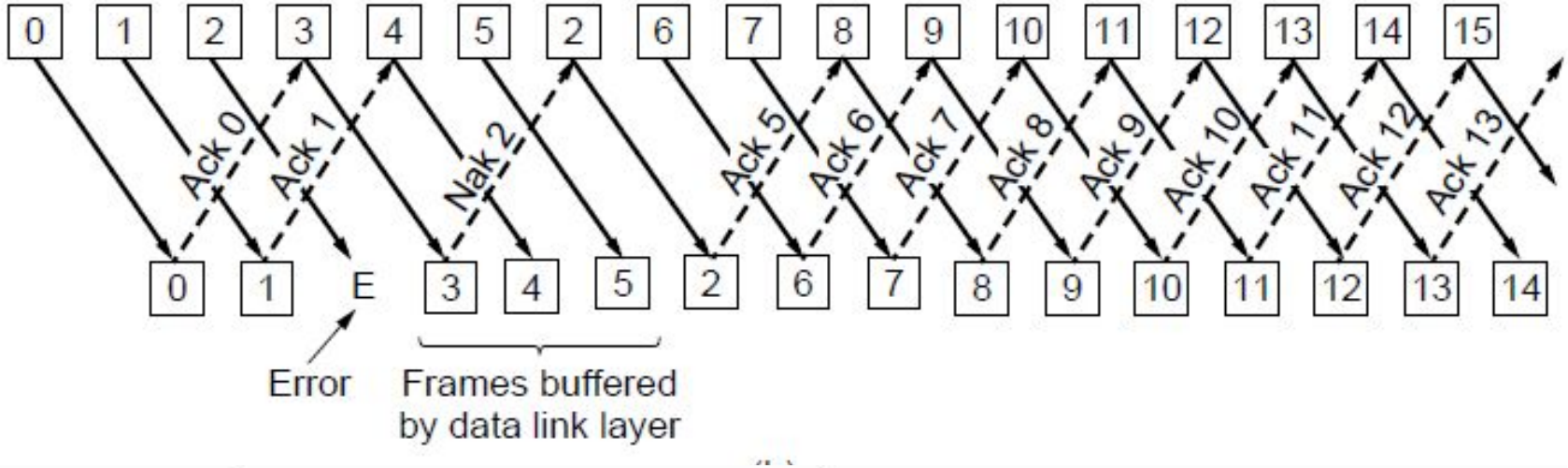
Select Repeat Protocol

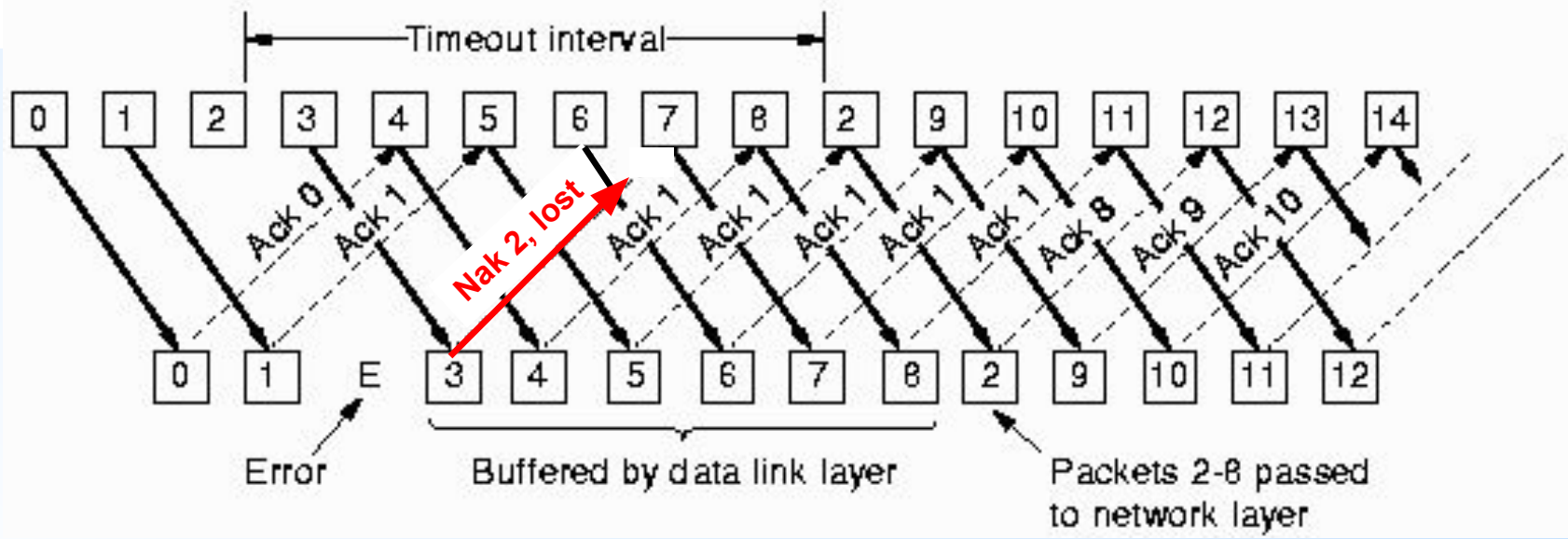
- Receiver stores correct frames following the bad one
- Sender retransmits the bad one after noticing
- Receiver passes data to network layer and acknowledge with the highest number
- Receiving window > 1 (i.e., any frame within the window may be accepted and buffered until all the preceding one passed to the network layer
- Might need large memory

Negative Acknowledgement (NAK)

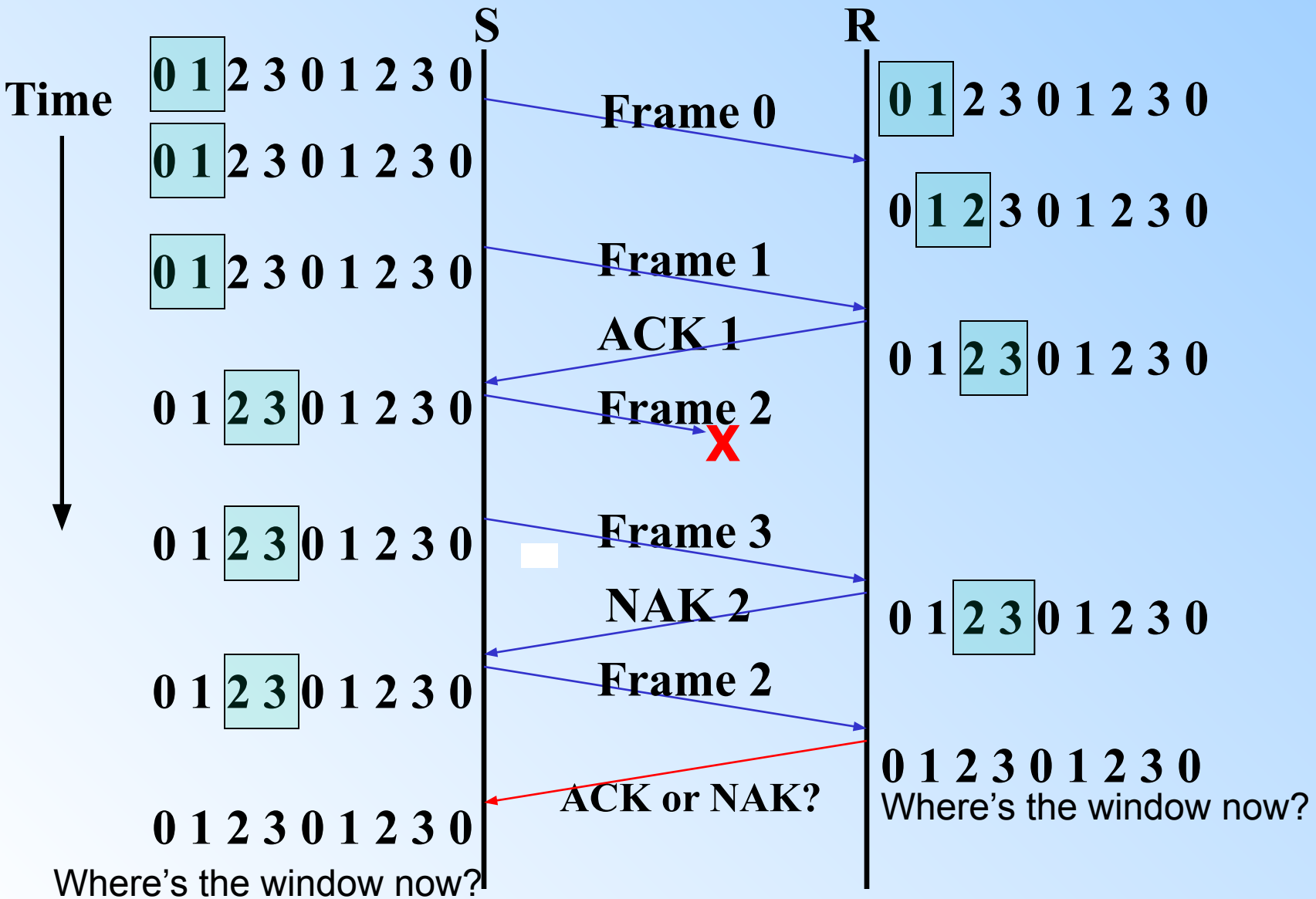
- SRP is often combined with NAK
- When error is *suspected* by receiver, receiver request retransmission of a frame
  - Arrival of a damaged frame
  - Arrival of a frame other than the expected
- Does receiver keep track of NAK?
- What if NAK gets lost?
- To nak, or not to nak: that is the question

Selective Repeat with NAK





Selective Repeat with NAK



Select Repeat Protocol Implementation

- Receiver has a buffer for each sequence number within receiving window
- Each buffer is associated with an "arrived" bit
- Check whether sequence number of an arriving frame within window or not
  - If so, accept and store
- Maximum window size = ? Can it be MAX_SEQ ?

Select Repeat Protocol - Window Size

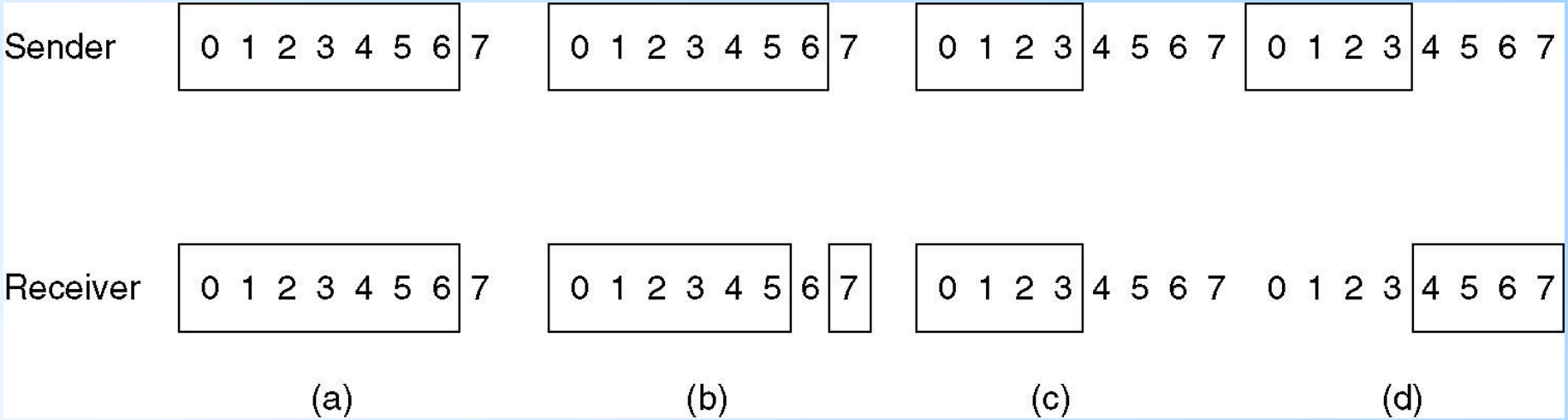
- Suppose 3-bit window is used and window size = MAX_SEQ = 7

sender	receiver
0 1 2 3 4 5 6 sent	0 1 2 3 4 5 6 accepted
	0 through 6 to network layer
	all acknowledgements lost
0 retransmitted	**0** accepted
ack 6 received	
7 sent	7 accepted
	7 and **0** to network layer

Select Repeat Protocol - Window Size

- Problem is caused by new and old windows overlapped
- Solution
  - Window size=(MAX_SEQ+1)/2
  - E.g., if 4-bit window is used, MAX_SEQ = 15
    $\Rightarrow$ window size = (15+1)/2 = 8
- Number of buffers needed
  = window size

Select Repeat Protocol



(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.

Acknowledgement Timer

- Problem
  - If the reverse traffic is light, effect?
  - If there is no reverse traffic, effect?
- Solution
  - Acknowledgement timer:

    If no reverse traffic before timeout

        send separate acknowledgement
  - Essential: ack timeout < data frame timeout Why?

PtP: window protocols

- Selective repeat protocol
  - Maximum size of sender window?
    - Sequence numbers
      - 0 .. MAXSEQ
    - Size? MAXSEQ ?

PtP: window protocols

- Selective repeat protocol: Maximum size = MAXSEQ?
  - Sender sends $F_0 \ldots F_6$ or $F_0^0 \ldots F_6^6$
  - Receiver sends Ack_6

    and expects $F_7 \ldots$ or $F_7^7 \ldots$.

    receive window: 7, 0, 1, …5
  - Ack6 lost
  - Sender retransmits $F_0^0 \ldots F_6^6$

    or $F_0 \ldots F_5 F_6$

Retransmissions accepted in new window

PtP: window protocols

- Selective repeat protocol: Max. size = (MAXSEQ+1)/2?
  - Sender sends $F_0 \ldots F_3$ or $F_0^0 \ldots F_3^3$
  - Receiver sends Ack_3

    and expects $F_4 \ldots$ or $F_4^4 \ldots$

    receive window: 4, 5, 6, 7
  - Ack_3 lost
  - Sender retransmits $F_0^0 \ldots F_3^3$

    or

    $F_0 \ldots F_2 \; F_3$

    No overlap with new window

PtP: window protocols

- Maximum window size?
  - Sender window: S
  - Receiver window:
    - Before receiving packets: R
    - After receiver packets : NR
  - Requirement?

$$S \cap NR \equiv o$$

Overview

- Design issues
- Point-to-point links
- Local area Networks
- Data Link layer Switching
- Window protocols
- Performance
- Protocol verification
- Examples

PtP: performance

- Approach: compute usage of channel
- Notation:

C	Channel capacity
D	Number of data bits in frame
H	Number of bits in header
F	= D + H
I	Propagation delay + Interrupt & service time

PtP: performance

- Stop-and-Wait protocol
  - Simplifications:
    - No piggybacking
    - No errors

Time	Action
0	Start of sending frame
F/C	Last bit sent
F/C + I	Last bit arrived at receiver
F/C + I + A/C	Last bit of ack sent
F/C + I + A/C + I	Last ack bit arrived at sender

  - During this time
    - F + A + 2 I C bits can be sent
    - D useful bits are sent
  - Efficiency:

$$\frac{D}{H + D + A + 2 \times I \times C}$$

PtP: performance

- Sliding window protocol
  - Simplifications:
    - No piggybacking
    - No errors
  - Conditions for continuous transmission

Time	Action
0	Start of sending first frame
F/C	Last bit sent of first frame
W x F/C	Sender gets blocked
F/C + I + (A/C) + I	Last ack bit arrived at sender

  - Critical window size:

$$W_c = 1 + \frac{2 \times I \times C}{F}$$

PtP: performance

- Sliding window protocol
  - Critical window size:

$$W_c = 1 + \frac{2 \times I \times C}{F}$$

  - Efficiency:
    - Large window

$$U = \frac{D}{H + D}$$

    - Small window

$$U = \frac{W \times D}{F + 2 \times I \times C}$$

$$U = \frac{D}{H + D} \times \frac{W}{W_c}$$

PtP: performance

- Sliding window protocol
  - Critical window size: $W_c = 1 + \frac{2 \times I \times C}{F}$
  - Interpretation:
    - C x I number of bits that can be sent in time I
    - C x I / F number of frame that can be sent in time I
      cable length in frames
  - Cable length?
    - 10 Mbps LAN, 1 km
      - C x I = 50 bits cable length << 1
    - 64 Kbps transcontinental cable, 3000 km
      - C x I = 960 bits cable length ≈ 1
    - 64 Kbps satellite channel, I ≈ 270 msec
      - C x I = 17280 bits cable length >> 1

PtP: performance

- Sliding window protocol
  - Critical window size:

$$W_c = 1 + \frac{2 \times I \times C}{F}$$

  - Figure overhead projector!

Overview

- Design issues
- Point-to-point links
- Local area Networks
- Data Link layer Switching
- Window protocols
- Performance
- Protocol verification
- Examples
  - HDLC
  - SLIP
  - PPP

Protocol verification

- Protocols + implementations are complex!
- Research to find mathematical techniques for
  - specification
  - verification

  of protocols
- Intro to 2 techniques:
  - Finite state machine models
  - Petri net models

Verification: finite state machine

- Approach:
  - Model (relevant) states of protocol entities & channels
  - Define all transitions between states
- Result: graph with
  - nodes: all states of global system (= protocol entities + channels)
  - edges: all transitions
- Verification:
  - Reachability analysis
  - Deadlock detection
  - ….

Verification: finite state machine

- Example: simplex protocol for a noisy channel (protocol 3 – fig 3.12)
- States:
  - Sender:
    - 0: packet with seqnr 0 is sent out
    - 1: packet with seqnr 1 is sent out
  - Receiver:
    - 0: receiver expects packet with seqnr 0
    - 1: receiver expects packet with seqnr 1
  - Channel:
    - 0 (1): packet with seqnr 0 (1) on channel
    - A: Ack packet on channel
    - - : channel empty

SRC
000
01A

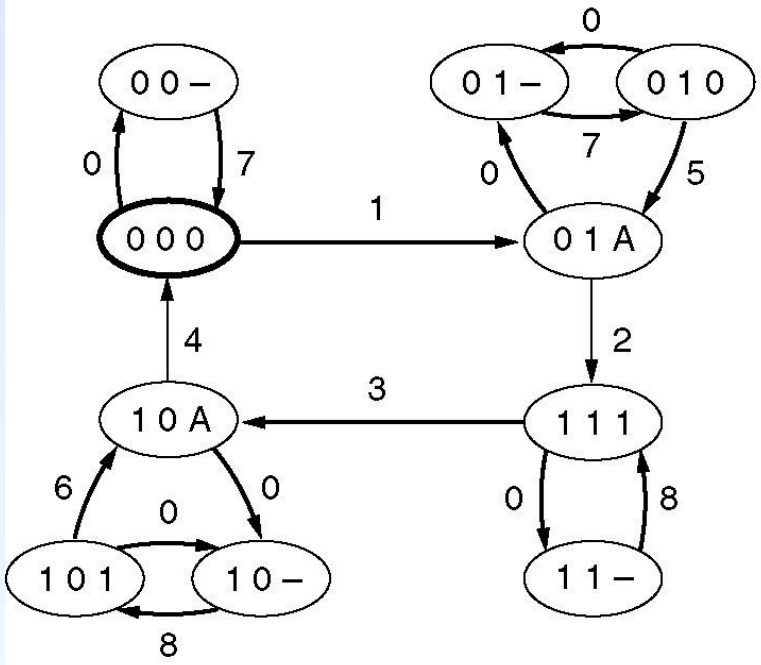
Verification: finite state machine

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

Verification: finite state machine

```
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 */
       }
  }
}
```

Verification: finite state machine



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

(b)

Modelling correct?

NO！！

Verification: finite state machine

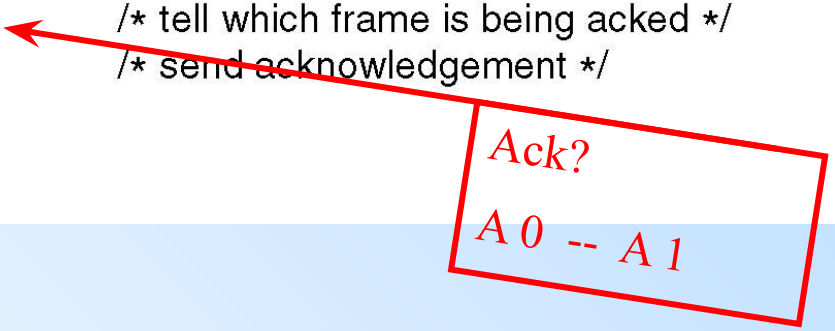
```
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */

#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 */
          }
      }
  }
}
```

Ack?

A 0 -- A 1

Verification: finite state machine

```
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 */
      }
 }
}
```

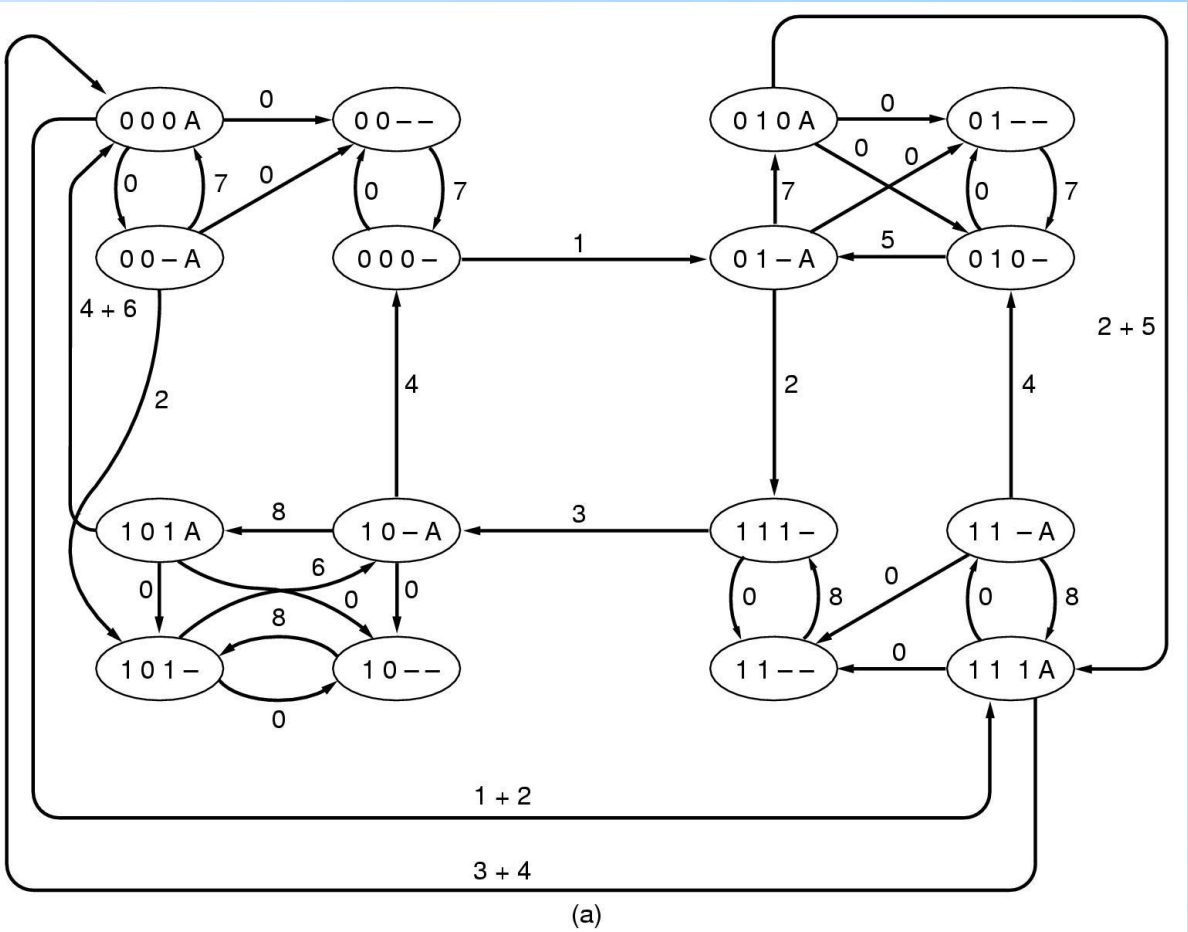


Verification: finite state machine

- Protocol with unnumbered ACKs?
  - wrong!
- Result from analysis
  - protocol ok!!
- Error? wrong modelling
  - half duplex <> full duplex channel
  - 2 channels iso 1 channel
    - Channel: s □ r:
      - Packet 0
      - Packet 1
      - - (empty)
    - Channel: r □ s
      - Ack
      - - (empty)

Verification: finite state machine

- States:
  - Sender
  - Receiver
  - Channel: s □ r
  - Channel: r □ s



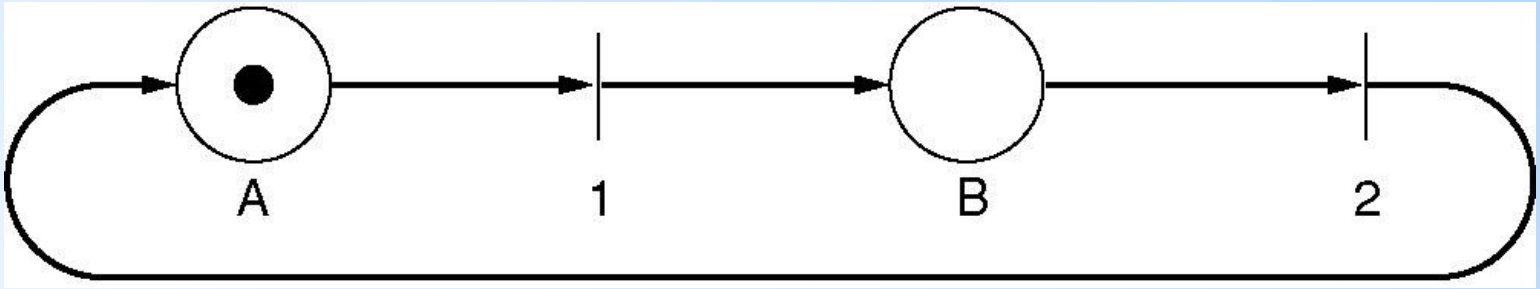
(0 0 0 –), (0 1 – A), (0 1 0 A), (1 1 1 A), (1 1 – A), (0 1 0 –), (0 1 – A), (1 1 1 –)

Transitions	1	7	2+5	0	4	5	2

Verification: petri net models

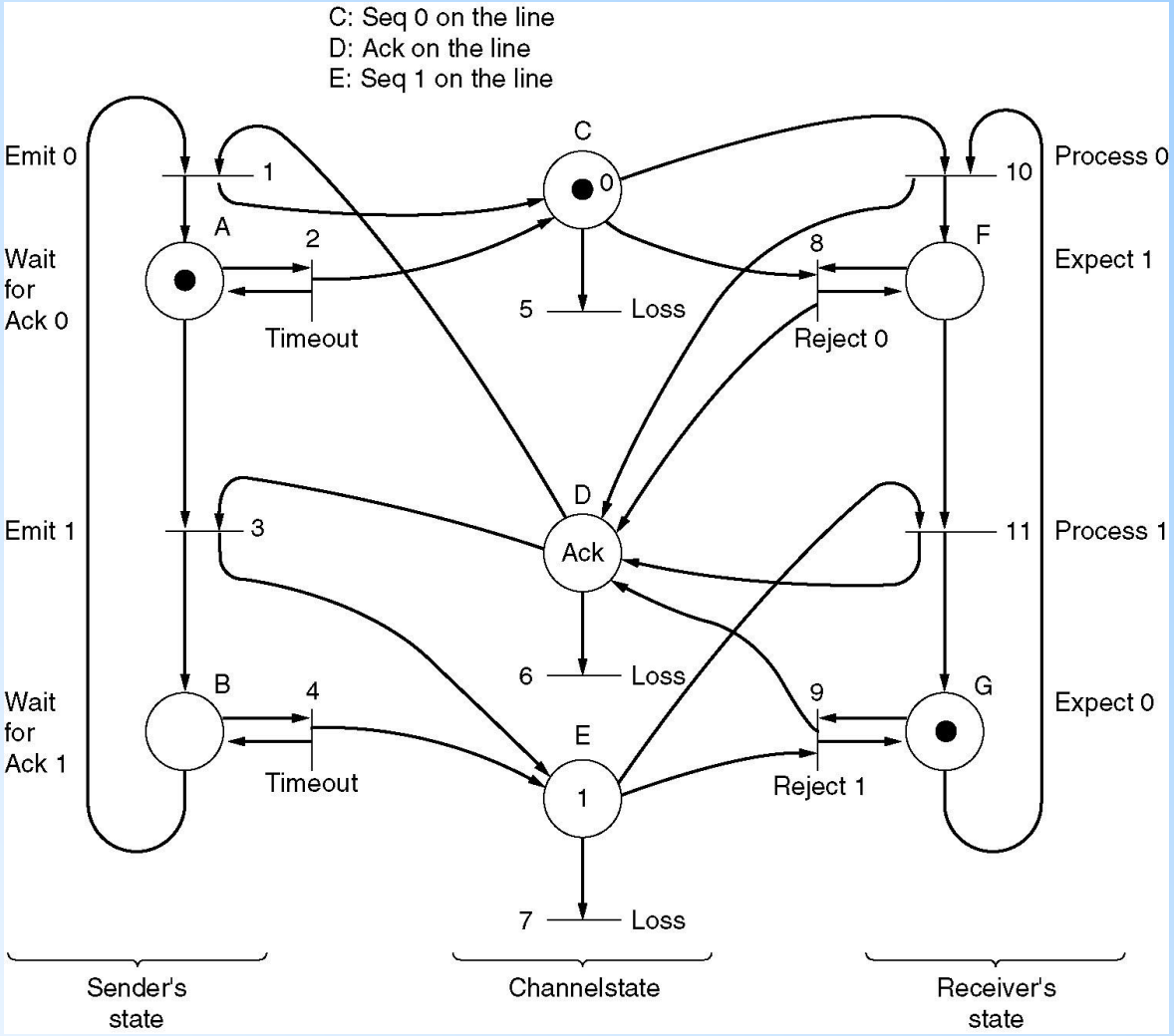
- Basic elements:
  - Places: represent system states
  - Transitions: represent actions in system
  - Arcs: relate states to actions
  - Tokens: current state
- Transitions:
  - Input places
  - Output places
- Transition enabled
  - At least one token in each input place
- Firing of transition
  - If enabled
  - 1 token removed from each input place
  - 1 token added in each output place

Verification: petri net models



Verification: petri net models

- Model protocol 3: fig 3.12
- Algebraic representation
- Analysis:
  - Reachability
  - Deadlock
  - …



Overview

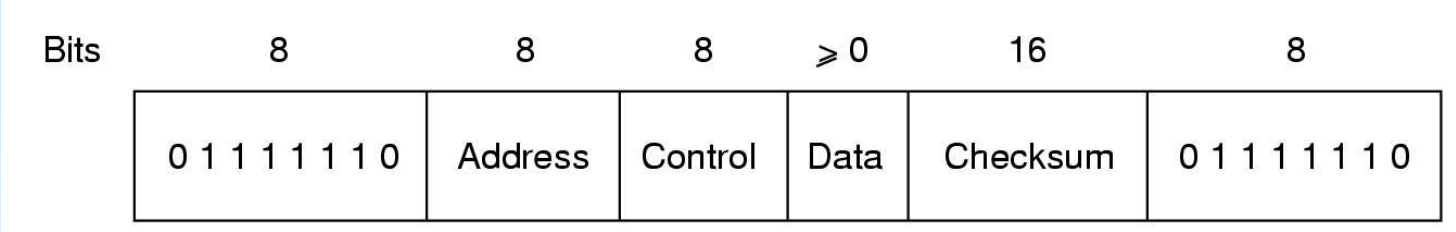
- Design issues
- Point-to-point links
- Local area Networks
- Data Link layer Switching
- Window protocols
- Performance
- Protocol verification
- Examples
  - HDLC
  - SLIP
  - PPP

PtP: examples

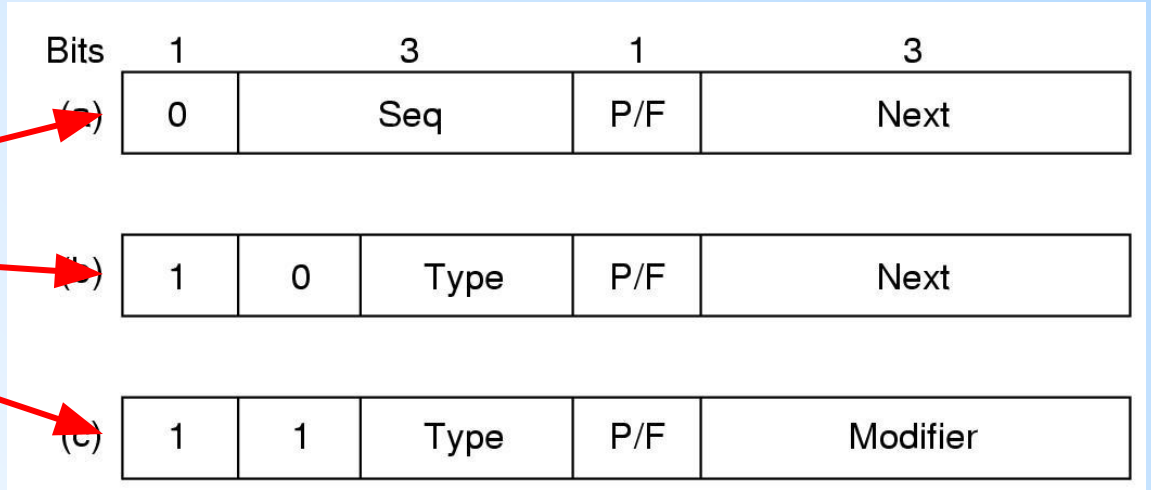
- HDLC: High Level Data Link Control
  - Origin
    - SDLC: Synchronous Data Link Control
      - from IBM's SNA
    - ADDCP: Advanced Data Communications Control Procedure
      - = SDLC as modified by ANSI
    - HDLC:
      - = SDLC as modified by ISO
    - LAP: Link Access Procedure
      - = HDLC as modified by CCITT for X25
    - LAPB:
      - More compatible with later version of HDLC
  - Standards?

PtP: examples

- HDLC: High Level Data Link Control
  - Frame structure:



- Address
  - for multi-point line
  - distinguish responses from redirected frames
- Control field



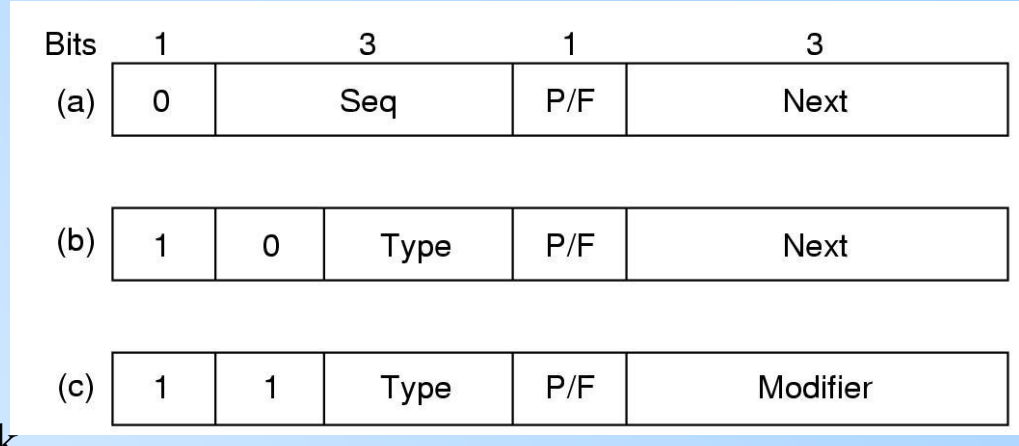
Information frame

Supervisory frames

Unnumbered frames

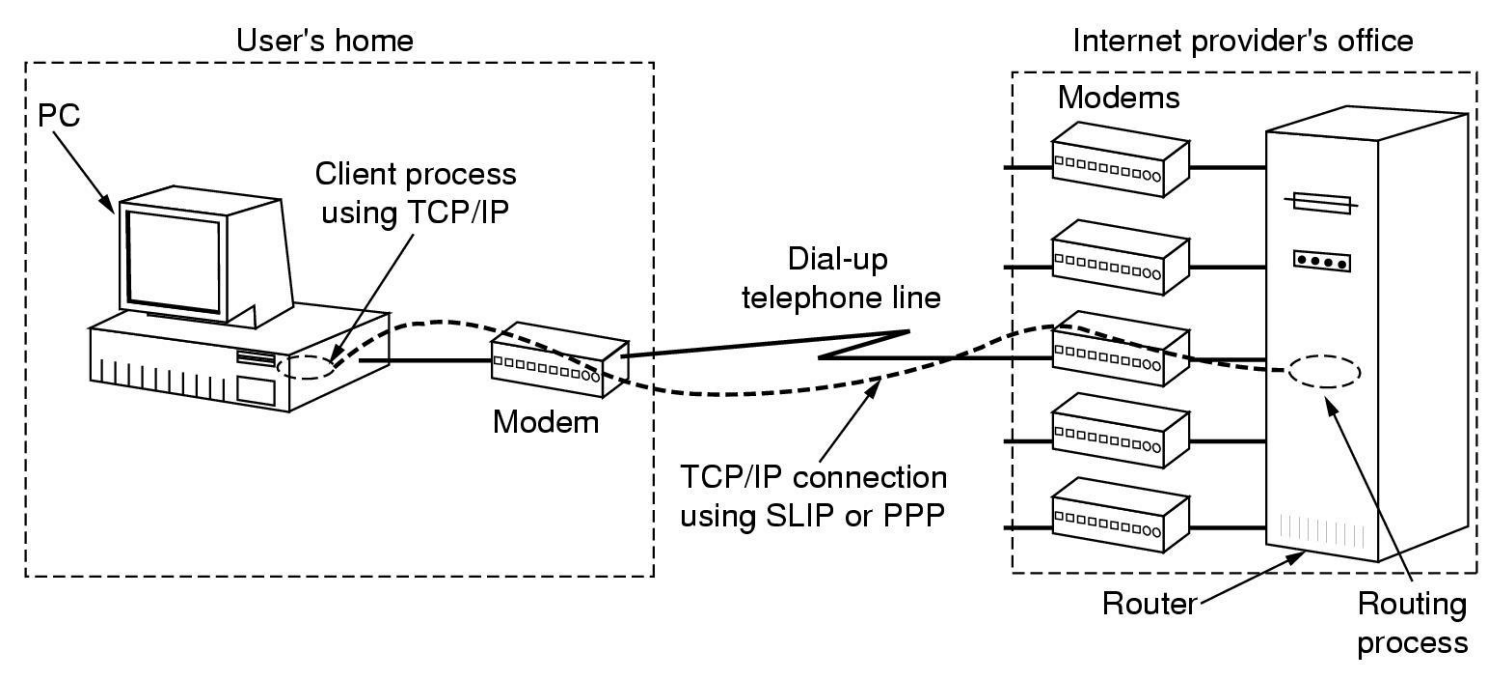
PtP: examples

- HDLC
  - Kind of frames:
    - Information frame
    - Supervisory frames:
      - Reject = nack
      - Receive ready = ack
      - Receive not ready = ack + stop sending
      - Selective reject = resend 1 frame
    - Unnumbered frames
      - Initialisation
      - Polling
      - Status reporting
      - ….



PtP: examples

- Data Link layer in the Internet
  - Context:
    - Dial-up
    - Leased lines between routers



PtP: examples

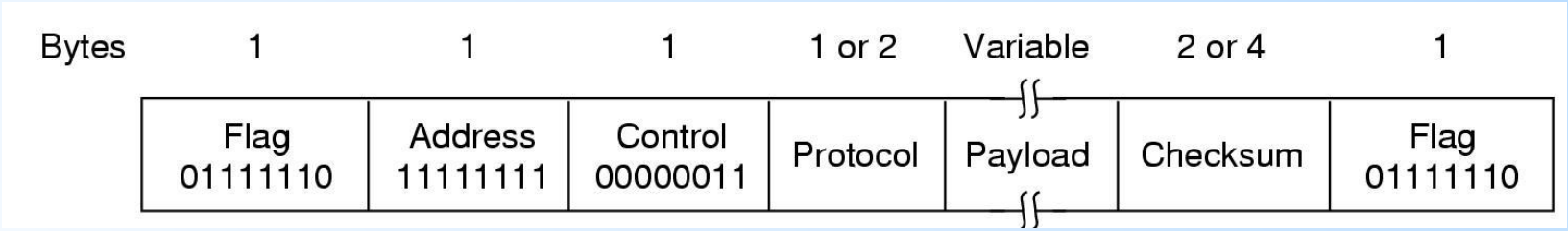
- SLIP – Serial Line IP
  - RFC 1055 (1984)
  - Frames:
    - Raw IP packets + flag byte (0xC0) at end and/or start of packet
    - Character stuffing
    - Optimisation: compression of TCP & IP header
  - Disadvantages:
    - No error detection: left to higher layers
    - No dynamic assignment of IP addresses
    - No support for authentication
    - Not an approved Internet standard

PtP: examples

- PPP – Point-to-Point Protocol
  - PPP provides
    - Framing + error detection
    - Link control protocol (LCP) for bringing up lines, test lines, negotiate options, bringing down lines
    - Network control Protocol (NCP) to negotiate options independent of the network protocol used
  - Scenario
    - PC calls router of provider via modem
    - Router's modem answers phone physical connection
    - Series of LCP packets to select PPP parameters
    - Series of NCP packets to e.g. get IP address
    - PC = Internet host!

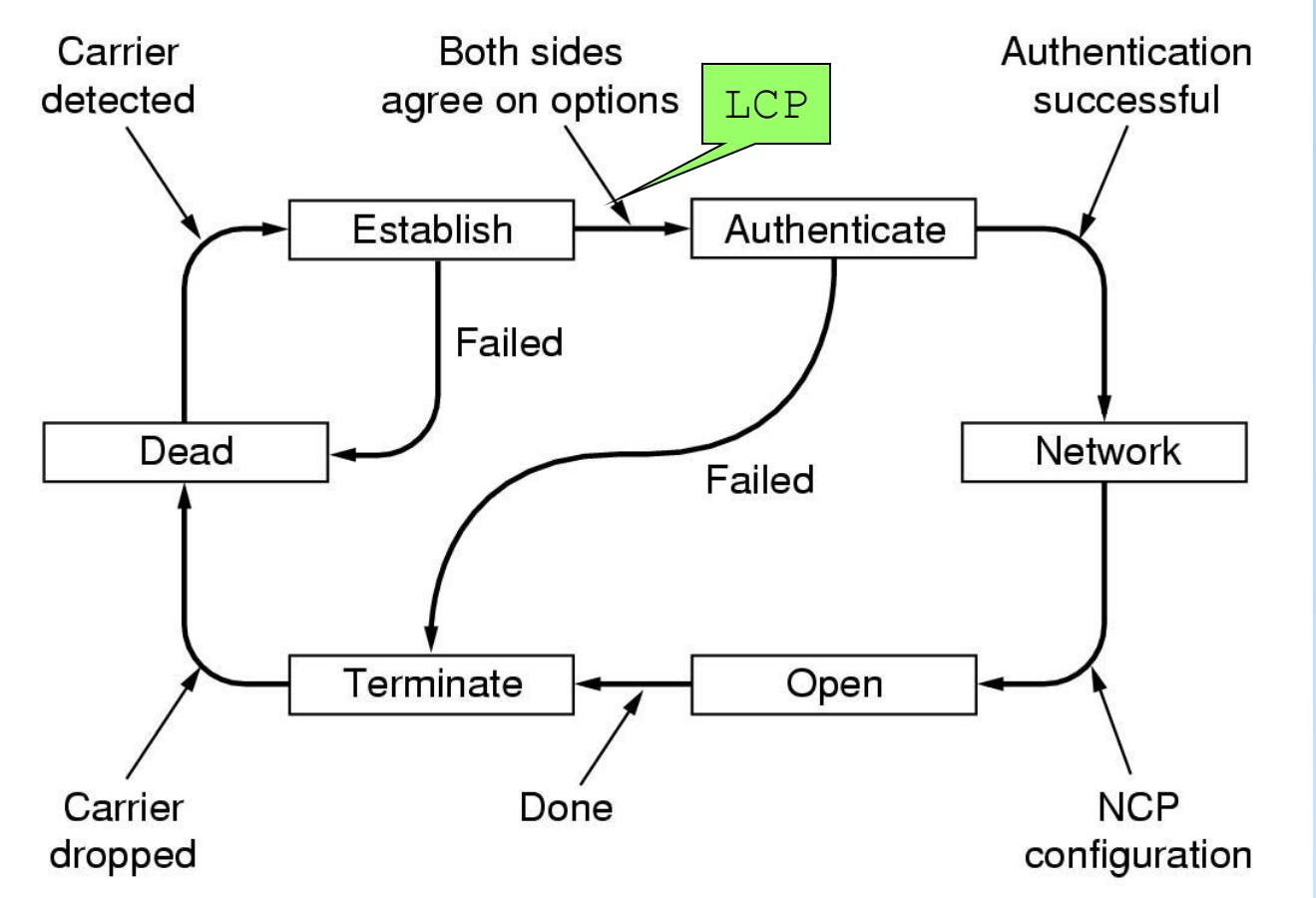
PtP: examples

- PPP – Point-to-Point Protocol
  - PPP frame format: ≈ HDLC but character oriented
    - Address: fixed value, can be omitted
    - Control: default = unnumbered frame
    - Protocol:
      - Kind of packet in payload
      - Code starts ‘0’: network protocols: IP, OSI CLNP,…
      - Code starts ‘1’: negotiate other protocol: LCP, NCP



PtP: examples

- PPP – bring line up or down – simplified diagram



PtP: examples

- PPP: LCP packet types (Initiator, Responder)

Name	Direction	Description
Configure-request	I → R	List of proposed options and values
Configure-ack	I ← R	All options are accepted
Configure-nack	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)

PtP: examples

- PPP
  - LCP options
    - Maximum payload size for data frames
    - Enabling authentication
    - Choosing protocol to use
    - Selecting various header compression options
  - NCP options
    - For IP: dynamic address assignment