

# Computer Networks

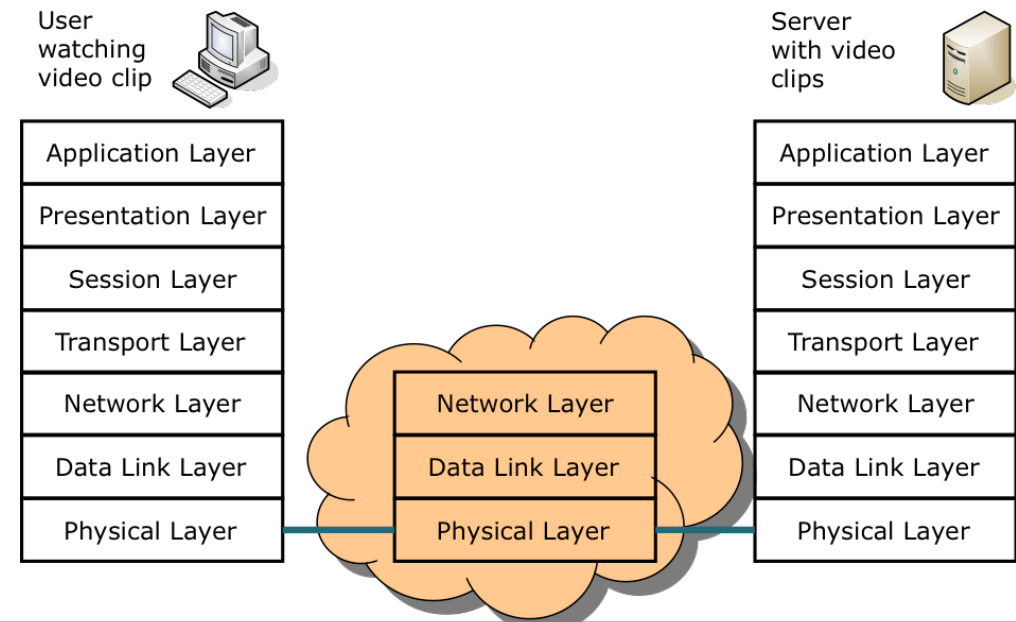
## Chapter 4: Data Link Layer

**Prof. Dr. Mesut Güneş**

Communication and Networked Systems (ComSys)

OVGU Magdeburg

[www.comsys.ovgu.de](http://www.comsys.ovgu.de) | [mesut.guenes@ovgu.de](mailto:mesut.guenes@ovgu.de)



# Contents

---

- **Design Issues**
- **Error Detection and Correction**
- **Elementary Data Link Protocols**
- **Sliding Window Protocols**
- **High Level Data Link Control (HDLC)**
- **Point-to-Point Protocol (PPP)**
- **Protocol Verification**

---

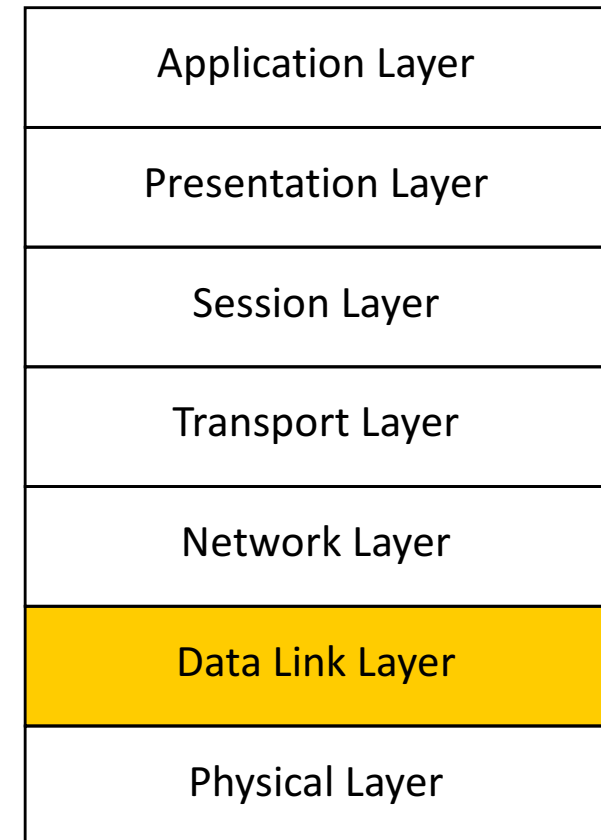
# Design Issues

# Design Issues

---

- The data link layer has a number of specific functions.
- **For this it ...**
  - provides a well-defined service interface to the network layer
  - deals with transmission errors
  - regulates ...
    - the flow of data
    - the access to the medium
    - that a slow receiver is not swamped by a fast sender

**OSI Reference Model**



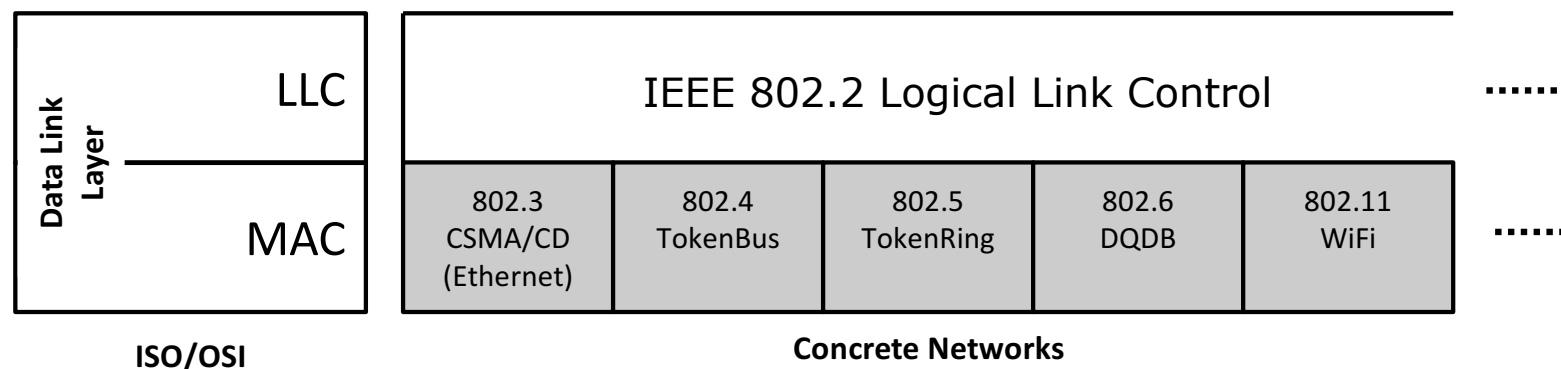
# Layer 2: Division into two parts

## ▪ Logical Link Control (LLC) → Layer 2b

- Organization of the data to be sent into **frames**
- Guarantee (if possible) an error free transmission between **neighboring** nodes by ...
  - Detection (and recovery) of transfer errors
  - Flow Control (avoidance of overloading the receiver)
  - Buffer Management

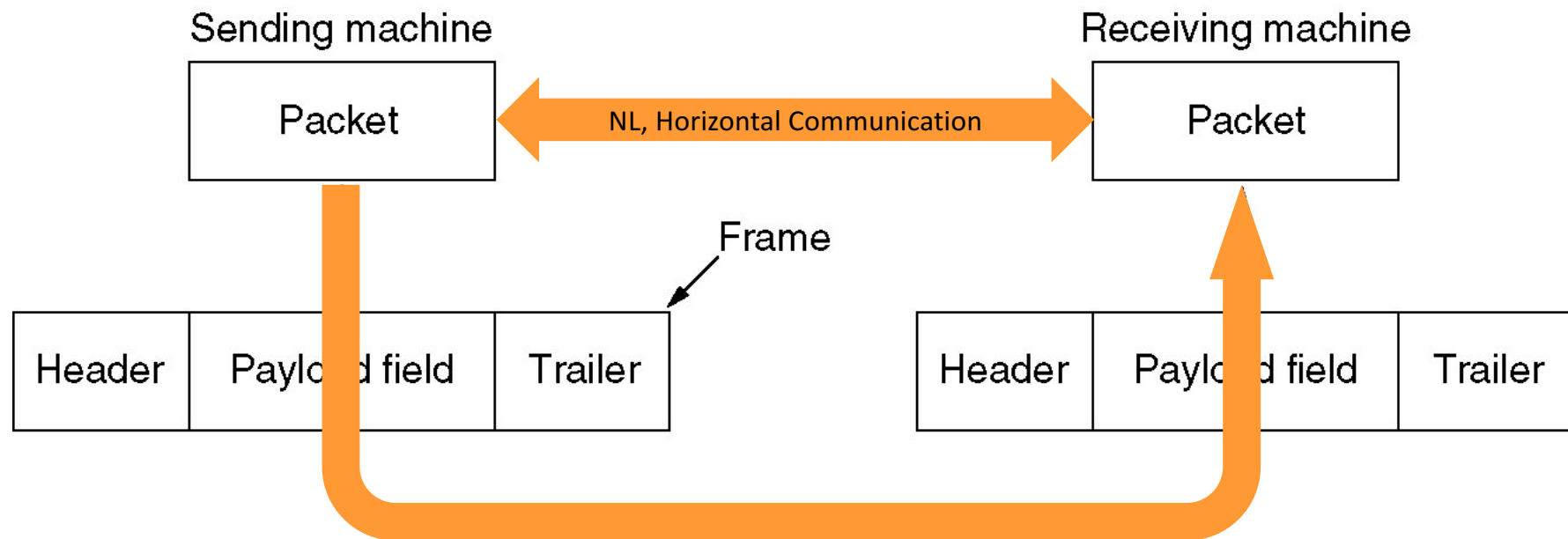
## ▪ Medium Access Control (MAC) → Layer 2a

- Access control to the communication channel in broadcast networks



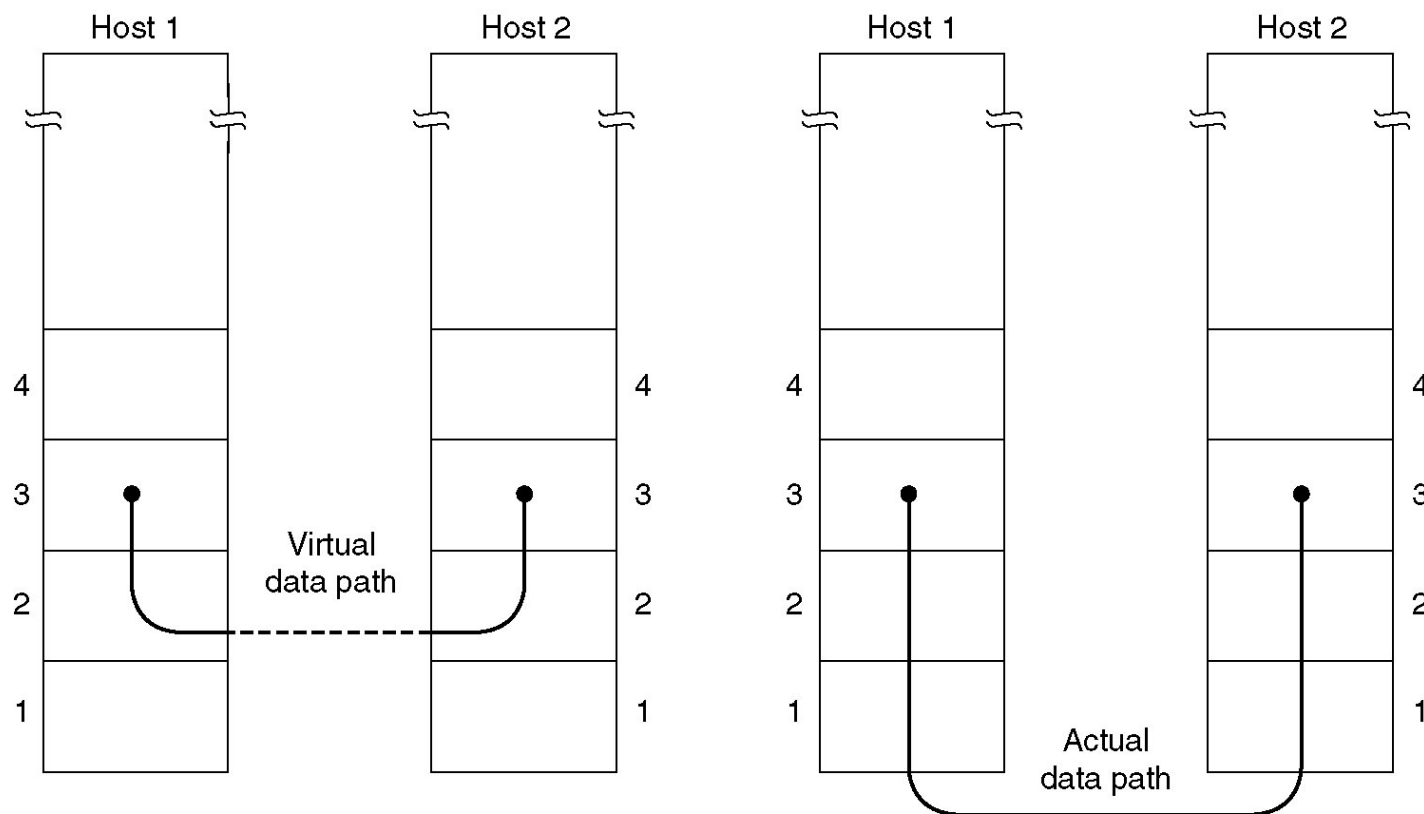
# Services Provided to the Network Layer

- The function of the data link layer is to provide services to the network layer
  - Main service is the transport of data from the network layer of the source to the network layer of the destination machine



# Services Provided to the Network Layer

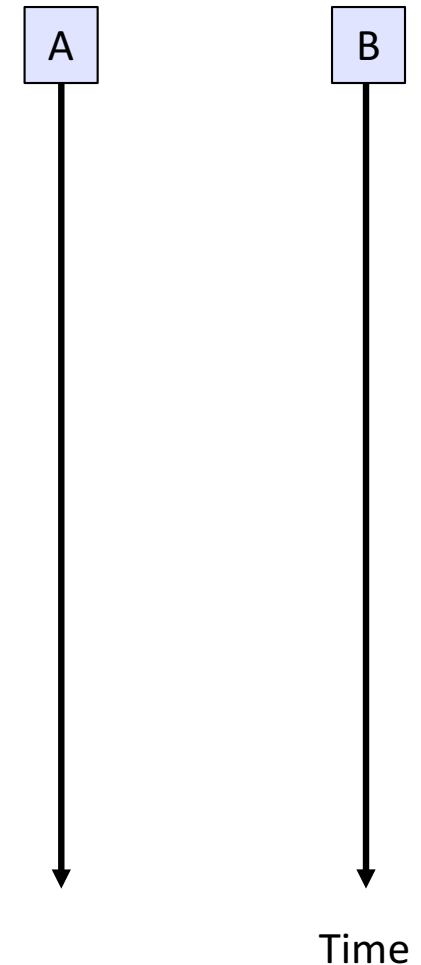
- Communication of two processes on the network layer
  - Virtual data path
  - Actual data path



# Services Provided to the Network Layer

---

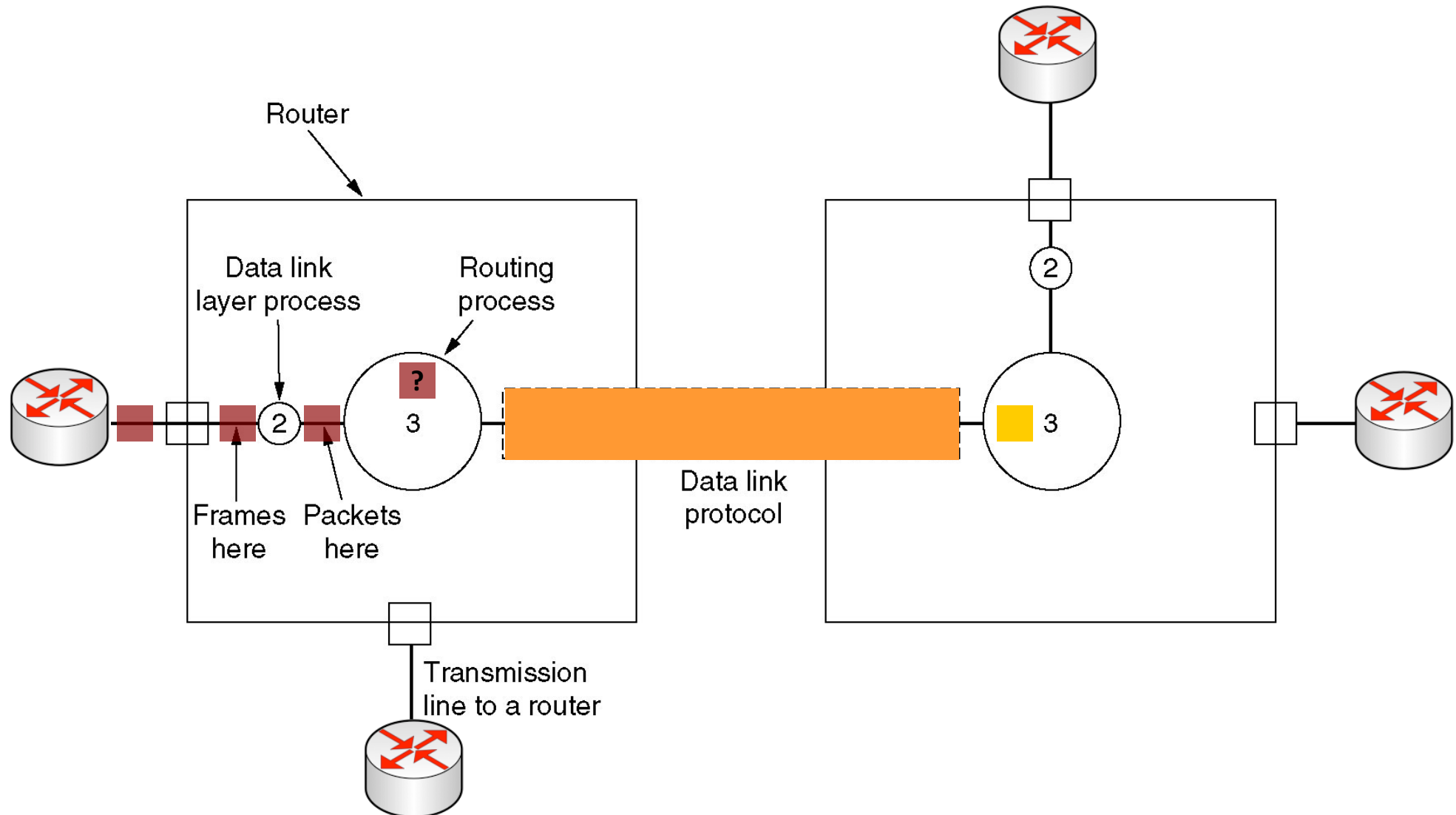
- **Unacknowledged connectionless service**
  - No logical connection beforehand
  - Source sends independent frames
  - Destination does not acknowledge frames
- **Acknowledged connectionless service**
  - No logical connection beforehand
  - Destination acknowledges frames
- **Acknowledged connection-oriented service**
  - Logical connection beforehand
  - Each sent frame is numbered
  - Each sent frame is received once and in the correct order





# Services Provided to the Network Layer

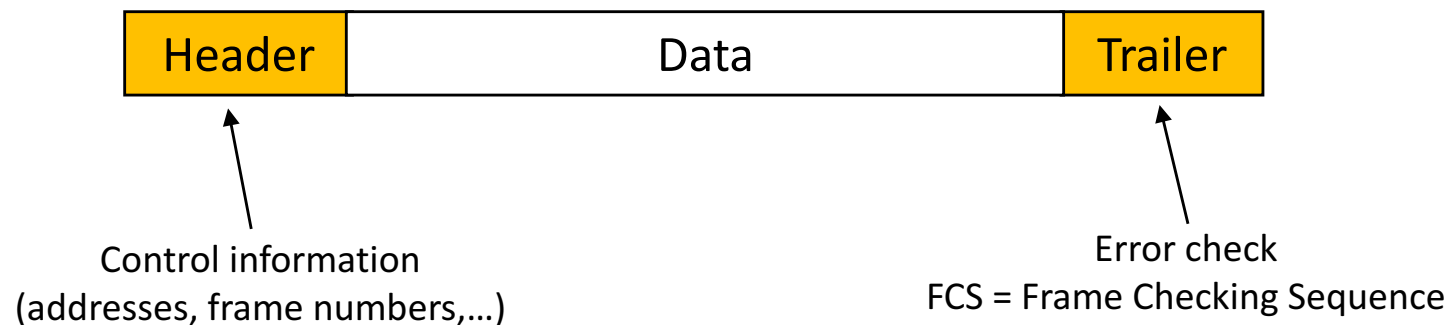
- Location of the data link protocol.



# LLC: Frame Construction

---

- Organization of a message into uniform units (simpler to transmit)
- Well-defined interface to the upper layer (layer 3)
- Marking of a unit:



Mark the frame by ...

- Start and end flags
- Start flag and length
- Code injuries

# LLC: Frame Construction

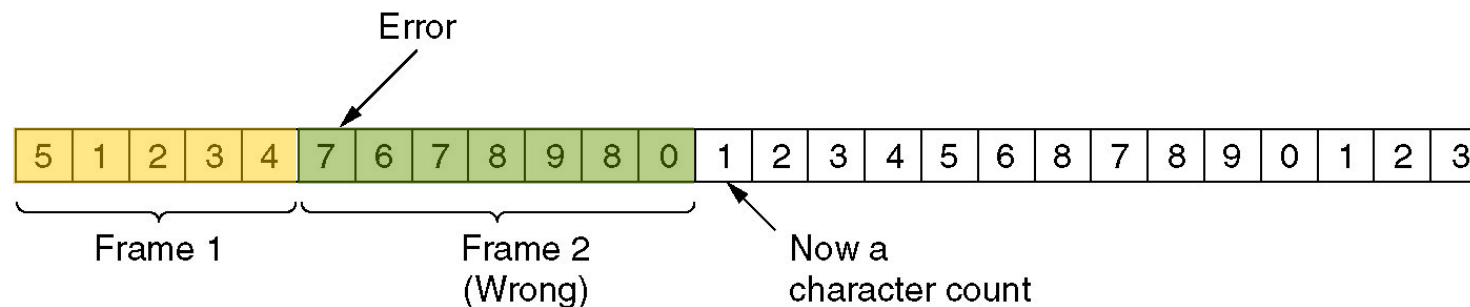
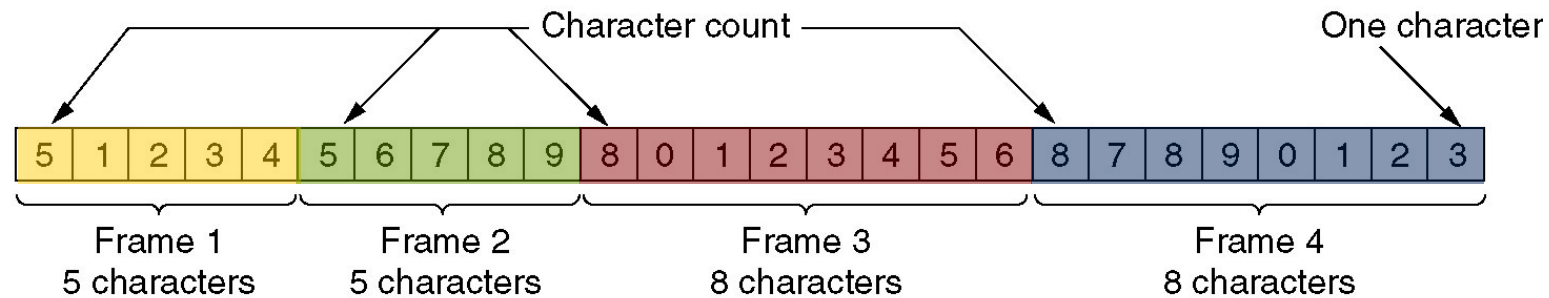
---

- **Marking the start and end of a frame**
  - Character count
  - Flag bytes with byte stuffing
  - Start and end flags with bit stuffing
  - Physical layer coding violations



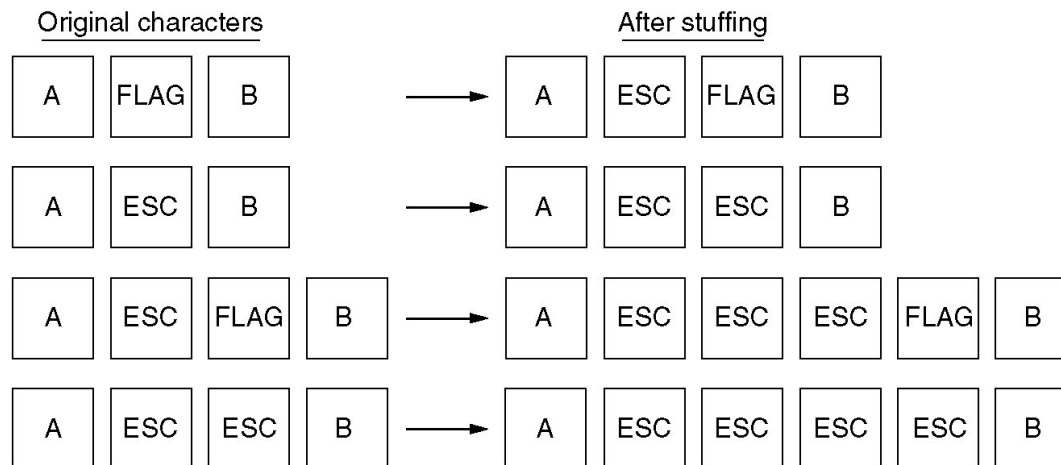
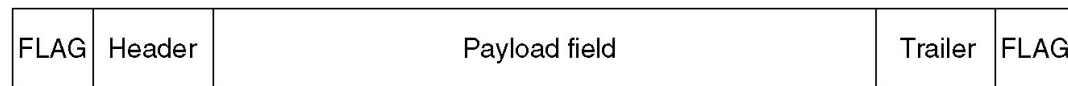
# Framing: Character Count

- Specify the number of characters in the frame



# Framing: Flag Bytes with Character Stuffing

- Start and end of a frame is represented by a special flag byte
- Problem: What happens if the flag byte occurs in the data?
- Byte stuffing/character stuffing
  - A special escape byte (ESC) is inserted (stuffed) by the sender and removed by the receiver
  - If the escape byte occurs in the data, then it is also stuffed




# Framing: Flag Bytes with Bit Stuffing

---

- **Character stuffing is bound to the character set**
  - Disadvantage!
- **General form: Bit stuffing**
  - Frames begin and end with a special pattern: 01111110
  - Sender inserts after five 1s a 0-bit, i.e.,  
011111x ➡ 0111110x and the receiver removes it

Original data:            0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

Transmitted data:      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



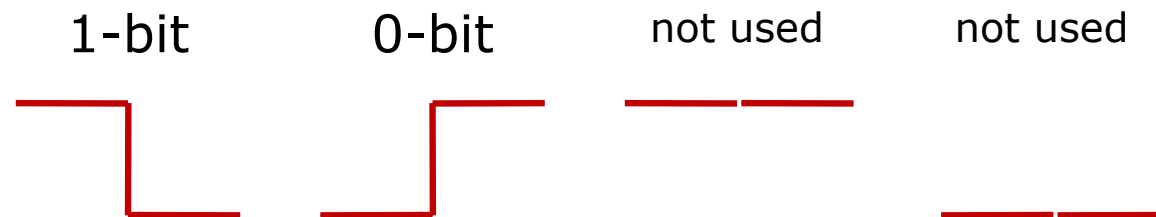
Stuffed bits

After destuffing:        0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

# Framing: Physical Layer Coding Violations

---

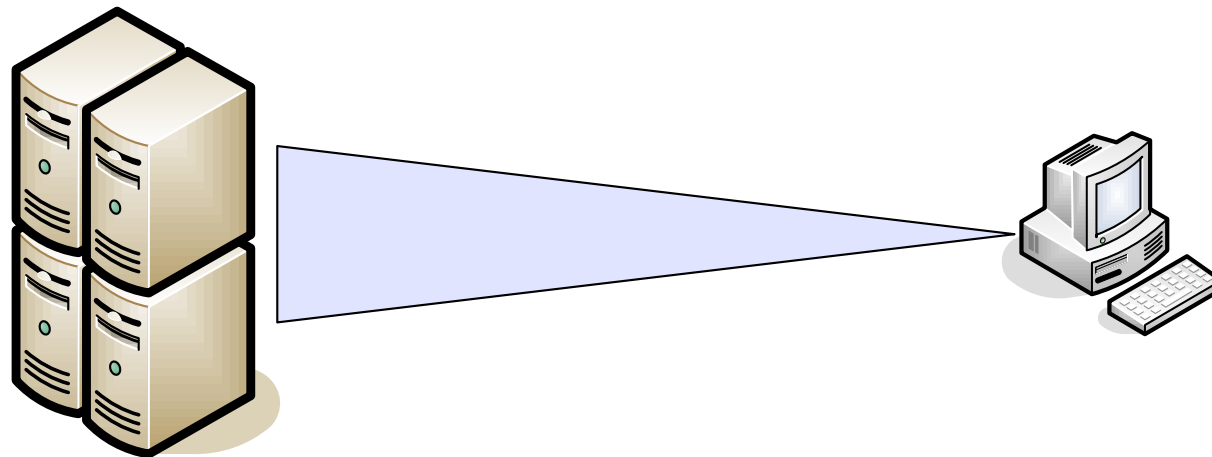
- **Only applicable if physical layer coding has some redundancy**
  - 1 bit data is encoded in 2 physical bits
  - 1-bit coded as high-low, 0-bit coded as low-high  
high-high and low-low are not used



# Framing: Flow Control

---

- **Scenario: Fast sender and slow receiver**
  - How to prevent a slow receiver swamped?
- **Two approaches**
  - Feedback-based flow control
    - Receiver sends back information to the sender giving permission to send more data
  - Rate-based flow control
    - Protocol limits the rate of data a sender may transmit without feedback from the receiver





---

# Error Detection and Correction

# Error Detection and Correction

---

- **Next task of the LLC layer: protected transmission of the frames to the communication partner**
  - The transmission over layer 1 is not necessarily free of errors!
  - Errors tend to come in bursts rather than single
- **Question: How to recognize errors and repair?**
  - Error-detecting codes
  - Error-correcting codes (Forward Error Correction, FEC)
- **Scenario**
  - A frame consists of  $m$  data bits and  $r$  redundant (check) bits
  - Total length:  $n = m + r$
  - The  $n$ -bit unit with data and check bits is called codeword

# Error-detecting and -correcting Codes

## Parity bits

---

- Compute a short checksum of the data and send it together with the data to the receiver.
- The receiver also computes a checksum of the received data and compares it with those of the sender.
- Simplest procedure: Parity bit
  - Count the number of 1s.
- Example:
  - Sender: 10111001 PB: 1 send: 10111001**1**
  - Receiver: **001011011** PB computed: 0 → Error!



- 1-Bit errors are detected
- 2-Bit errors are not detected
- Corrections are not possible!

# Error-detecting and -correcting Codes

## Parity bits

---

- **Variant: double parity**
  - Improvement of the parity bit procedure by increasing the number of parity bits. For this, several blocks of bits are grouped and treated together.

Sender	
1011	1
0010	1
1100	0
0110	0
0011	

Receiver	
1011	1
0110	0
1100	0
0110	0
0111	

- **An incorrect bit can be identified and corrected by this approach.**

---

# Error Detection: CRC

# Error Detection with Cyclic Codes

---

**Problem:** how to recognize errors in several bits, especially sequences of bit errors?

The use of simple parity bits is not suitable. However, in data communication (modem, telephone cables) such errors occur frequently.

Most often used: **Polynomial Codes**

Idea: a  $m$ -bit PDU ( $a_{m-1}, \dots, a_1, a_0$ ) is considered as a polynomial

$$a_{m-1}x^{m-1} + \dots + x^1a_1 + x^0a_0$$

with the coefficients  $a_i$  “0” and “1”.

Example: 1100101 is interpreted as  $x^6x^5x^4x^3x^2x^1x^0 \Rightarrow x^6 + x^5 + x^2 + 1$

# Error Detection with Cyclic Codes

---

- For computations, polynomial arithmetic modulo 2 is used, i.e., addition and subtraction without carriage.
  - Both operations become Exclusive-OR (XOR) operations.
- Example:

10011011	00110011	11110000	01010101
+ 11001010	+ 11001101	- 10100110	- 10101111
<hr/>	<hr/>	<hr/>	<hr/>
01010001	11111110	01010110	11111010

# Error Detection with Cyclic Codes

---

- **Idea for error detection:**

- Sender and receiver agree upon a **generator polynomial**  $G(x)$

$$G(x) = g_r x^r + g_{r-1} x^{r-1} + \dots + g_1 x^1 + g_0 x^0$$

- The first ( $g_0$ ) and the last ( $g_r$ ) coefficient must be 1

$$G(x) = x^r + g_{r-1} x^{r-1} + \dots + g_1 x^1 + 1$$

- The sender interprets a data block of length  $m$  as polynomial  $M(x)$

$$M(x) = a_{m-1} x^{m-1} + \dots + a_1 x^1 + a_0 x^0$$

- The sender extends  $M(x)$ , i.e., adds »redundant« bits in a way that the extended polynomial  $M'(x)$  is divisible by  $G(x)$

- Redundancy = remainder  $R(x)$  by division of the sequence with  $G(x)$

- The receiver divides the received extended  $M(x)$  by  $G(x)$ . If the remainder is 0, there was no error, otherwise some error occurred

- **Cyclic Redundancy Checksum (CRC)**

- Note: also the parity bit can be seen as CRC, with generator polynomial:  $x + 1$



# Error Detection with Cyclic Codes

---

- **Algorithm for computing the checksum**
  - Let  $r$  be the degree of  $G(x)$ .
    - Append  $r$  zero bits to the low-order end, so it now contains  $m+r$  bits.
    - The corresponding polynomial is  $x^rM(x)$ .
  - Divide the bit string corresponding to  $G(x)$  into the bit string corresponding to  $x^rM(x)$ .
  - Subtract the remainder from the bit string corresponding to  $x^rM(x)$ .
    - The result is the check-summed frame to be transmitted, denoted as  $T(x)$ .

# CRC: Example

Data to be transmitted: 10111001  
Generator polynomial:  $x^4 + x + 1$

Note: usually, the »extra« bits are preset with zeros, but due to some reason Ethernet uses the inverted bits as presets.

Sender:

101110010000 : 10011 = 10100111  
10011  
10000  
10011  
11100  
10011  
11110  
10011  
11010  
10011  
1001 =  $x^3 + 1 = R(x)$

CRC = 1001, sending 10111001**1001**

Receiver:

101110011001 : 10011 = 10100111  
10011  
10000  
10011  
11110  
10011  
11010  
10011  
10011  
10011  
0

Data received correctly!

# CRC is not perfect

---

Receiver:

001010010001 : 10011 = 00101110

```
  10011
  11110
  10011
  11010
  10011
  10010
  10011
  11
```

Error detected

Receiver:

001110110001 : 10011 = 00111111

```
  10011
  11101
  10011
  11100
  10011
  11110
  10011
  11010
  10011
  10011
  10011
  0
```

Error not detected

# CRC: Characteristics

---

- **What kind of errors will be detected?**
  - Instead of  $T(x)$ , erroneous bit string  $T(x)+E(x)$  is received
- **Each 1 bit in  $E(x)$  corresponds to a bit that has been inverted**
  - If there are  $k$  1 bits in  $E(x)$ ,  $k$  single-bit errors have occurred
- **Receiver calculates:  $[ T(x) + E(x) ] / G(x)$** 
  - $T(x)/G(x) = 0$ , thus result is  $E(x)/G(x)$
  - Errors that contain  $G(x)$  as a factor will be not detected
- **Form of  $E(x)$** 
  - $E(x) = x^i$ 
    - Single-bit error,  $i$  determines the bit in error
  - $E(x) = x^i + x^j = x^j (x^i + 1)$ ,  $i > j$ 
    - Double errors can be detected if  $G(x)$  does not divide  $x^{k+1}$ ,  $k$  up to  $i-j$
- **Most important**
  - A polynomial code with  $r$  check bits will detect all burst errors of length  $\leq r$

# CRC: Characteristics

---

Common 16-bit generator polynomials:

- CRC-16:  $G(x) = x^{16} + x^{15} + x^2 + 1$
- CRC CCITT:  $G(x) = x^{16} + x^{12} + x^5 + 1$
- Ethernet:  $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Error detection (for 16-bit generator polynomials):

- all single bit errors
- all double bit errors
- all three-bit errors
- all error samples with odd number of bit errors
- all error bursts with 16 or fewer bits
- 99.997% of all 17-bit error bursts
- 99.998% of all error bursts with length  $\geq 18$  bits
- Remaining error rate  $< 0.5 \times 10^{-5}$  block error rate (original)

Not all errors are detected - an error could consist of adding a multiple of  $G(x)$  to  $M'(x)$

# CRC: Error Correction

---

- In exceptional cases even errors can be corrected by CRCs.
  - **Example: ATM (Asynchronous Transfer Mode)**
    - Data units have fixed length
    - 5 byte header + 48 byte data
    - The last header byte is a **checksum for the header**
    - Generator polynomial  $G(x) = x^8 + x^2 + x + 1$
- ➡ **It is even possible to correct a 1-bit error, due to:**
- there are 40 possible 1-bit errors and those lead on 40 different non-zero remainders.
- 
- **Correction is not assigned with e.g. Ethernet: an Ethernet frame has a length between 64 and 1512 byte.**

---

# Error Correcting: Hamming Code

# Error-detecting and -correcting Codes

---

- **Hamming distance of two codewords  $w_1, w_2$** 
  - Number of places, in which two binary strings differ
  - Example: two codewords  $w_1=10001001$  and  $w_2=10110001$

$$\begin{array}{r} 10001001 \\ \text{XOR } \underline{10110001} \\ 00\color{red}{11}\color{red}{1}000 \Rightarrow d(w_1, w_2) = 3 \end{array}$$

- If two codewords have Hamming distance  $d$ ,  $d$  single bit errors are required to convert one codeword into the other.



# Error-detecting and -correcting Codes

---

- **Hamming distance of a code**
  - $m$  bit data
    - $2^m$  possible data words, typically all used
  - $r$  check bits
  - $n = m + r$  bit codeword
    - $2^n$  possible codewords, typically not all used
  - construct a list of all legal codewords
  - find the two codewords with minimum Hamming distance
    - ➡ this distance is the **Hamming Distance** of the whole code

# Error-detecting and -correcting Codes

---

- **Error-detecting and error-correcting properties of a code depends on its Hamming distance**
  - To **detect**  $d$  errors, a distance of  $d+1$  is required
  - To **correct**  $d$  errors, a distance of  $2d+1$  is required
- **Example:**
  - Code with only four valid codewords
    - $w_1=0000000000$
    - $w_2=0000011111$
    - $w_3=1111100000$
    - $w_4=1111111111$
  - Distance 5
    - it can detect 4 bit errors
    - it can correct 2 bit errors
  - If 0000000111 is received, the original must be 00000**11**111 (correct)
  - If 0000000000 is changed to 0000000**111**, the error is not corrected properly

# Error-detecting and -correcting Codes

---

- **Goal:** A code with  $m$  data bits and  $r$  check bits, which can **correct all single bit errors**
  - Each of the  $2^m$  data bits has  $n$  illegal codewords with distance 1
    - Systematically invert each of the  $n$  bits
  - Each of the  $2^m$  data bits requires  $n+1$  bit patterns
  - Total number of bit sequences  $2^n$   
Requirement:  $(n+1)2^m \leq 2^n$   
With:  $n = m+r$   
➡  $(m + r + 1) \leq 2^r$
  - Given  $m$ , the lower limit on the number of check bits needed to correct single errors
  - The **Hamming Code** fulfills this lower limit

# Hamming Code

---

- **Goal: Use of several parity bits, each of them considering several bits (overlapping). Errors can be identified and corrected by combining the parity bits.**
  - The Hamming code is the »minimal« code of this category.
- **Idea: Representation of each natural number by sum of powers of two.**
- **In a codeword with  $n$  bits:  $w = z_1, \dots, z_n$** 
  - the parity bits are placed exactly at the  $r$  positions, which are a power of two. At the remaining  $m = n - r$  positions the data bits are placed.
  - each bit of the  $r$  additional bits is a parity bit for all places, for which the representation in powers of two contains the position of the additional bit.

# Hamming Code

- Representation of numbers as sum of powers of 2
- Example:  
 $11 = 1 + 2 + 8$
- A bit position is checked by those check bits occurring in its expansion
  - Bit 11 is checked by bits 1, 2, and 8
  - Bit 21 is checked by bits 1, 4, and 16

Number	1	2	4	8	16
1	•				
2		•			
3	•	•			
4			•		
5	•		•		
6		•	•		
7	•	•	•		
8				•	
9	•			•	
10		•		•	
11	•	•		•	
12			•	•	
13	•		•	•	
14		•	•	•	
15	•	•	•	•	
16					•
17	•				•
18		•			•
19	•	•			•
20			•		•
21	•		•		•
22		•	•		•
23	•	•	•		•
24				•	•
25	•			•	•
26		•		•	•
27	•	•		•	•
28			•	•	•

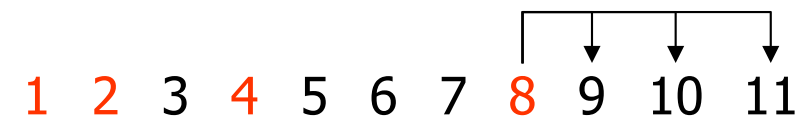
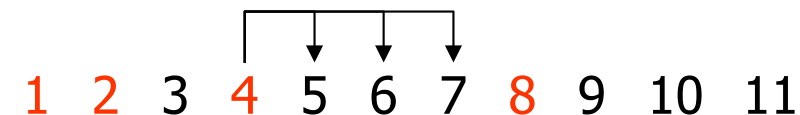
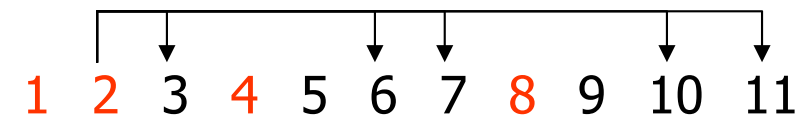
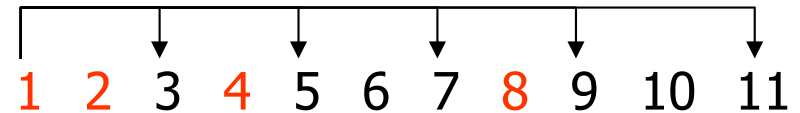
# Hamming Code

- **Method**

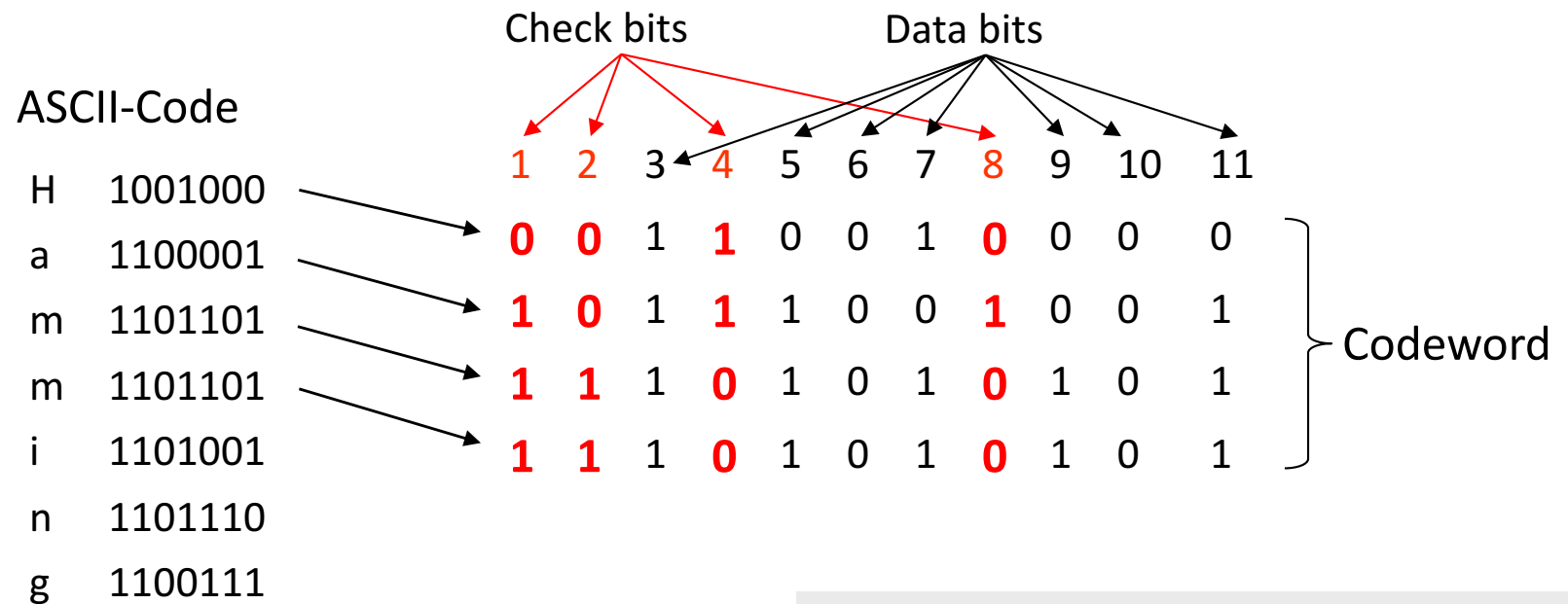
- $m$  data bits
- $r$  check bits
- $n = m + r$  bits codeword
- The bits of the codeword are numbered consecutively starting with 1
- The bits that are powers of 2 are check bits
  - Bits: 1, 2, 4, 8, ...
- The rest are filled up with data bits
  - Bits: 3, 5, 6, 7, ...

- **Example**

- $m=7, r=4$



# Hamming Code



Parity bit 1: Data bit 3, 5, 7, 9, 11       $3 = 1 + 2$

Parity bit 2: Data bit 3, 6, 7, 10, 11       $5 = 1 + 4$

Parity bit 4: Data bit 5, 6, 7       $6 = 2 + 4$

Parity bit 8: Data bit 9, 10, 11       $7 = 1 + 2 + 4$

$9 = 1 + 8$

Problem with Hamming code: errors       $10 = 2 + 8$

involving several following bits are usually       $11 = 1 + 2 + 8$   
corrected wrong.

Receiver:

- examine parity bits
  - if necessary, sum up indices of the incorrect parity bits
    - ➡ index of the incorrect bit
- 1-bit errors can definitely be identified and corrected

# Hamming Code

00110010000  $\xrightarrow{\text{Transmission error}}$  00110000000

Receiver computes parity bits:

11100000000

Summing up the indices 1, 2 and 4 ➡ bit 7 is detected as false

Weaknesses:

- 2-bit errors are not corrected (or wrongly corrected!)
- 3-bit errors are not recognized
- ...

Hamming Code is expensive in terms of required bits!

- a) Bit 4 and bit 11 inverted:
  - ➡ parity bits 1, 2, 4, 8 are wrong
  - ➡ bit 15 is to be corrected, but does not exist
- b) Bit 2 and bit 4 inverted
  - ➡ parity bits 2, 4 wrong
  - ➡ bit 6 is falsely recognized as incorrect
- c) Bits 1, 8, 9 inverted
  - ➡ all parity bits are correct
  - ➡ no error is recognized



---

# Elementary Data Link Protocols

# Error Protection Mechanisms

---

## Error correction: FEC (Forward Error Correction)

- Use of error-correcting codes
- Falsified data in most cases can be corrected. Uncorrectable data are simply dismissed.
- Feedback from the receiver to the sender is not necessary.
- Suitable for transmissions tolerating no transmission delays (video, audio) as well as for coding resp. protecting data on CDs or DVDs.

## Error detection: ARQ (Automatic Repeat Request)

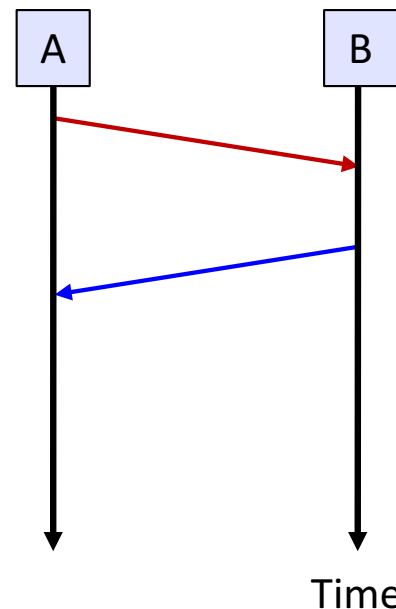
- Use of error-detecting codes (CRC)
- Errors are detected, but cannot be corrected. Therefore, falsified data must be requested again → Retransmission.
- Introduction of flow control:
  - number the data blocks to be sent
  - acknowledgement of blocks by the receiver
  - incorrectly transferred blocks are retransmitted
- Suitable for transmissions which do not tolerate errors (files).

# Implementation of Protocols

---

- **General assumptions**

- Host A sends a long stream of data to Host B
  - Simplex protocol
- Reliable, connection-oriented service
- Infinite supply of data
- Wire-like delivery of packets, i.e., in the sent order



# Implementation of Protocols

---

- **Example: Implementation of a protocol for layer 2**
  - First: Definition of some data types (protocol.h):

```
#define MAX_PKT 1024                /* determines packet size in bytes */
typedef enum {false, true} boolean; /* boolean type */
typedef unsigned int seq_nr;        /* sequence or ack numbers */

typedef struct {
    unsigned char data[MAX_PKT];
} packet;                          /* packet definition */

typedef enum {data, ack, nak} frame_kind; /* frame_kind definition */

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

# Implementation of Protocols

---

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

void from_network_layer(packet *p);    // Fetch a packet from the network layer

void to_network_layer(packet *p);      // Deliver packet to the network layer

void from_physical_layer(frame *r);    // Get frame from the physical layer

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

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

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

void start_ack_timer(void);            // Start an auxiliary timer; enable ack_timeout

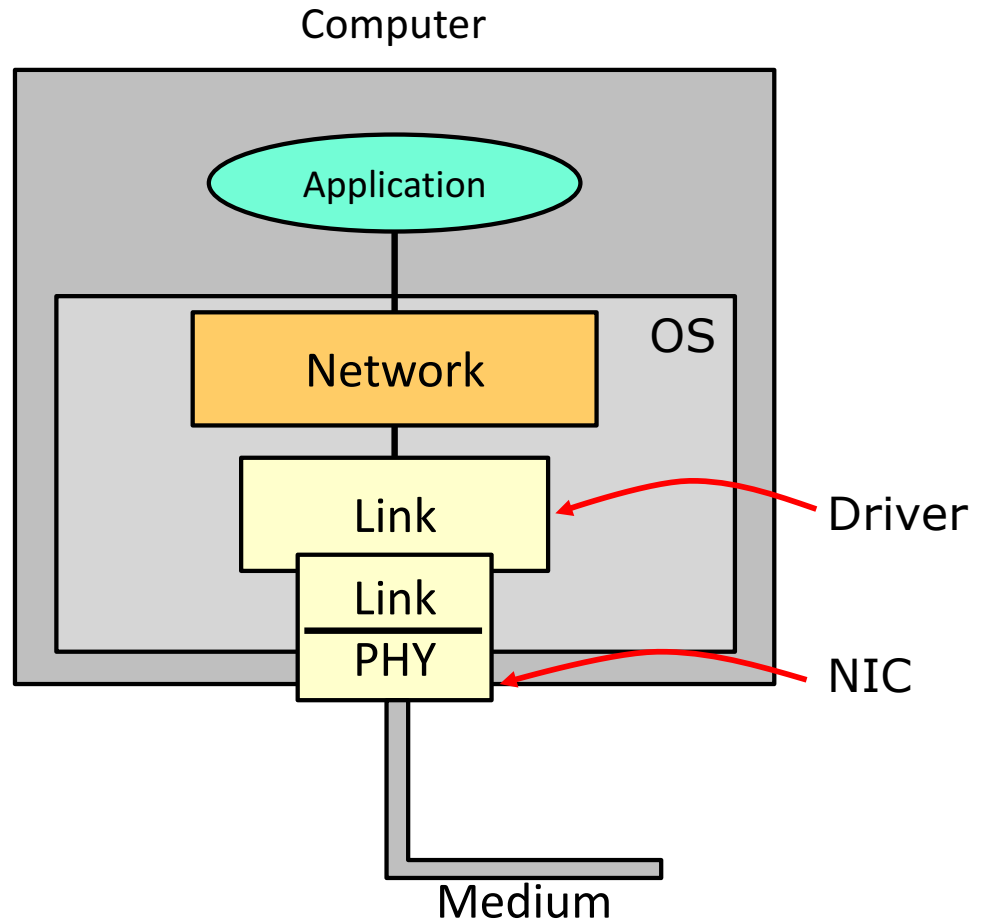
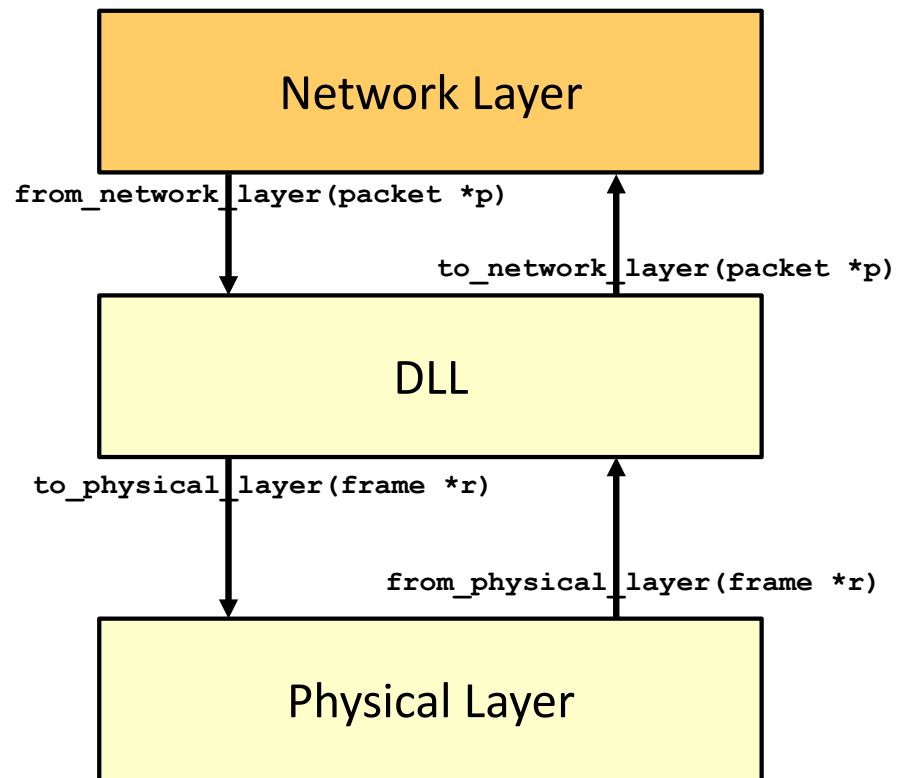
void stop_ack_timer(void);             // Stop auxiliary timer; disable ack_timeout

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

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

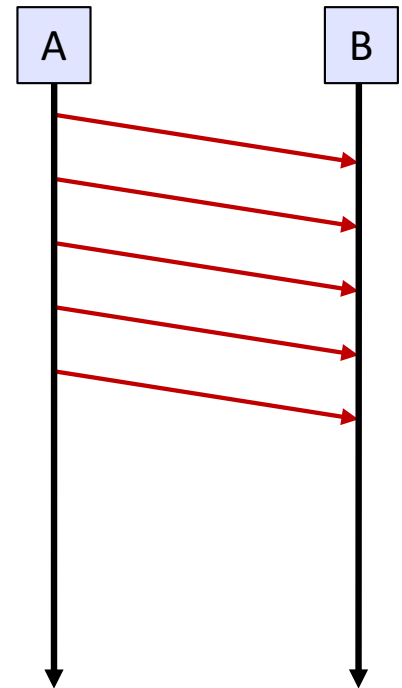
// Macro inc is expanded in-line: Increment k circularly.
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

# Implementation of Protocols



# Simplex Protocol

- **Protocol 1: Simplex Protocol**
  - Transmission in one direction
  - Network layer is always ready
  - Processing time is ignored
  - Communication channel never damages frames
  - Communication channel never loses frames
  - No usage of sequence numbers or acknowledgements
- **Implementation**
  - Two procedures: sender1() and receiver1()
  - Sender in an infinite loop
    - Fetch data, send data
  - Receiver in an infinite loop
    - Get data, pass to network layer



# Simplex Protocol: Implementation

```
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
   sender to receiver. The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */
```

```
typedef enum {frame_arrival} event_type;
```

```
#include "protocol.h"
```

```
void sender1(void)
```

```
{
    frame s;                                /* buffer for an outbound frame */
    packet buffer;                          /* buffer for an outbound packet */
    while (true) {
        from_network_layer(&buffer);        /* go get something to send */
        s.info = buffer;                   /* copy it into s for transmission */
        to_physical_layer(&s);              /* send it on its way */
    }
}
```

Step 1: Send without restrictions

- No transmission errors
- No flow control

```
void receiver1(void)
```

```
{
    frame r;
    event_type event;                      /* filled in by wait, but not used here */
    while (true) {
        wait_for_event(&event);             /* only possibility is frame_arrival */
        from_physical_layer(&r);            /* go get the inbound frame */
        to_network_layer(&r.info);          /* pass the data to the network layer */
    }
}
```



# Simplex Stop-and-Wait Protocol

- **Protocol 2: Simplex Stop-and-Wait Protocol**

- Drop assumption that receiving network layer processes infinitely quick

- **Main problem**

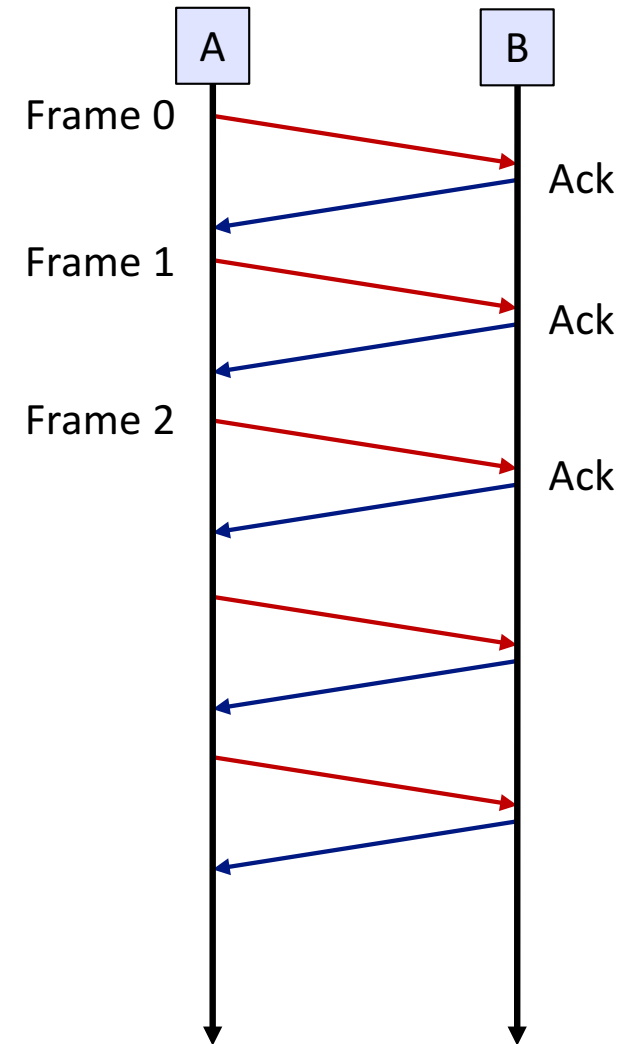
- How to prevent the sender from flooding the receiver with data?

- **Assumptions**

- Communication channel is error free
- Data traffic is simplex
- Channel is bidirectional

- **Solution**

- Simple procedure: The sender sends a data block and **waits**, until an **acknowledgement** from the receiver arrives **or** a **timeout** is reached.
- Incorrect blocks are repeated, otherwise the next block is sent.
- Disadvantage: **large waiting periods** between the transmission of single blocks. Thus much transmission capacity is wasted.



# Simplex Stop-and-Wait Protocol:

## Implementation

```
/* Protocol 2 (stop-and-wait) also provides for a one-directional flow of data from
sender to receiver. The communication channel is once again assumed to be error
free, as in protocol 1. However, this time, the receiver has only a finite buffer
capacity and a finite processing speed, so the protocol must explicitly prevent
the sender from flooding the receiver with data faster than it can be handled. */
```

```
typedef enum {frame_arrival} event_type;
```

```
#include "protocol.h"
```

```
void sender2(void)
```

```
{
    frame s;                /* buffer for an outbound frame */
    packet buffer;          /* buffer for an outbound packet */
    event_type event;       /* frame_arrival is the only possibility */
    while (true) {
        from_network_layer(&buffer); /* go get something to send */
        s.info = buffer;             /* copy it into s for transmission */
        to_physical_layer(&s);       /* bye-bye little frame */
        wait_for_event(&event);      /* do not proceed until given the go ahead */
    }
}
```

```
void receiver2(void)
```

```
{
    frame r, s;             /* buffers for frames */
    event_type event;       /* frame_arrival is the only possibility */
    while (true) {
        wait_for_event(&event); /* only possibility is frame_arrival */
        from_physical_layer(&r); /* go get the inbound frame */
        to_network_layer(&r.info); /* pass the data to the network layer */
        to_physical_layer(&s);     /* send a dummy frame to awaken sender */
    }
}
```

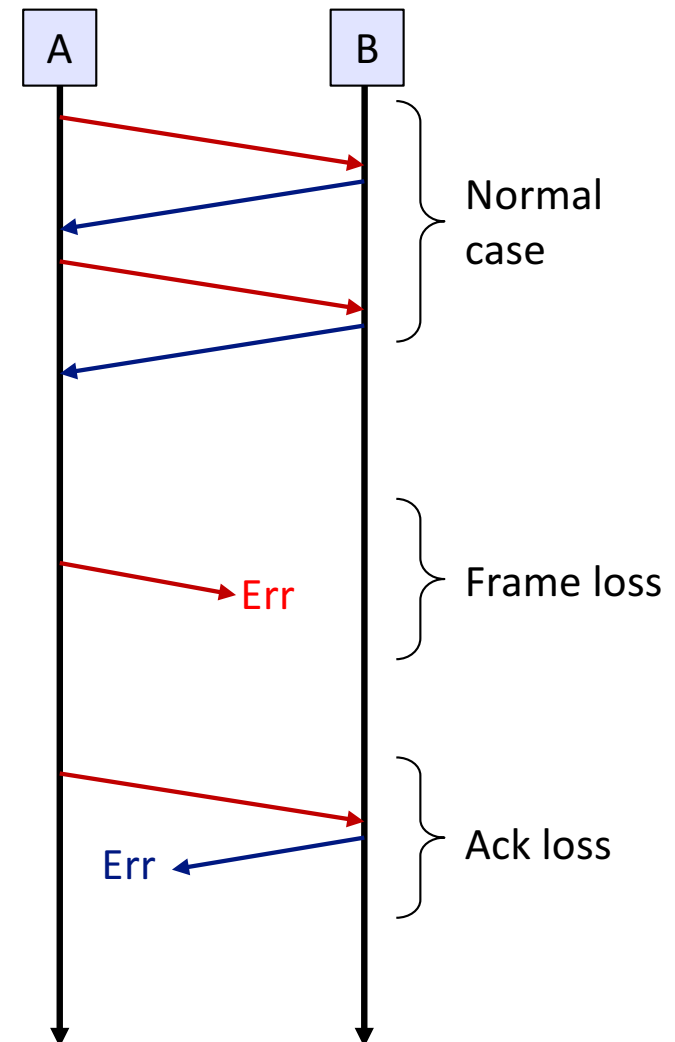
Step 2: Simple flow control

- No transmission errors
- Send-and-Wait as flow control

# Simplex Protocol for a Noisy Channel

- **Protocol 3: Simplex Protocol for a Noisy Channel**

- Communication channel makes errors
  - Frames may be damaged or lost completely
  - If frame is damaged, receiver (hardware?) detects errors
- Types of lost messages
  - Frame is lost
  - Ack is lost

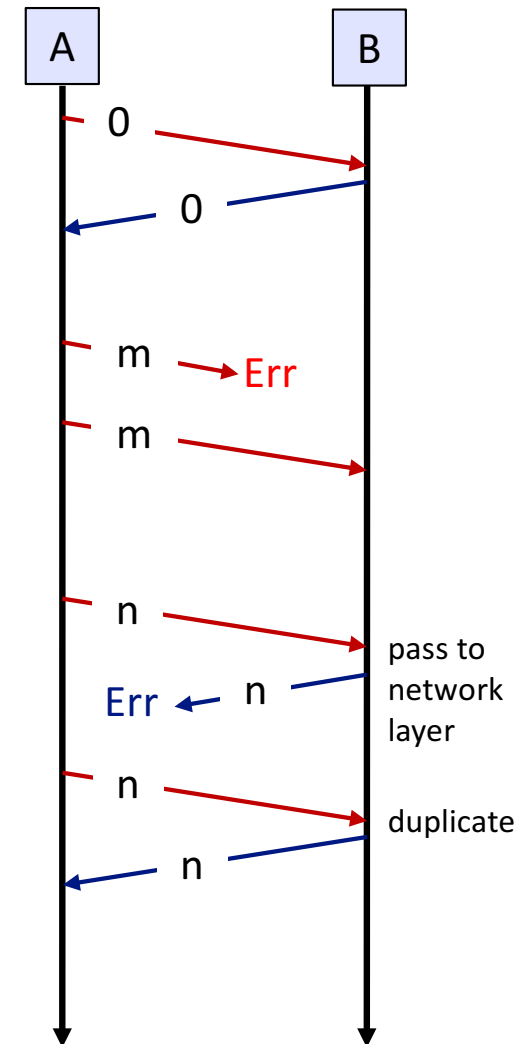


# Simplex Protocol for a Noisy Channel

## Automatic Repeat Request (ARQ)

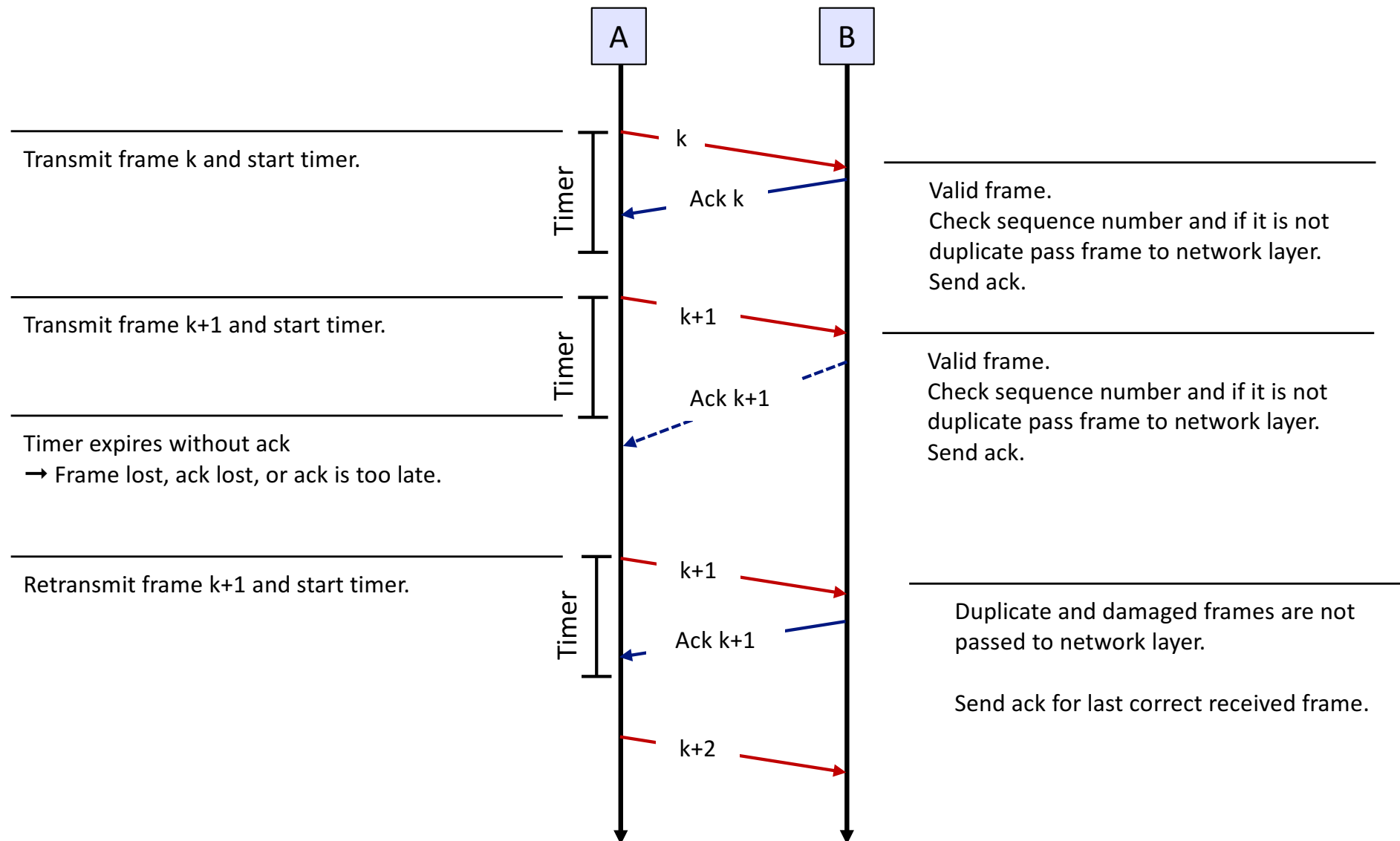
- **Requirement**
  - The sender needs a guarantee that the frame was correctly received
    - Acknowledgement from receiver
- **How to distinguish original and retransmitted frame?**
  - Solution: Put a sequence number into each frame
  - What is the minimum number of bits needed for the sequence number?
  - Enough to distinguish between frame  $m$  and frame  $m+1$ 
    - ➔ 1 bit sequence number sufficient
- **Receiver expects a particular sequence number**
  - If arriving frame has correct sequence number accept, otherwise reject

### ➔ Automatic Repeat reQuest (ARQ)



# Automatic Repeat Request (ARQ)

## Simplex Protocol for a Noisy Channel



# Simplex Protocol for a Noisy Channel:

## Implementation

- Step 3: Error handling and flow control – Sender

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

# Simplex Protocol for a Noisy Channel:

## Implementation

---

```
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);
        }
    }
}
```

# Full-Duplex Communication and Piggybacking

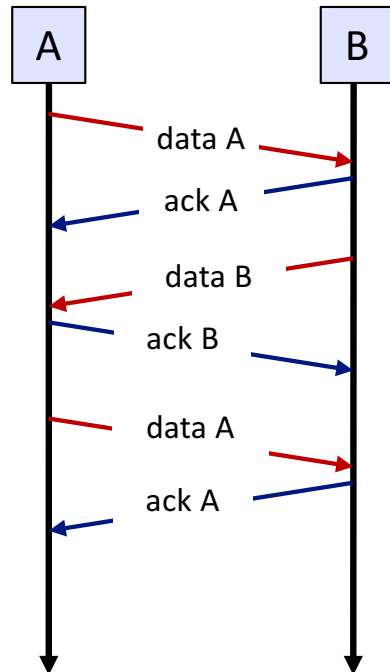
---

- **Previous protocols work only in one direction (simplex)**
- **What to do if full-duplex communication is required?**
  - Use two different simplex communication channels
    - Wasting of resources, since the acknowledgements are rare and small
  - Better idea: use same channel for both directions
    - Data frames and acks are intermixed
    - Kind-field in header distinguishes data- and ack-frames
- **Piggybacking**
  - Instead of using special ack-packets, use a field in the header of a data frame to inform the receiver
    - When a data packet arrives, the receiver does not send immediately an ack, instead waits a particular time interval for a data packet to the other direction
  - Question: How long to wait?
    - Estimate/Guess
    - Fix time
    - RTT

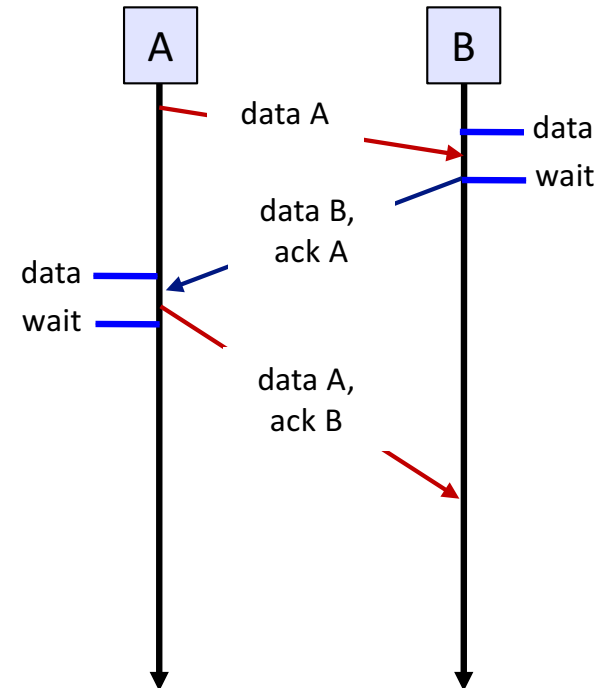


# Full-Duplex Communication and Piggybacking

Data- and ack-frames as individual messages



Ack-frames piggybacked



# Sliding Window

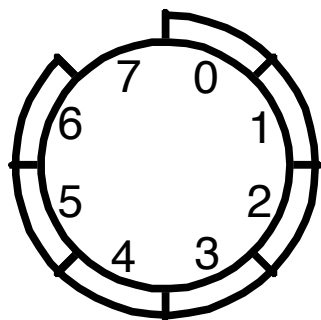
---

- **Introduction of a transmission window**
  - Common procedure to avoid long waiting periods of the sender
  - Sender and receiver agree upon a **transmission window**. If  $W$  is the window size, it means: the sender may send up to  $W$  messages without an acknowledgement of the receiver.
  - Sender and receiver window do not need to have the same limits
  - The messages are sequentially numbered in the frame header
    - Sequence number in range of 0, 1, 2, ...,  $2^n-1$
    - $2^n = \text{MODULUS}$
    - In stop-and-wait  $n=1 \Rightarrow \text{Sequence number} \in \{0, 1\}$
  - The sender may send up to  $W$  messages sequentially numbered without getting an acknowledgement for the first frame.
  - The receiver confirms the reception of a frame by an acknowledgement (ACK).
  - The sender moves the window as soon as an ACK arrives.
  - All frames in the window must be **buffered** for retransmission
    - Window size  $n \Rightarrow \text{Buffer for } n \text{ frames required}$
  - Receiver window corresponds to frames it may accept
    - Frame outside the window is discarded

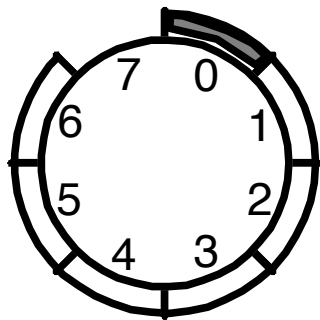
# Sliding Window

**Example** (for 3-bit sequence/acknowledgement number)

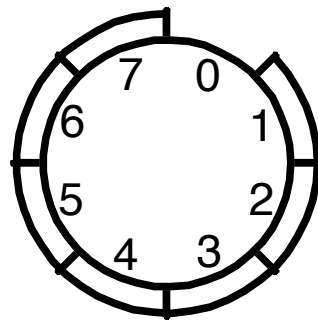
- with 3 bits for sequence/acknowledgement number,  $m = 8$  possible combinations
- Stations agree upon a window size  $W$  with  $1 \leq W < m$ , e.g.  $W = 7$
- The window limits the number of unconfirmed frames allowed at one time (here max. 7, because of  $W = 7$ )
- With receipt of an acknowledgement, the window is shifted accordingly
- Frames are numbered sequentially modulo  $m$  (for  $m=8$  thus numbers from 0 to 7)



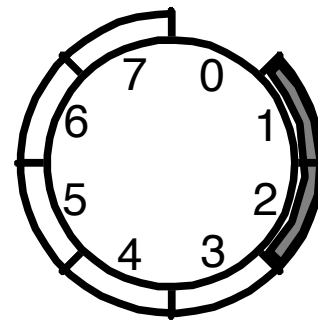
Station sends frames  
0 - 6



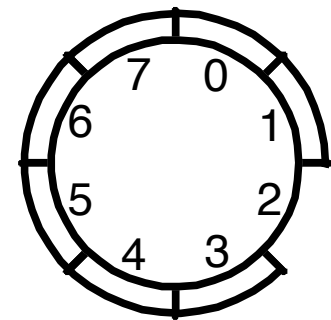
Station receives  
acknowledgement 0



Station slides window by 1  
and sends frame 7



Station receives  
acknowledgement 1,2



Station slides window by 2  
and sends frames 0,1

# Maximum Window Size with Sliding Window

---

There is a reason why window size  $W$  has to be smaller than MODULUS:

- Sequence numbers e.g. have 3 bits:  $2^3 = 8$  sequence numbers (0, ..., 7)

➡ MODULUS = 8

- Assume the window size to be  $W = 8$ . A sends 3 frames to B.

- B acknowledges frame 2 (ACK 3 was sent to A)

B has acknowledged 0 1 2

- A sends 8 frames, without receiving an acknowledgement from B

A sends 0 1 2 3 4 5 6 7 0 1 2

- A receives an acknowledgement ACK 3.

Case 1: B only has received 0 1 2

Case 2: B has received 0 1 2 3 4 5 6 7 0 1 2

- A does not know whether case 1 or 2 holds for B

→ the acknowledgement is not clear!

➡  $W \leq 7$  ( $W < \text{MODULUS}$  in general)

# Sliding Window

---

- **Protocol 4: A one-bit sliding window protocol**
  - Special case with max. window size of 1 ➡ Stop-and-wait
  - Sender transmits a frame and waits

# A One-bit Sliding Window Protocol: Implementation

```
/* 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 */

    // while() part - see next slide
}
```

Only one side has to  
run that!

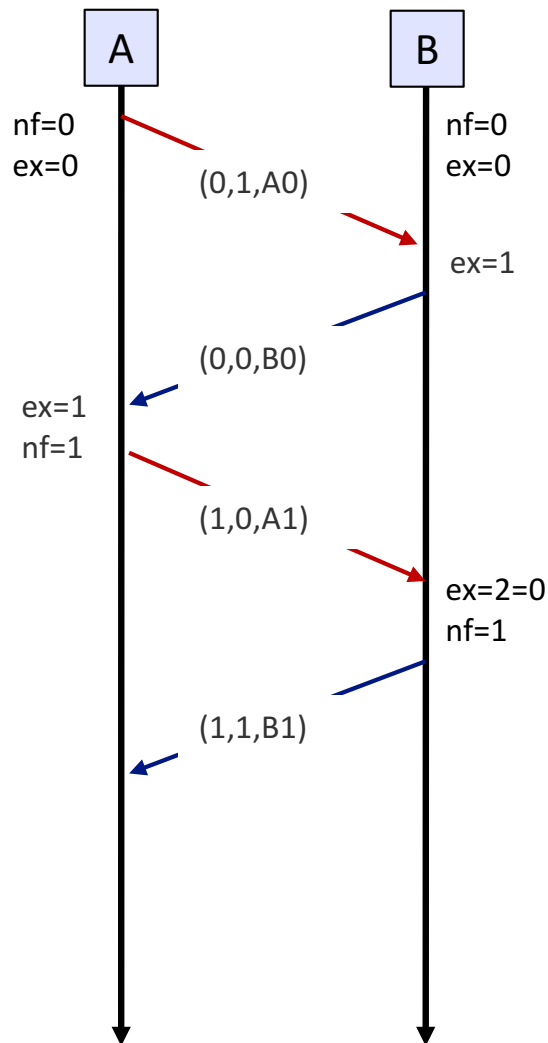
# A One-bit Sliding Window Protocol: Implementation

---

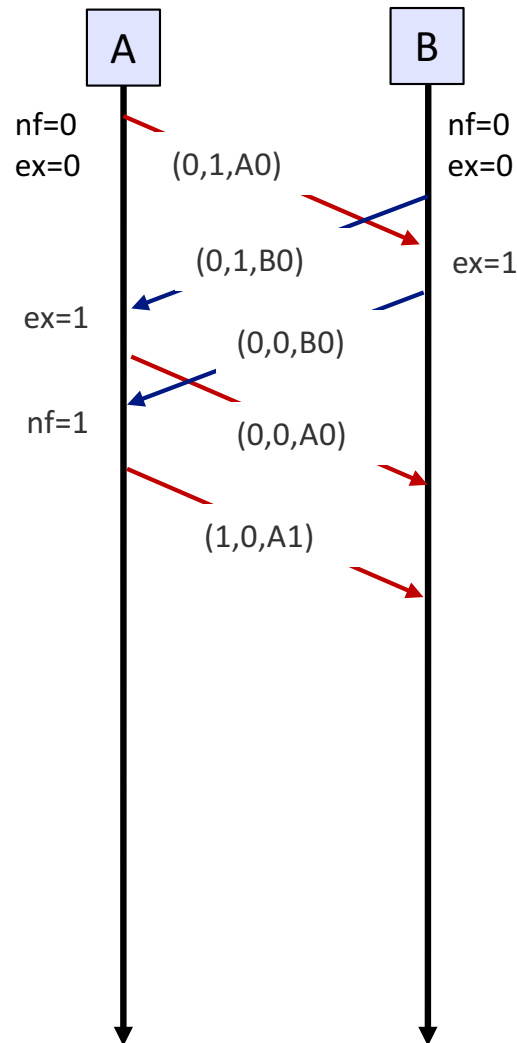
```
while (true) {  
    wait_for_event(&event);          /* frame_arrival, cksum_err, or timeout */  
    if (event == frame_arrival) {    /* a frame has arrived undamaged. */  
        from_physical_layer(&r);    /* go get it */  
  
        if (r.seq == frame_expected) { /* handle inbound frame stream. */  
            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 sender's sequence number */  
        }  
    }  
  
    s.info = buffer;                  /* construct outbound frame */  
    s.seq = next_frame_to_send;       /* insert sequence number into it */  
    s.ack = 1 - frame_expected;       /* seq number of last received frame */  
    to_physical_layer(&s);            /* transmit a frame */  
    start_timer(s.seq);              /* start the timer running */  
}
```

# A one-bit Sliding Window Protocol

Normal operation of the protocol



Particular situation when both sides send an initial frame



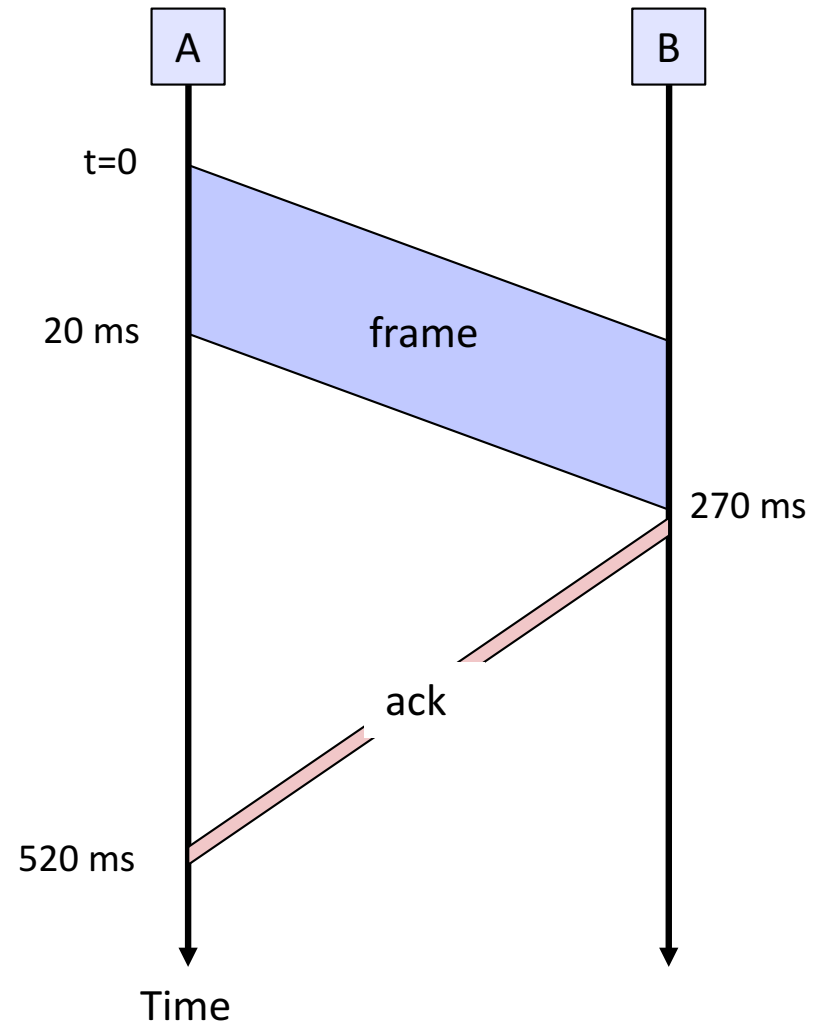
nf = next frame  
ex = expected frame  
Packet format is  
(seq, ack, packet number)

Half of the frames contain  
duplicates, even no transmission  
errors!



# Sliding Window

- Until now transmission times assumed negligible
- **Example scenario**
  - Long round-trip time
  - 50kbps satellite channel with 500msec round-trip propagation time
  - Transmission of 1000-bit frame
  - Sender is blocked  $500/520=96\%$  of the time
  - Utilization only 4%



# Sliding Window

---

- **Source of problem**

- Sender has to wait for an ack before sending next frame

- **Solution**

- Allow sender to transmit up to  $W$  frames before blocking
- In the example  $W=520/20=26$ 
  - ➡ Maximum window size = 26
- If bandwidth x round-trip-delay large, large window is required
  - ➡ Capacity of the pipe
- Sender has to fill the pipe

## ➡ Pipelining

- **Pipelining**

- Assumptions
  - Channel capacity is  $b$  bits/sec
  - Frame size  $l$  bits
  - Round-trip-time  $R$  sec
- Time to transmit a frame:  $l/b$  sec
- After sending last bit, delay of  $R/2$  sec
- Another  $R/2$  sec delay for ack

- **Example: Stop-and-wait**

- Busy for  $l/b$  sec
- Idle for  $R$  sec
- Utilization  $l/(l+bR)$
- If  $l < bR$  ➡ Utilization  $< \frac{1}{2}$
- If  $l = bR$  ➡ Utilization  $= \frac{1}{2}$

# Sliding Window: Pipelining and Go-back-N

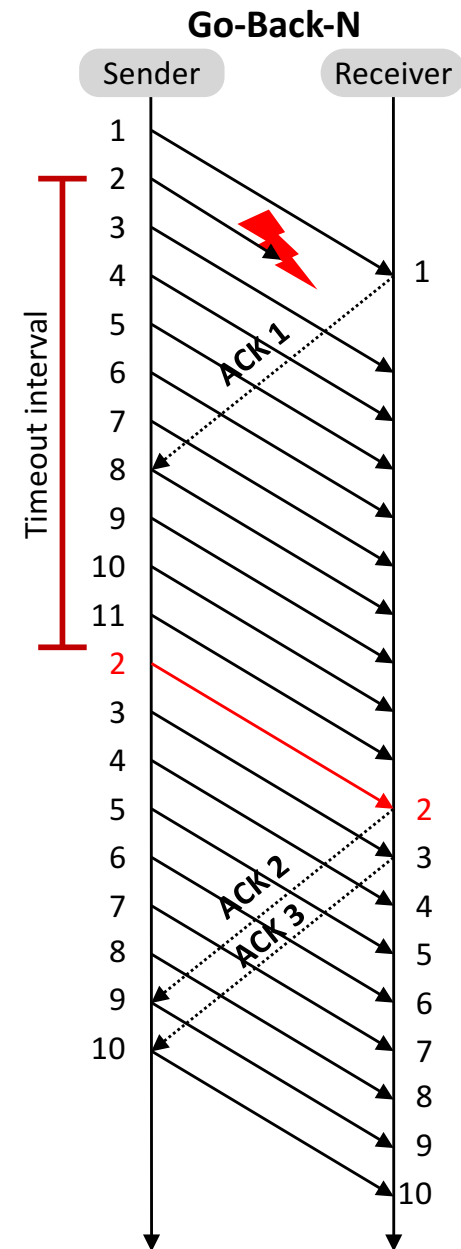
- **Reaction of the sender**

- Transmits frames with sequence number according to the transmission window
- Reception of ack
  - Advance transmission window
- Time out event
  - Retransmit unacknowledged frames in transmission window

- **Reaction of the receiver**

- Sequence number correct
  - Accept frame
  - Send ACK j: everything up to frame j is correct (cumulative ack)
- Sequence number wrong
  - Discard frame

- **Advantage: the receiver needs only one buffer place**



# Sliding Window: Pipelining and Go-back-N

## Implementation

```
/* Protocol 5 (go back n) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike in 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[])
{
    frame s;
    s.info = buffer[frame_nr];
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    to_physical_layer(&s);
    start_timer(frame_nr);

    /* Construct and send a data frame. */
    /* scratch variable */
    /* insert packet into frame */
    /* insert sequence number into frame */
    /* piggyback ack */
    /* transmit the frame */
    /* start the timer running */
}
```

# Sliding Window: Pipelining and Go-back-N

## Implementation

---

```
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 */
}
```

# Sliding Window: Pipelining and Go-back-N

## Implementation

```
while (true) {
    wait_for_event(&event);
    switch(event) {
        case network_layer_ready:
            from_network_layer(&buffer[next_frame_to_send]);
            nbuffered = nbuffered + 1;
            send_data(next_frame_to_send, frame_expected, buffer);
            inc(next_frame_to_send);
            break;

        case frame_arrival:
            from_physical_layer(&r);

            if (r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
            }

            while (between(ack_expected, r.ack, next_frame_to_send)) {
                nbuffered = nbuffered - 1;
                stop_timer(ack_expected);
                inc(ack_expected);
            }
            break;
    }
}
```

*/\* four possibilities: see event\_type above \*/*

*/\* the network layer has a packet to send \*/*  
*/\* Accept, save, and transmit a new frame. \*/*  
*/\* fetch new packet \*/*  
*/\* expand the sender's window \*/*  
*/\* transmit the frame \*/*  
*/\* advance sender's upper window edge \*/*

*/\* a data or control frame has arrived \*/*  
*/\* get incoming frame from physical layer \*/*  
*/\* Frames are accepted only in order. \*/*  
*/\* pass packet to network layer \*/*  
*/\* advance lower edge of recv's window \*/*  
*/\* Ack n implies n-1, n-2, etc. \*/*  
*/\* Check for this. \*/*  
*/\* Handle piggybacked ack. \*/*  
*/\* one frame fewer buffered \*/*  
*/\* frame arrived intact; stop timer \*/*  
*/\* contract sender's window \*/*

# Sliding Window: Pipelining and Go-back-N

## Implementation

---

```
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 */
    }
} // of switch

if (nbuffered < MAX_SEQ)
    enable_network_layer();
else
    disable_network_layer();
} // of while
}
```

# Sliding Window: Pipelining and Selective Repeat

---

- **Selective Repeat (SREPEAT)**
  - Only positive acknowledgements are used
  - When a frame is missing, the **receiver buffers** the following (correct) frames
  - If the sender does not get an acknowledgement for a frame, a timeout occurs and the **sender retransmits** (like in go-back-N) everything beginning with the missing frame
  - When the missing frame arrives, an acknowledgement for the subsequently received frames is sent
  - When the sender gets this acknowledgement, it stops repeating old frames and goes on with new frames
  - Thus, the capacity is used more efficiently, but the **receiver needs more buffer**



# Sliding Window: Pipelining and Selective Repeat Implementation

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

#define MAX_SEQ 7 /* should be 2^n - 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum { frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout } event_type;

#include "protocol.h"
boolean no_nak = true; /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1; /* initial value is only for the simulator */

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

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

# Sliding Window: Pipelining and Selective Repeat Implementation

```
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;
```

# Sliding Window: Pipelining and Selective Repeat Implementation

```
while (true) {
    wait_for_event (&event);
    switch (event) {
        case network_layer_ready:
            nbuffered = nbuffered + 1;
            from_network_layer (&out_buf[next_frame_to_send % NR_BUFS]);
            send_frame (data, next_frame_to_send, frame_expected, out_buf);
            inc (next_frame_to_send);
            break;
        case frame_arrival:
            from_physical_layer (&r);
            if (r.kind == data) {
                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))
                {
                    arrived[r.seq % NR_BUFS] = true;
                    in_buf[r.seq % NR_BUFS] = r.info;
                    while (arrived[frame_expected % NR_BUFS]) {
                        to_network_layer (&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc (frame_expected);
                        inc (too_far);
                        start_ack_timer ();
                    }
                }
            }
    }
}
```

/\* five possibilities: see event\_type above \*/

/\* accept, save, and transmit a new frame \*/

/\* expand the window \*/

/\* fetch new packet \*/

/\* transmit the frame \*/

/\* advance upper window edge \*/

/\* a data or control frame has arrived \*/

/\* fetch incoming frame from physical layer \*/

/\* An undamaged frame has arrived. \*/

/\* Frames may be accepted in any order. \*/

/\* mark buffer as full \*/

/\* insert data into buffer \*/

/\* Pass frames and advance window. \*/

/\* advance lower edge of receiver's window \*/

/\* advance upper edge of receiver's window \*/

/\* to see if a separate ack is needed \*/

# Sliding Window: Pipelining and Selective Repeat Implementation

```
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 ();
}
```

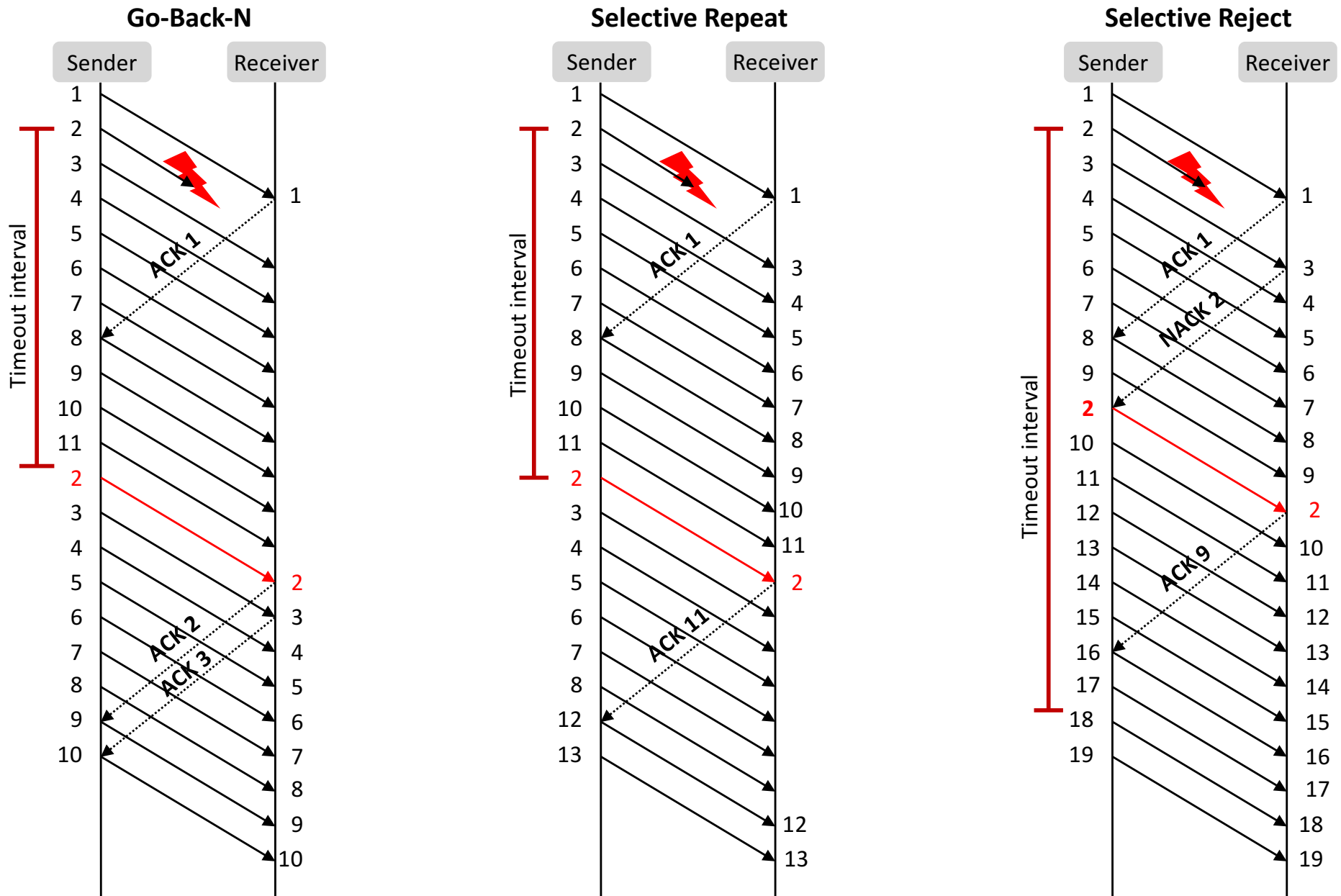
# Sliding Window: Pipelining and Selective Reject

---

- **Selective Reject SREJj**

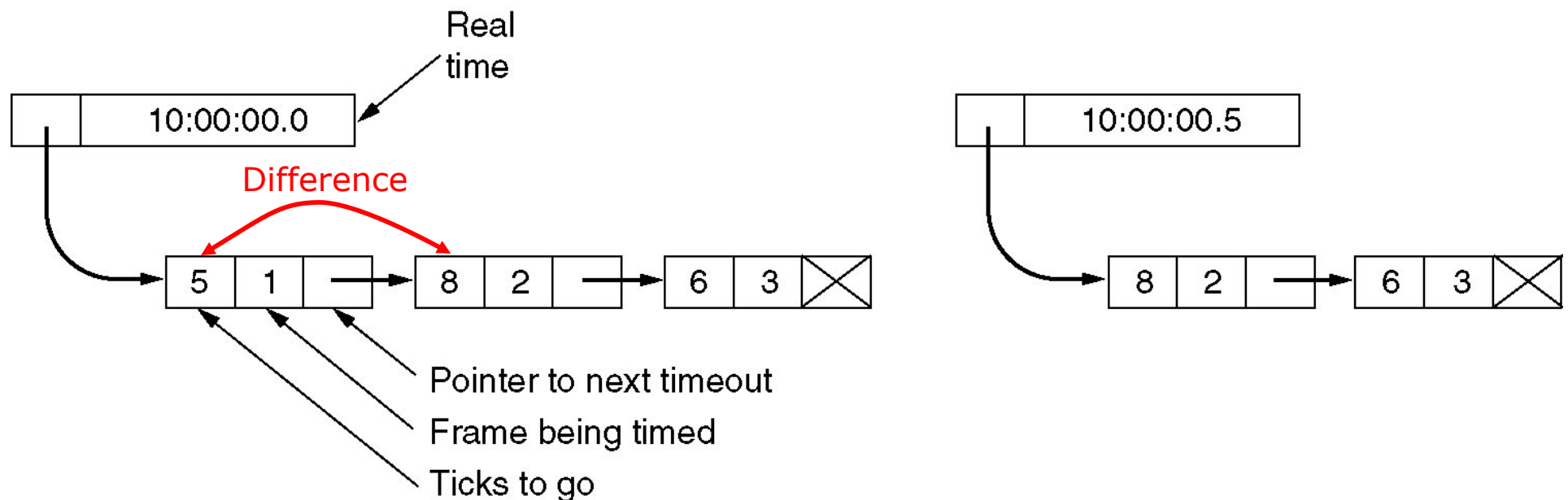
- Like in SREPEAT, correctly received frames after a missing frame are buffered
- The receiver sends a **negative acknowledgement (nack)** for the missing frame j
- The sender retransmits only frame j
  - By this, no unnecessary duplicates are sent, the efficiency of capacity usage again is enhanced.
- Variant: the receiver can send a list of missing frames to the sender, not only single negative acknowledgements
- But: again the receiver needs more buffer.

# Sliding Window: Retransmission Strategies



# Implementation of Timer

- **Some protocols require many timers, but only a couple hardware timers exist**
  - Implement timers in software by using one hardware timer
  - Store expiration times in a linked list and update it during protocol runtime



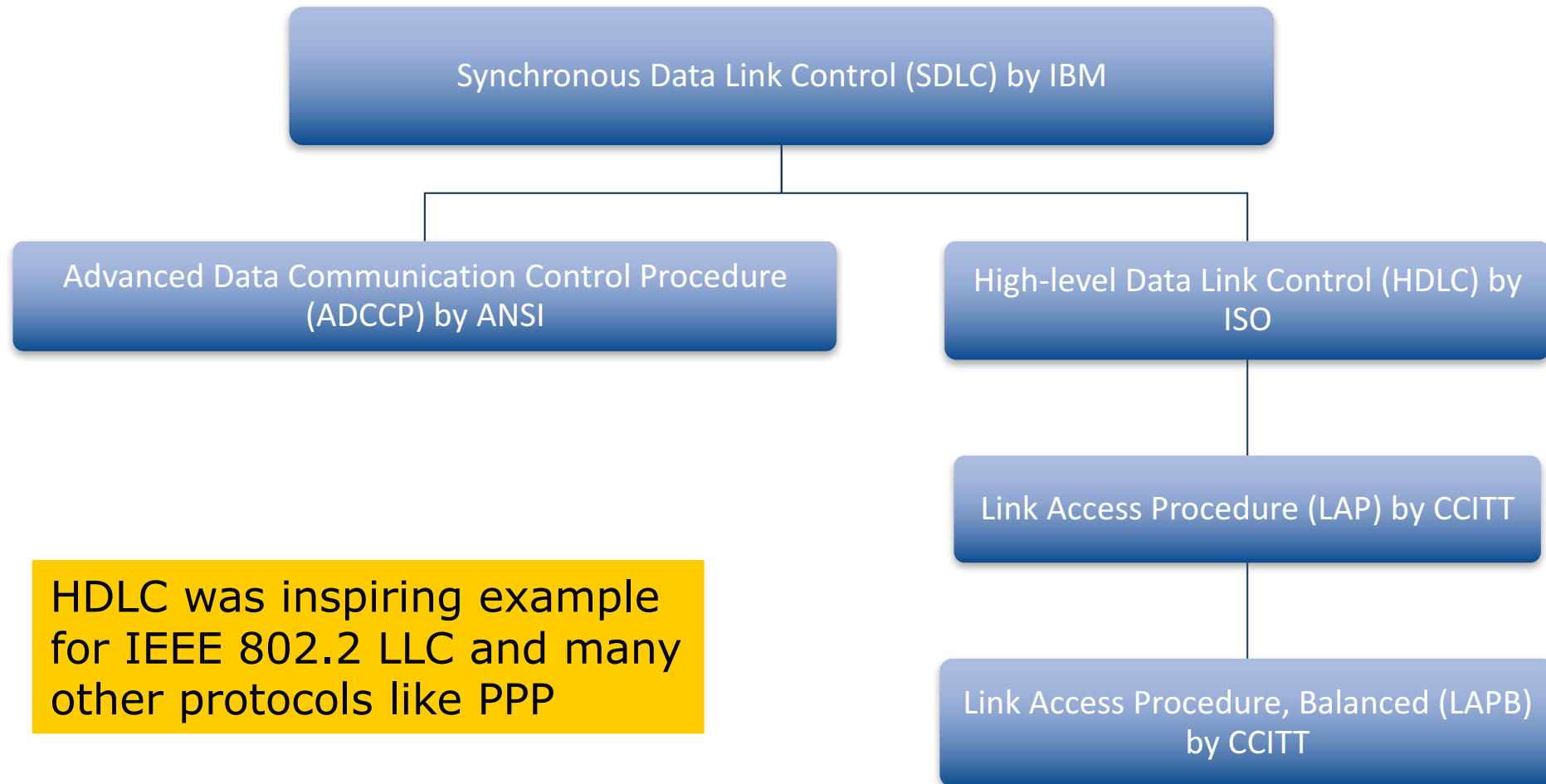
---

# High Level Data Link Control (HDLC)



# High Level Data Link Control (HDLC)

---



# High Level Data Link Control (HDLC)

---

- **Three types of stations**

- Primary station: responsible for controlling the operation of the link.
  - Frames are called commands
- Secondary station: operates under the control of the primary station.
  - Frames are called responses.
- Combined station: combination of primary and secondary.

- **Link configurations**

- Unbalanced configuration
  - One primary and one or more secondary stations
  - Full-duplex and half-duplex transmissions
- Balanced configuration
  - Two combined stations
  - Full-duplex and half-duplex transmissions

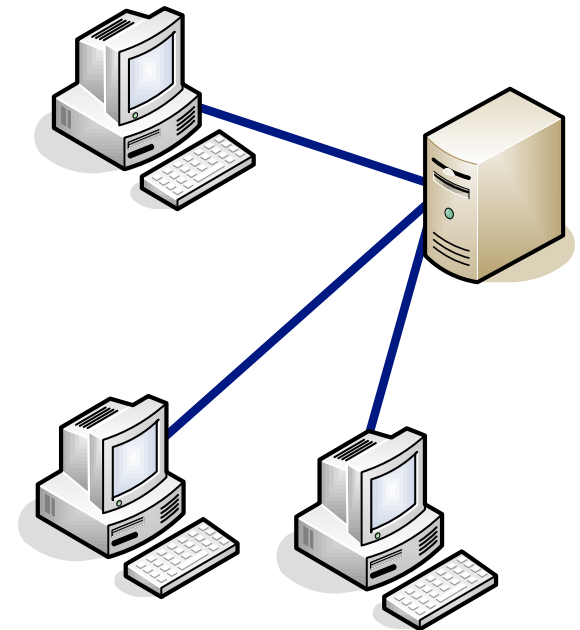


# High Level Data Link Control (HDLC)

## Data transfer modes

---

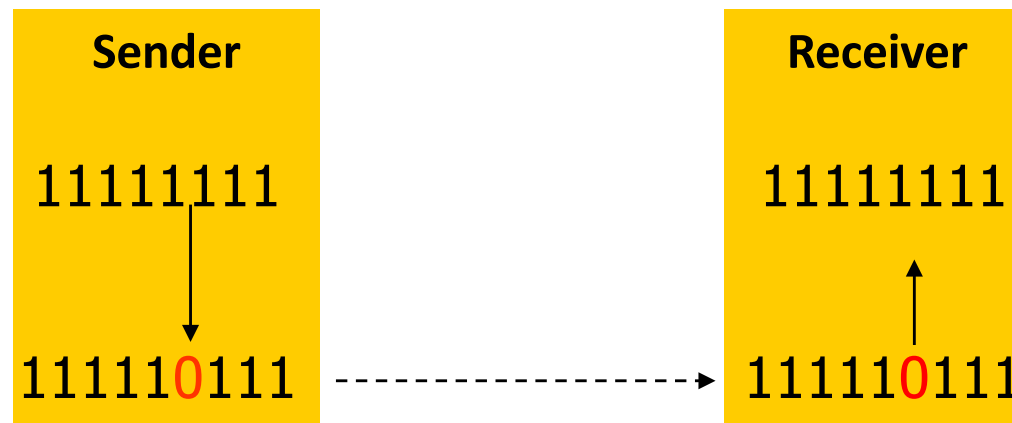
- **Normal response mode**
  - Unbalanced configuration
  - Primary station initiates communication and secondary responses
  - One computer controls several terminals
    - Computer polls the terminals for input
- **Asynchronous balanced mode**
  - Balanced configuration
  - Either combined station initiates communication
  - Most widely used one, no polling
- **Asynchronous response mode**
  - Unbalanced configuration
  - Secondary initiates communication
  - Primary controls the line
  - Rarely used, for special situations



# High Level Data Link Control (HDLC)

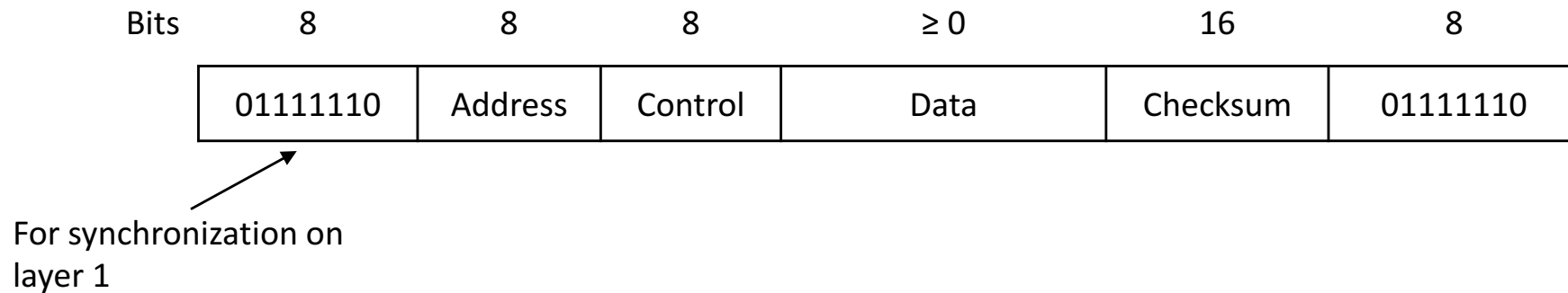
---

- Bit oriented protocol
- Frame identification
  - mark the beginning and end with a flag: “01111110” 0x7E
  - Flag may never occur within a frame
  - Used for this purpose: **bit stuffing**
    - Sender inserts a zero after each sequence of five ones. The receiver removes this zero.



# High Level Data Link Control (HDLC)

---



- **Address**
  - contains the address of the receiver, only important when multiple terminal on line
  - on point-to-point lines used to distinguish commands from responses
- **Control**
  - sequence numbers, acknowledgements, and some other purposes
- **Data**
  - arbitrary long, however efficiency of the checksum falls of with increasing frame length
- **Checksum**
  - is computed by using a CRC
- **Minimum frame consists of 32 bit, excluding the flags**

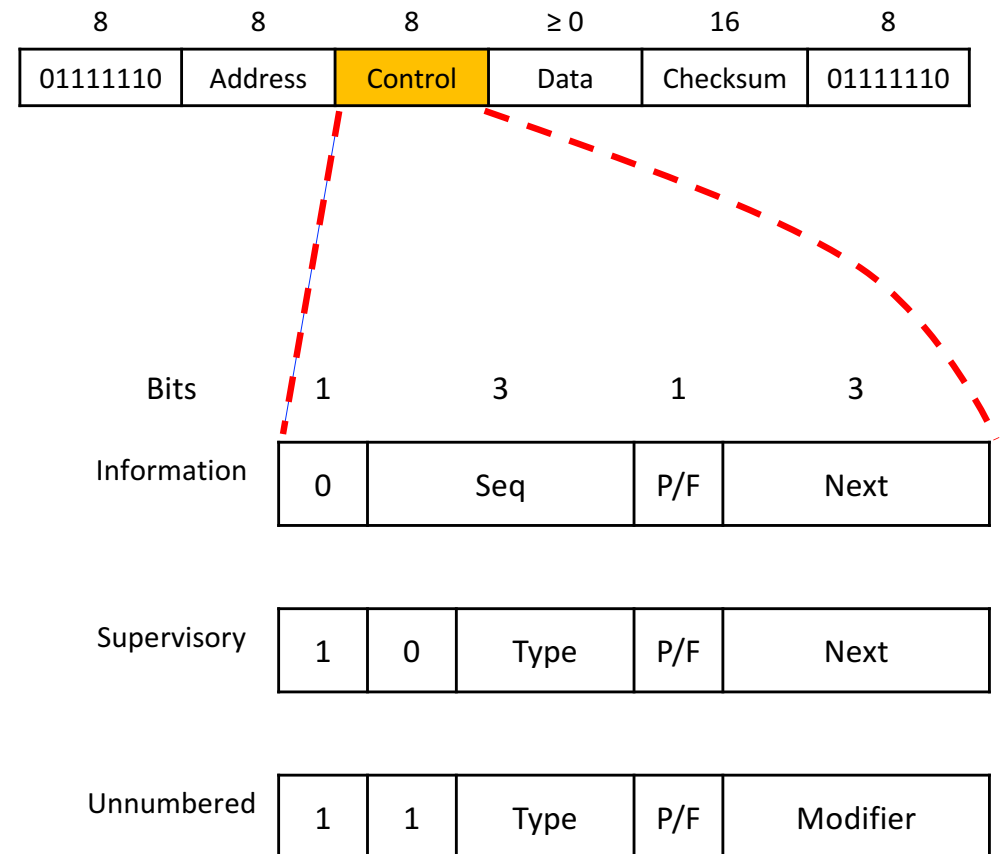
# High Level Data Link Control (HDLC)

---

- **Three kinds of frames**
  - Information
    - Information frames transport user data from the network layer
    - They can include flow and error control information piggybacked on data. The subfields in the control field define these functions.
  - Supervisory
    - Supervisory frames are used for flow and error control when piggybacking is not possible.
  - Unnumbered
    - Unnumbered frames are used for link management.
    - Can carry data when unreliable connectionless service is required.
    - They exchange session management and control information between connected devices.

# High Level Data Link Control (HDLC)

- **Control field of the header distinguishes the frame types**
  - Seq: sequence number
    - HDLC uses sliding window with a 3-bit sequence number
  - Next: piggybacked acknowledgement
    - Number of the first frame not received, i.e., the next frame expected
  - P/F: Poll/Final
    - Used when a master polls a group of terminals
    - If P: Terminal is invited to send data. All frames of the terminal have the P/F bit set to P, only the last frame is set to F.
  - Type: used to distinguish various kinds of frames



# High Level Data Link Control (HDLC)

Type	Name	Fields							
		1	2	3	4	5	6	7	8
I	I (Data Frame)	0	Seq.			P	Next		
S	RR (Receive Ready)	1	0	0	0	P/F	Next		
	RNR (Receive not Ready)	1	0	1	0	P/F	Next		
	REJ (Reject)	1	0	0	1	P/F	Next		
	SREJ (Selective Reject)	1	0	1	1	P/F	Next		
U	SABM (Set Asynchronous Balanced Mode)	1	1	1	1	P	1	0	0
	DISC (Disconnect)	1	1	0	0	P	0	1	0
	UA (Unnumbered ACK)	1	1	0	0	F	1	1	0
	CMDR (Command Reject)	1	1	1	0	F	0	0	1
	FRMR (Frame Reject)	1	1	1	0	F	0	0	1
	DM (Disconnect Mode)	1	1	1	1	F	0	0	0

Send frame Seq., ACK up to Next -1 in the other direction

Ready to receive, ACK up to Next-1 in the other direction

Temporarily not ready to receive, acknowledge up to Next -1 in the other direction

NACK for Next, ACK up to Next-1. The sender should repeat everything beginning with Next

ACK up to Next-1, selective NACK for Next

Connection Establishment

Announcement of connection termination

General ACK (e.g. for connection establishment)

Frame/command not valid (invalid frame, wrong sequence number, ...)

Connection Termination



# High Level Data Link Control (HDLC)

---

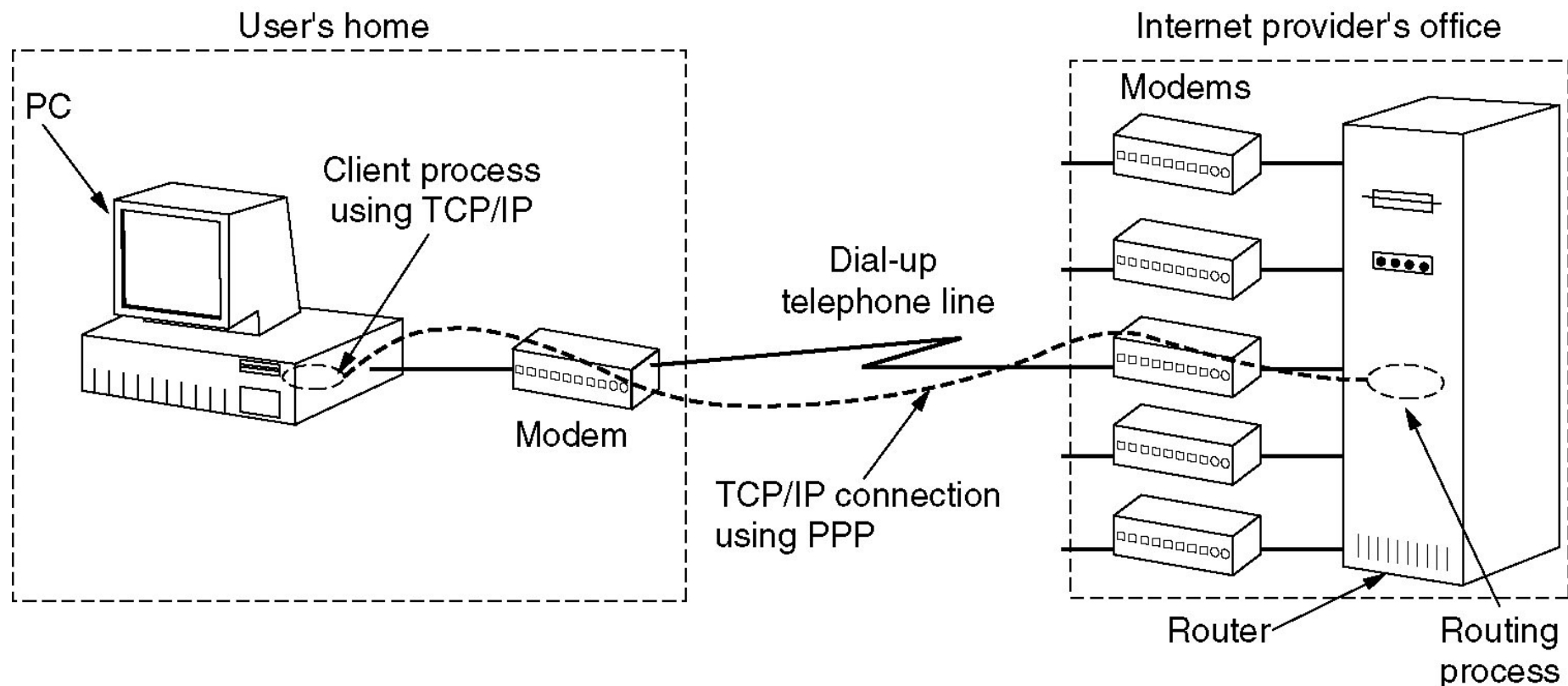
- **Special commands provided ...**
  - DISC (Disconnect): machine announces that it is going down
  - SNRM (Set Normal Response Time): machine announces that it is back
    - Sequence numbers are set to zero
  - SABM (Set Asynchronous Balanced Mode)
    - Resets the line and sets both devices equal
- **Enhancements**
  - 7-bit sequence numbers, instead of 3-bit sequence numbers

---

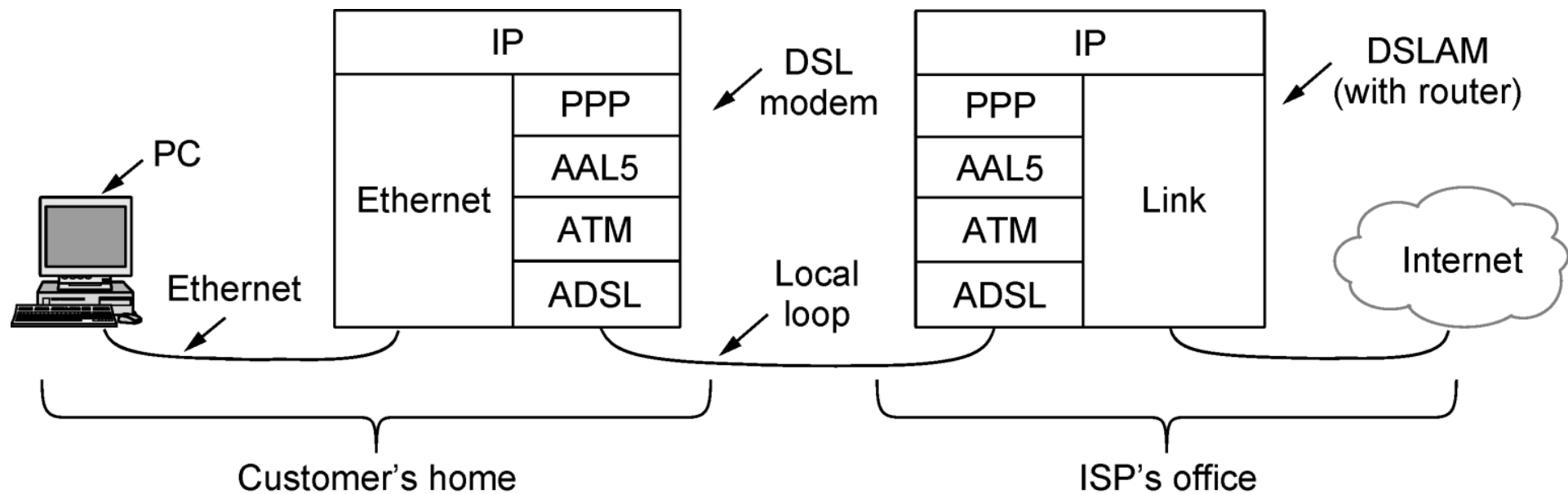
# Point-to-Point Protocol (PPP)

# Point-to-Point Protocol (PPP)

- A home personal computer acting as an internet host



# Point-to-Point Protocol (PPP)



ADSL protocol stacks.

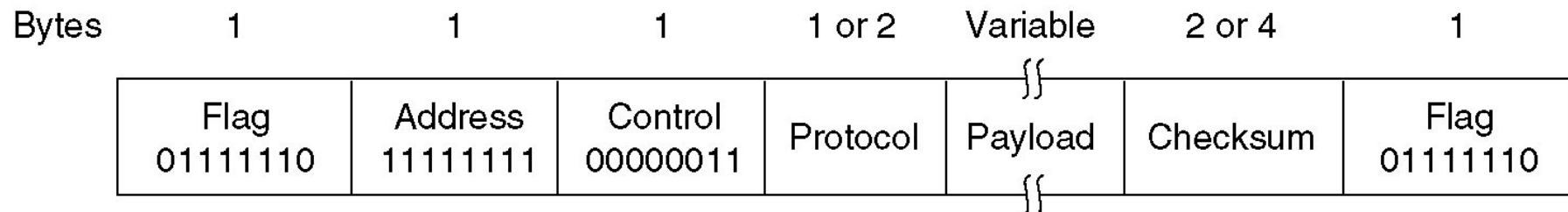
# Point-to-Point Protocol (PPP)

---

- **Point-to-point connections in the Internet**
  - Establish a direct connection between two nodes
    - Dial-up host-router connections
    - Router-router connection
  - PPP is defined in RFC 1661, RFC 1662, RFC 1663
  - Handles error detection, support multiple protocols
  - Supports synchronous and asynchronous connections
- **Features of PPP**
  - Framing method with error detection
  - Link Control Protocol (LCP)
    - Initiation of connections
    - Testing of connections
    - Negotiation of options
    - Terminating of connections
  - Network Control Protocol (NCP) for each network layer supported
    - Negotiate network-layer options, e.g., network address or compression options, after the connection has been established.

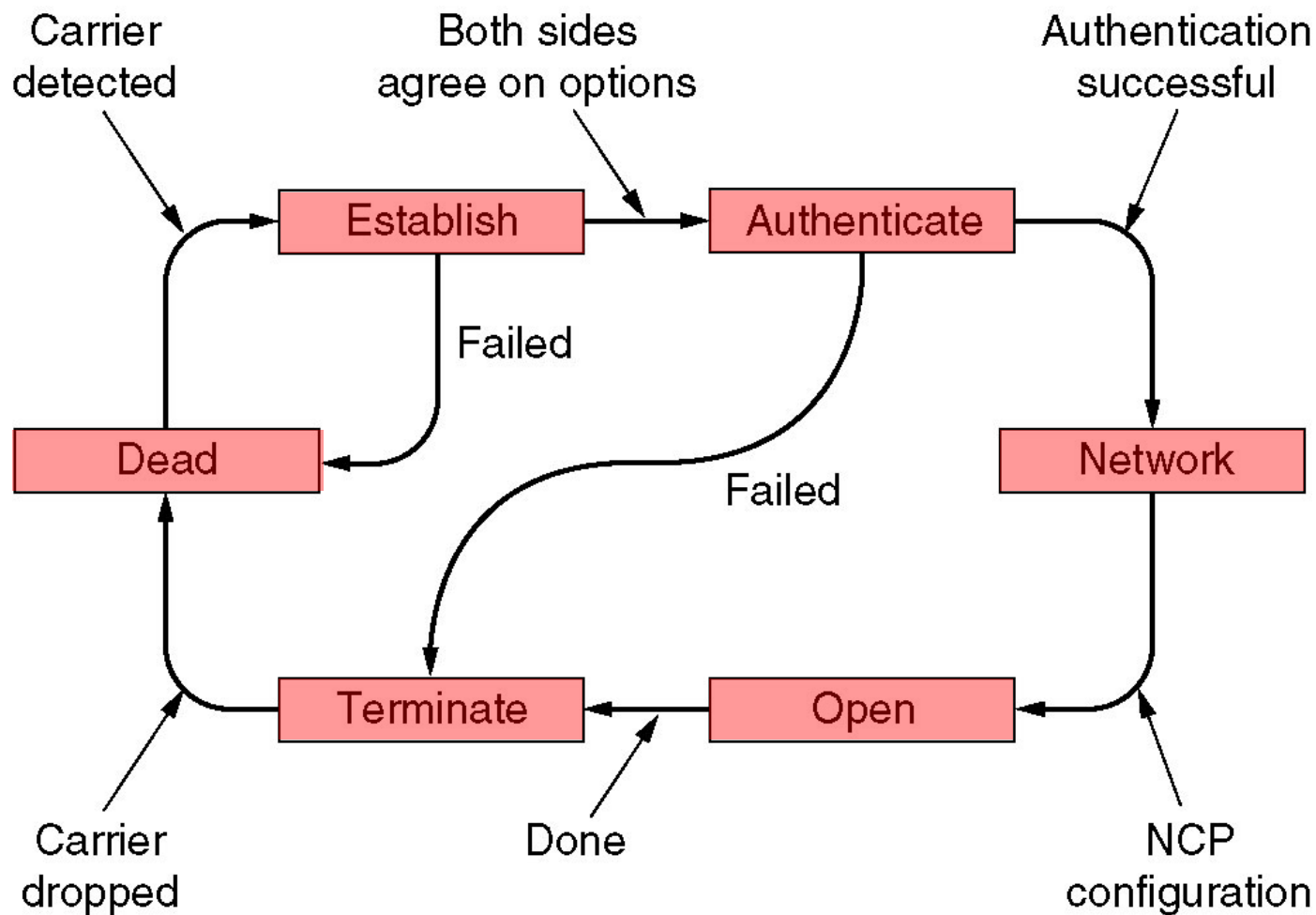
# Point-to-Point Protocol (PPP)

- **PPP frame format resembles the HDLC frame format**
  - Major difference is that PPP is **character oriented** and uses **byte-stuffing**



- **Structure of frames is oriented at HDLC:**
  - Flag: same as for HDLC
  - Address: unnecessary, therefore set to 11111111
  - Control: unnumbered mode (marked by 00000011) without sequence numbers and acknowledgments as default procedure
  - Protocol: specifies contents of the data part (Payload), i.e., contains an identifier to inform the receiver about what to do. "00000110" means "IP protocol used for processing the data".
  - Checksum: 2 byte error detection is usually used

# Point-to-Point Protocol (PPP)



## **Dead**

- Start of the protocol

## **Establish**

- Physical connection is established
- LCP negotiation starts

## **Authenticate**

- Check of identities

## **Network**

- NCP negotiation starts
- Configure network layer

## **Open**

- Data transmission

# Point-to-Point Protocol (PPP)

---

- **The LCP frame types defined in RFC 1661**
  - I = Initiator propose option values
  - R = Responder accepts or rejects proposed options

Name	Direction	Description
Configure-request	I→R	List of proposed options and values
Configure-ack	I←R	All options are accepted
Configure-nak	I←R	Some options are not accepted
Configure-reject	I←R	Some options are not negotiable
Terminate-request	I→R	Request to shut the line down
Terminate-ack	I←R	OK, line shut down
Code-reject	I←R	Unknown request received
Protocol-reject	I←R	Unknown protocol requested
Echo-request	I→R	Please send this frame back
Echo-reply	I←R	Here is the frame back
Discard-request	I→R	Just discard this frame (for testing)



---

# Protocol Verification

# Protocol Verification

---

- **Realistic protocols and their implementations are very complicated**
  - How to verify that an implementation of a protocol is correct?
- **Two concepts for the verification of protocols**
  - Finite State Machine (FSM) Models
  - Petri Net Models

# Finite State Machine Models

---

- **Construct a finite state machine for a protocol**
  - ➔ **Protocol machine**
    - The machine is always in a **specific state**
    - The states consists of all the values of its variables
    - Often, a large number of states can be grouped
    - Number of state is  $2^n$ , where  $n$  is the number of bits needed to represent all variables
    - Well-known techniques from **graph theory** allow the determination of which states are **reachable** and which are not
- **Reachability analysis**
  - Incompleteness: protocol machine is in a state and the protocol does not specify what to do
  - Deadlock: no exit or progress from a state
  - Extraneous transition: event occurs in a state in which it should not occur

# Finite State Machine Models

---

- **Formal definition of a protocol machine as a quadruple**

**$(S, M, I, T)$**

- **S** set of states the processes and channel can be in
- **M** set of frames that can be exchanged over the channel
- **I** set of initial states of the processes
- **T** set of transitions between states

# Finite State Machine Models

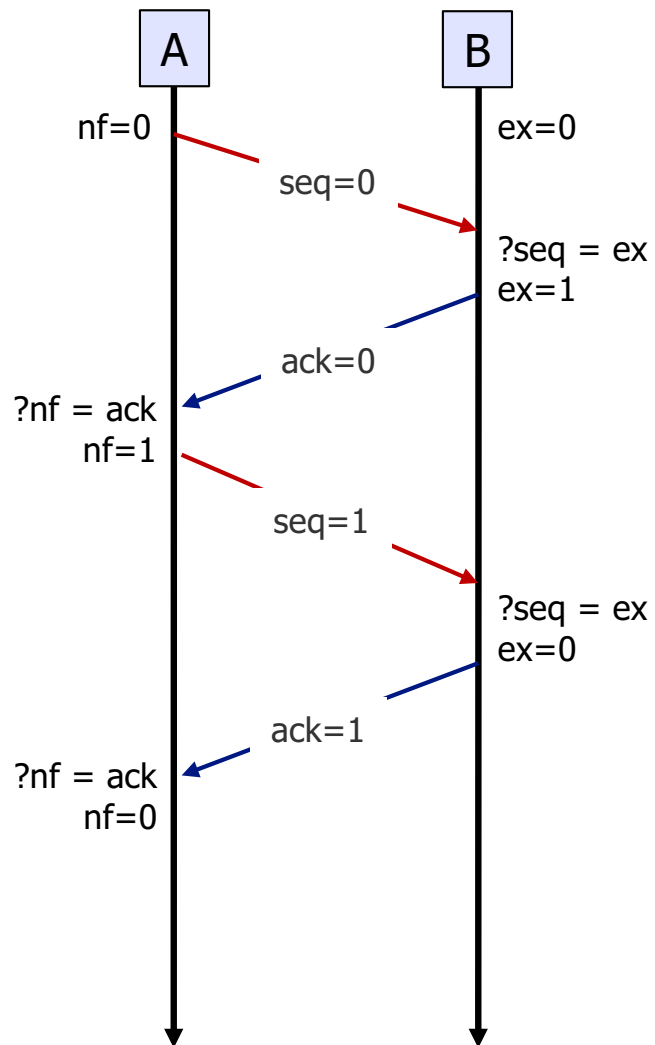
## Example Protocol 3

---

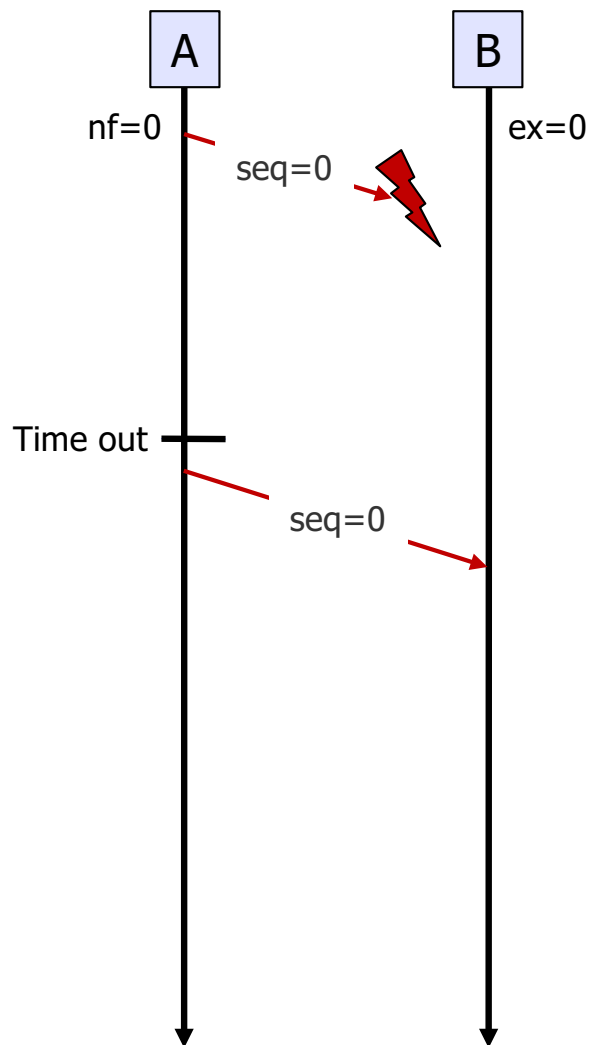
- **Unidirectional data flow over an unreliable channel**
- **Channel has 4 possible states ➡ 2 bits**
  - 0 frame moving from sender to receiver
  - 1 frame moving from sender to receiver
  - Ack frame moving from receiver to sender
  - Empty channel
- **Sender and receiver have 2 possible states ➡ 2 bits each**
- **The whole system has 16 possible states ➡ 4 bits**
- **Each state is labeled by three characters: SRC**
  - $S \in \{0 = \text{sender is trying to send a 0-frame}, 1 = \text{sender is trying to send a 1-frame}\}$
  - $R \in \{0 = \text{receiver expects 0-frame}, 1 = \text{receiver expects 1-frame}\}$
  - $C \in \{0 = \text{0-frame}, 1 = \text{1-frame}, A = \text{ack-frame}, - = \text{empty}\}$
- **Example**
  - State: (SRC) = (000)
    - Sender has sent 0-frame, Receiver expects 0-frame, 0-frame on channel

# Finite State Machine Models: Example Protocol 3

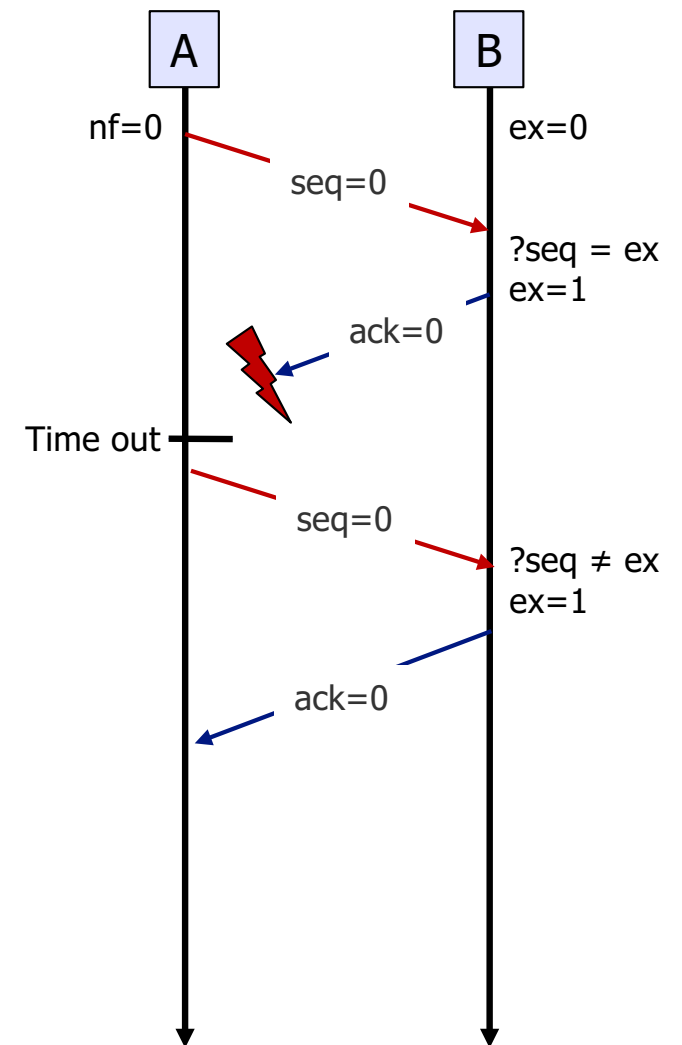
Protocol 3  
Normal operation



Protocol 3  
Loss of a data frame

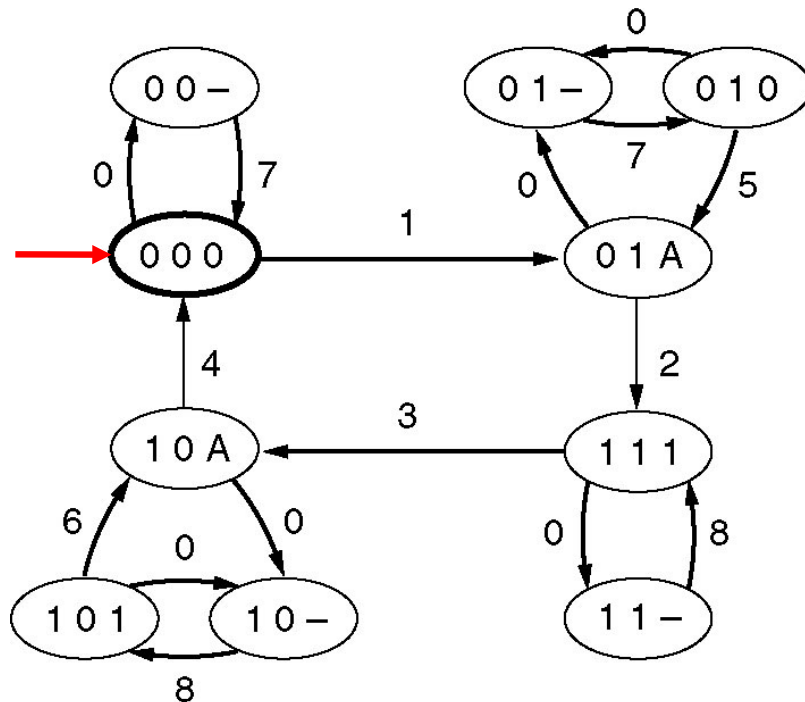


Protocol 3  
Loss of an ack



# Finite State Machine Models: Example Protocol 3

State diagram for protocol 3



Transitions

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	–

Unreachable states are not shown.

State is given by (Sender, Receiver, Channel) = (SRC)

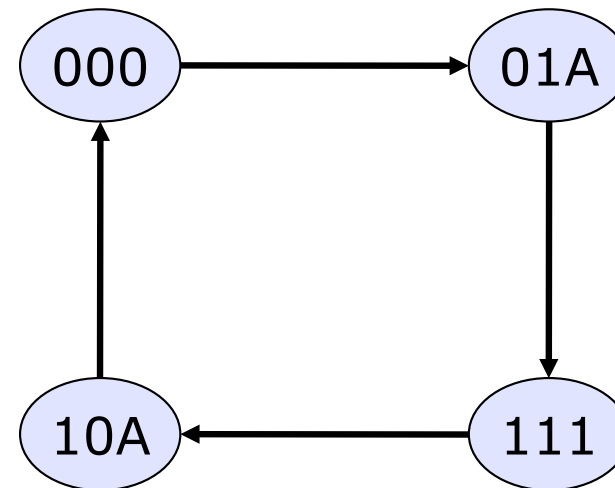
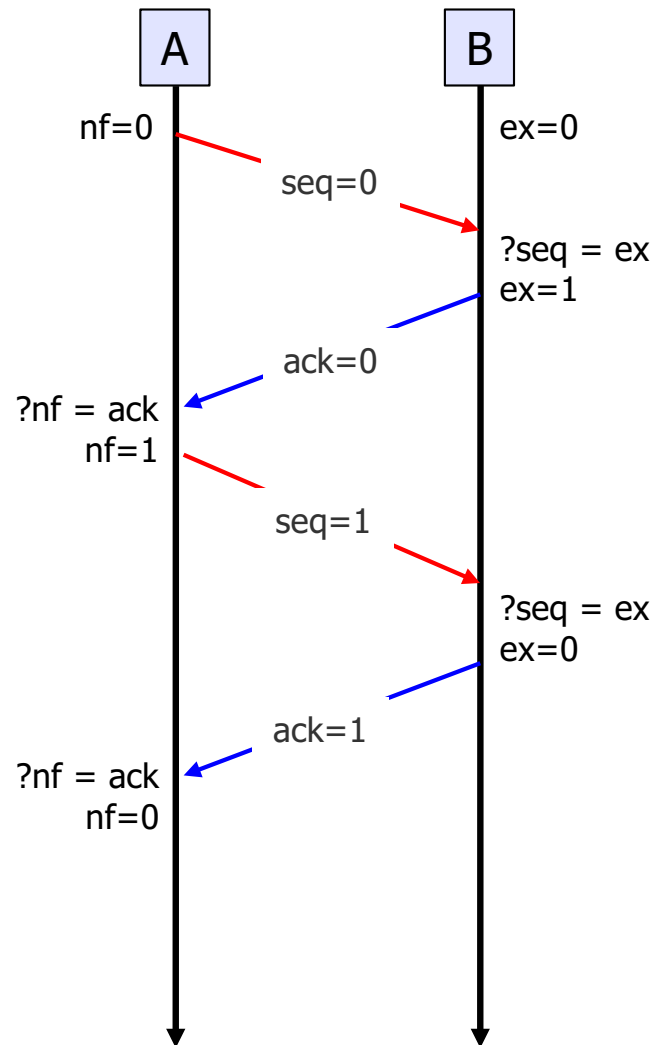
$S \in \{0 = 0\text{-frame}, 1 = 1\text{-frame}\}$

$R \in \{0 = \text{expects } 0\text{-frame}, 1 = \text{expects } 1\text{-frame}\}$

$C \in \{0 = 0\text{-frame}, 1 = 1\text{-frame}, A = \text{ack-frame}, - = \text{empty}\}$

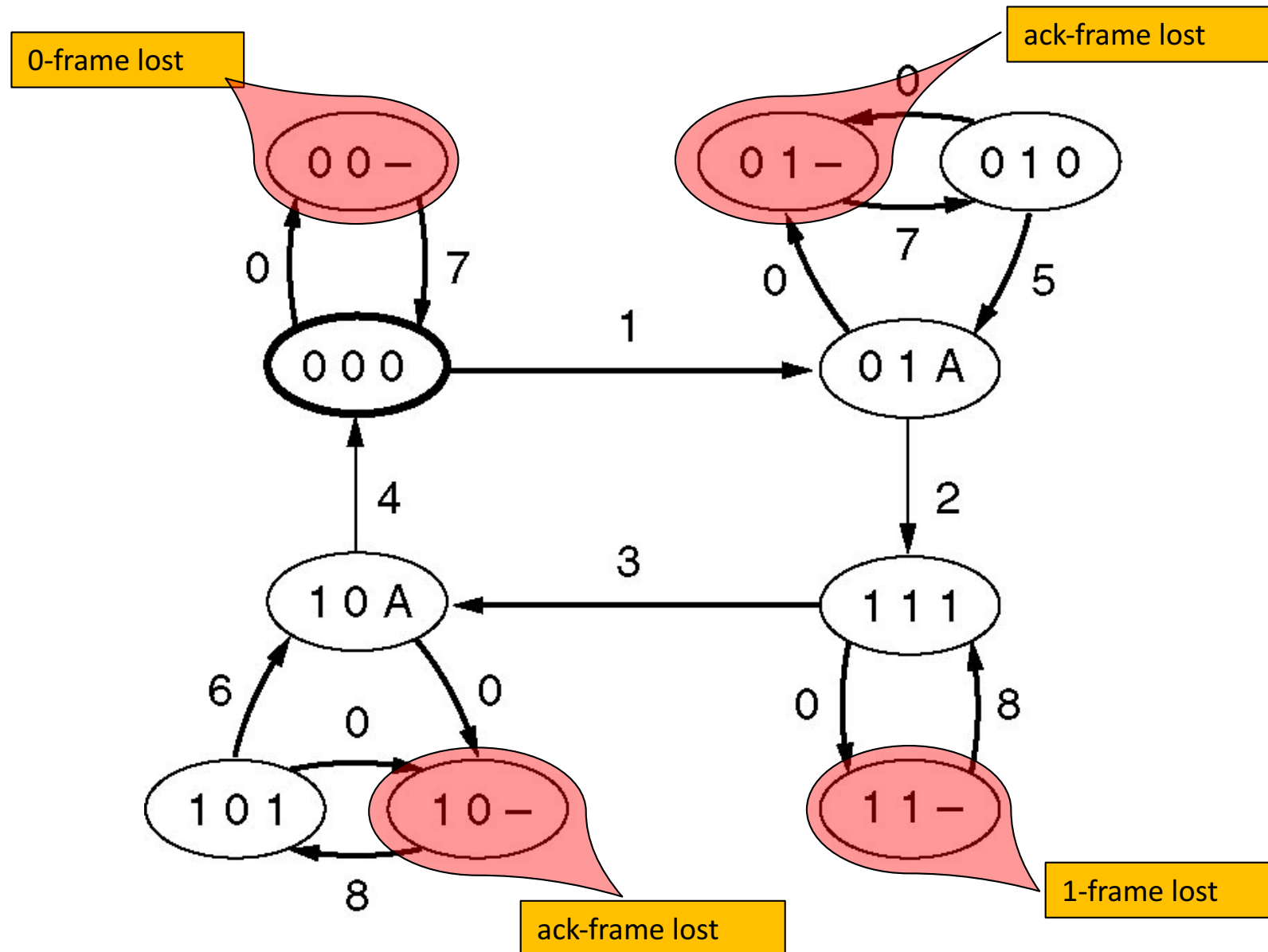
# Finite State Machine Models: Example Protocol 3

## Protocol 3 Normal operation





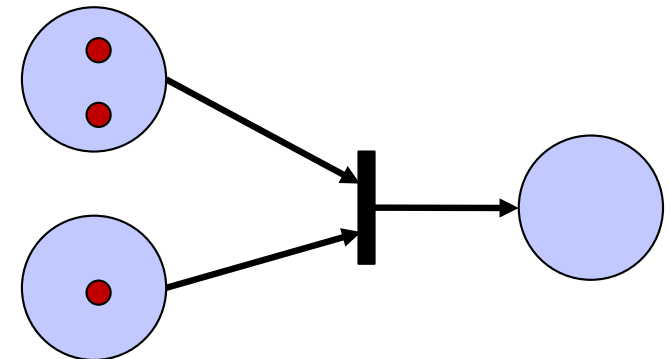
# Finite State Machine Models: Example Protocol 3



# Petri Net Models

---

- **A Petri Net has four basic elements**
  - Places
    - drawn as circles
    - represent a state of the system
  - Transitions
    - drawn as horizontal or vertical bar
    - have 0 or more inputs arcs
    - have 0 or more output arcs
  - Arcs
    - connect places and transitions
  - Tokens
    - drawn as filled circles
    - number of resources in state

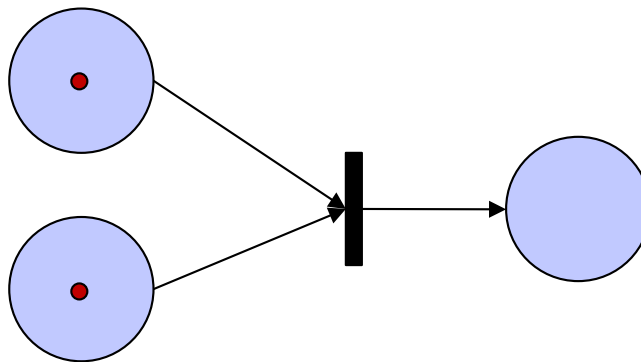


# Petri Net Models

---

- **Dynamics of Petri Nets**

- A transition is **enabled** if there are at least one token in each of its input places
- Any enabled transition may **fire** at will
  - Tokens are removed from input places and created in output places
- If several transitions are enabled any of them may fire
  - The choice of the firing transition is indeterminate



# Petri Net Models

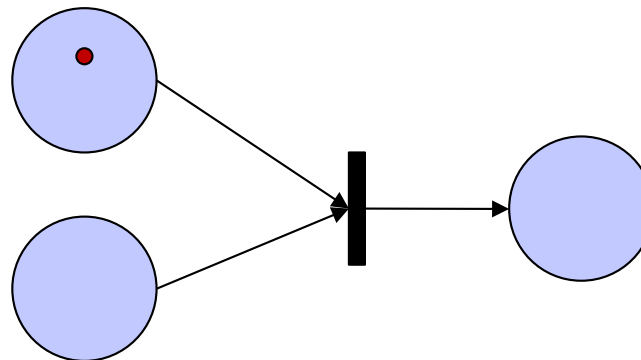
---

- A Petri Net is defined as a tuple

$$\text{PN} = (P, T, F, W, M_0)$$

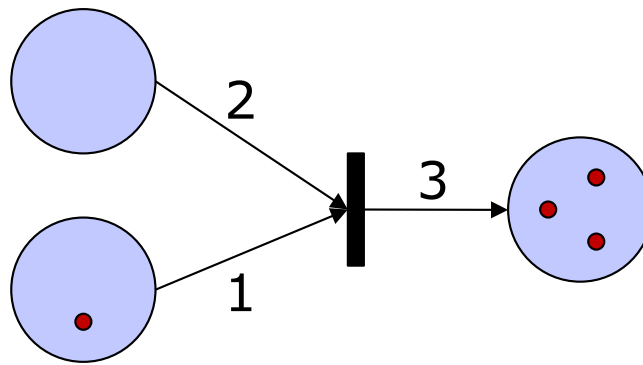
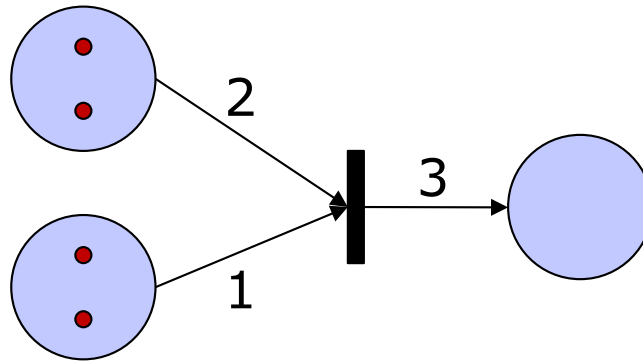
- with

- Places:  $P = \{P_1, P_2, \dots, P_m\}$
- Transitions:  $T = \{T_1, T_2, \dots, T_n\}$
- Arcs:  $F \subseteq (P \times T) \cup (T \times P)$
- Weight function:  $W: F \rightarrow \{1, 2, 3, \dots\}$
- Initial marking:  $M_0: P \rightarrow \{0, 1, 2, \dots\}$



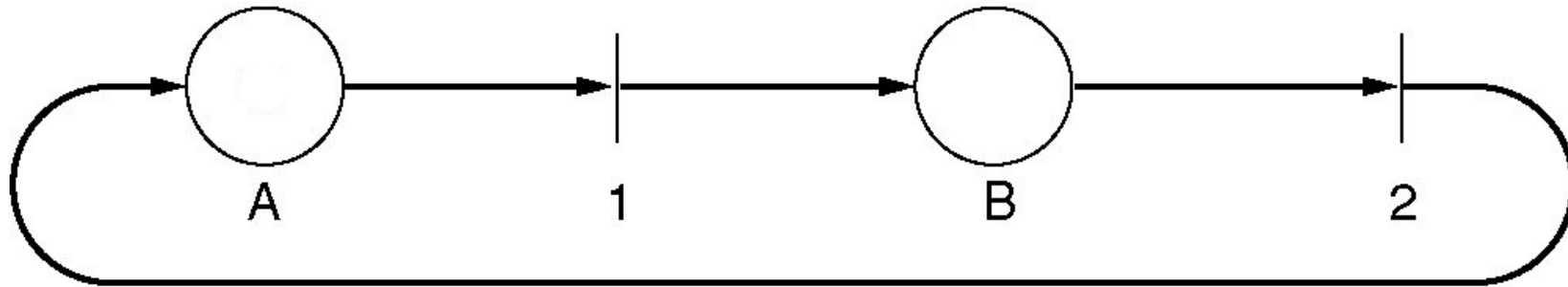
# Petri Net Models: Examples

---

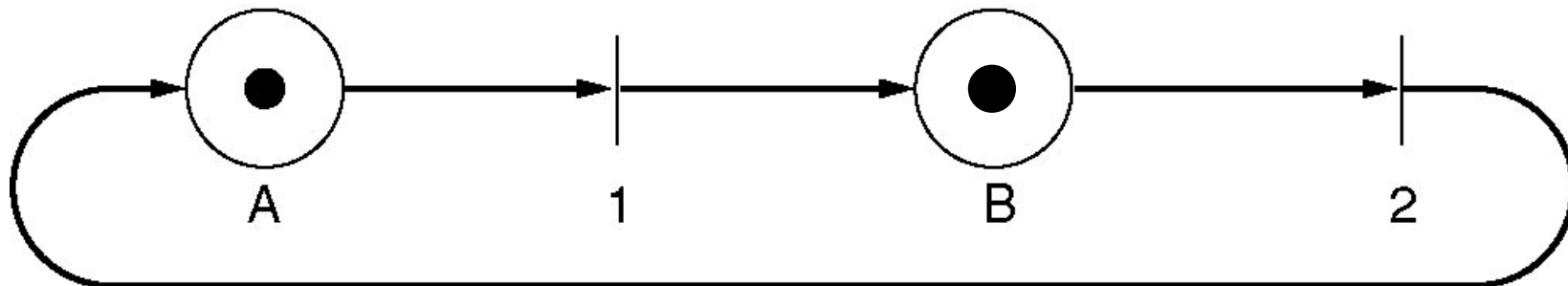


# Petri Net Models

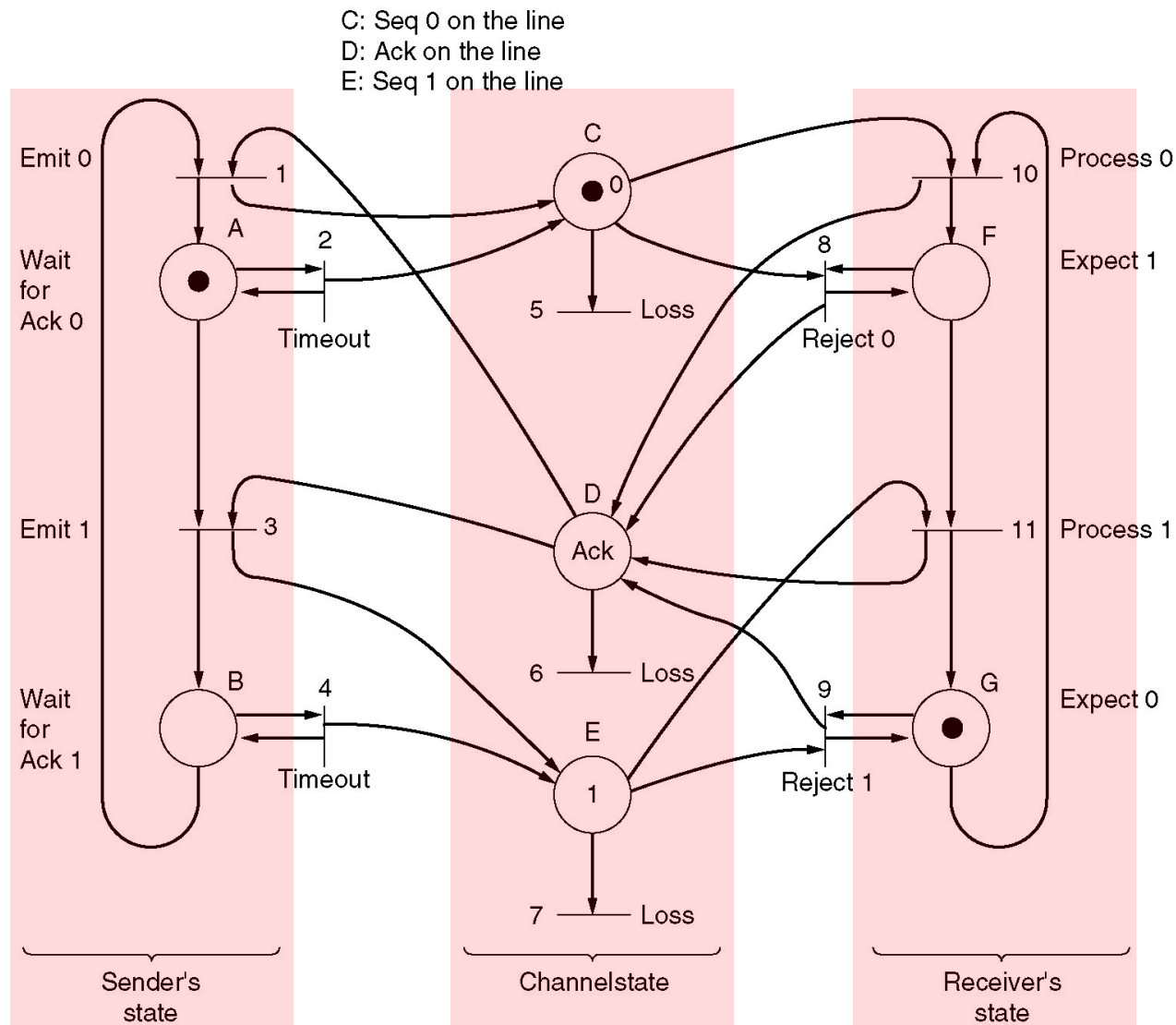
- A Petri net with two places and two transitions.
- Transition 1 is enabled, transition 2 is not



- After transition 1 has fired, transition 2 is enabled

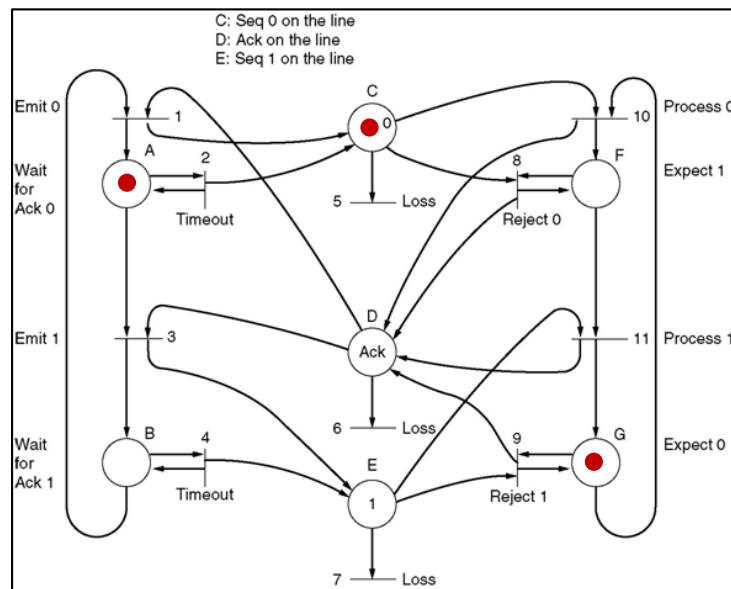


# Petri Net Models: Example Protocol 3

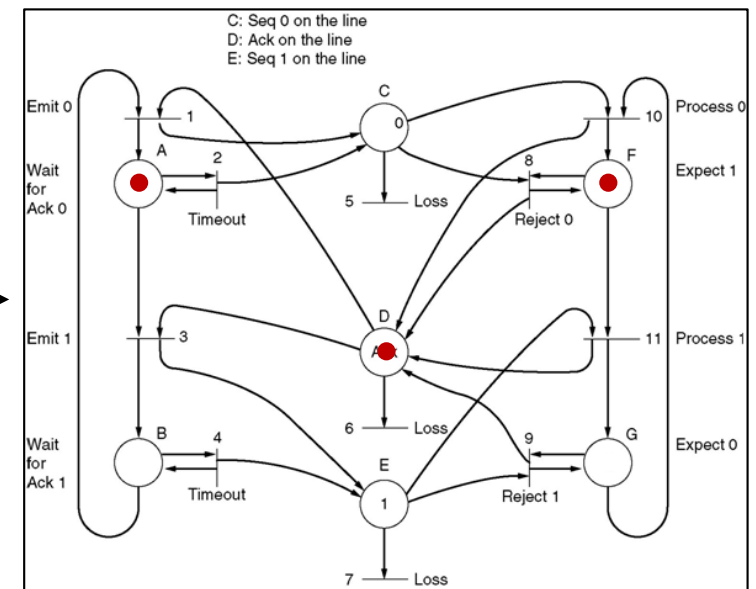


**A Petri net model for protocol 3.**

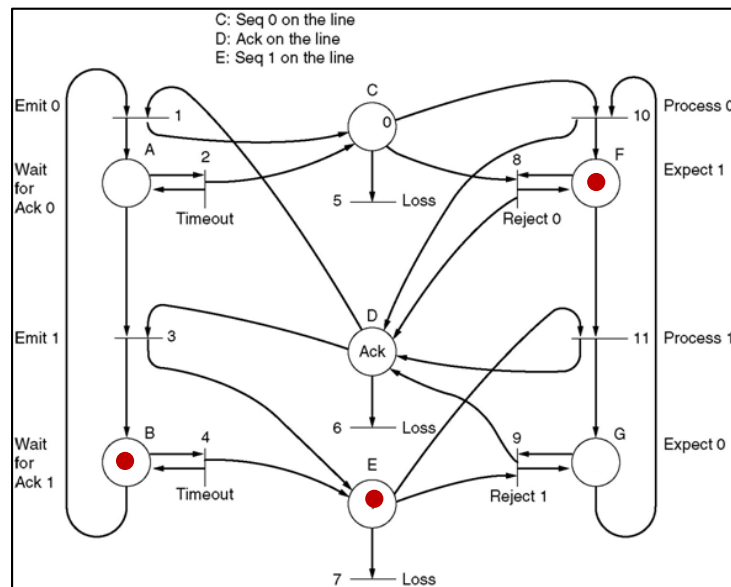
# Petri Net Models: Example Protocol 3



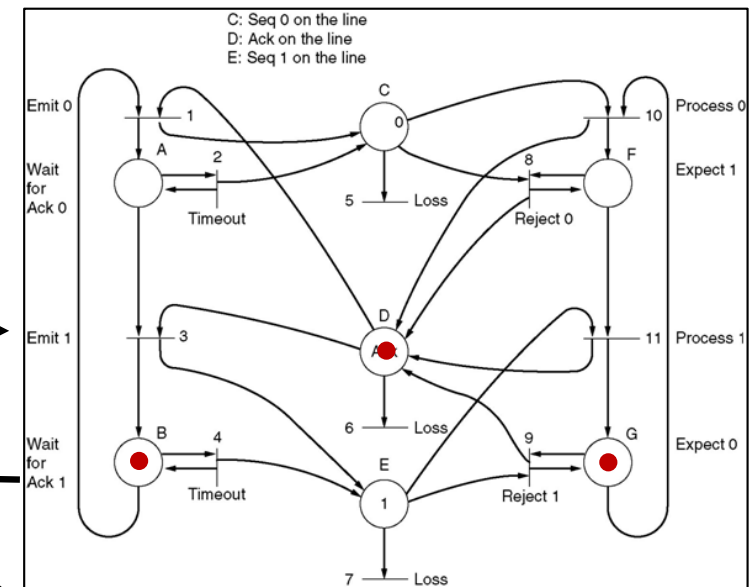
Transition 10



Transition 3



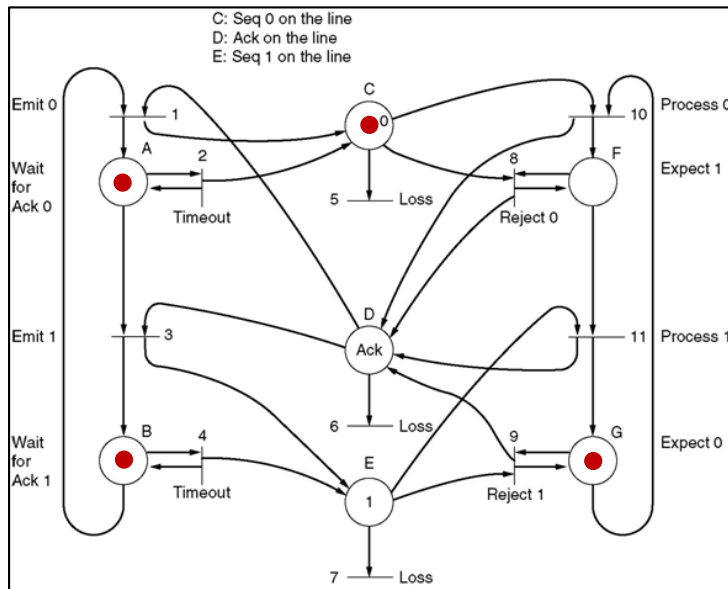
Transition 11



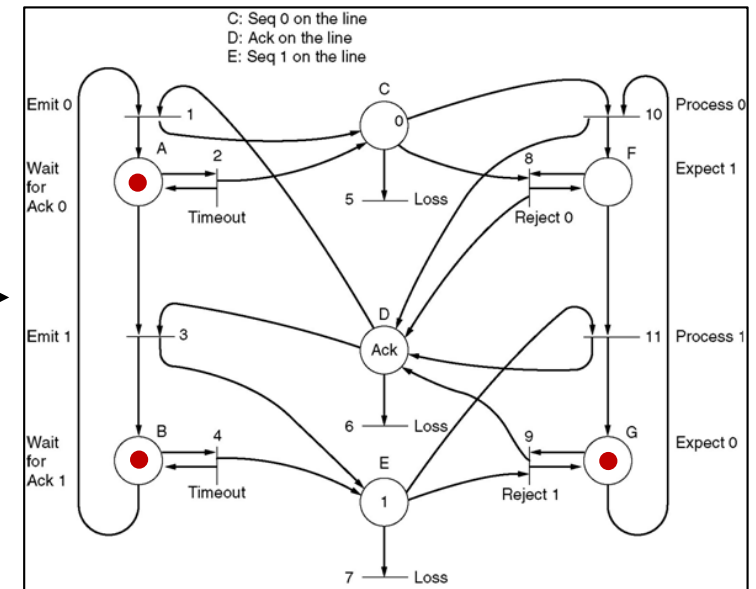
Transition 1  
(next slide)



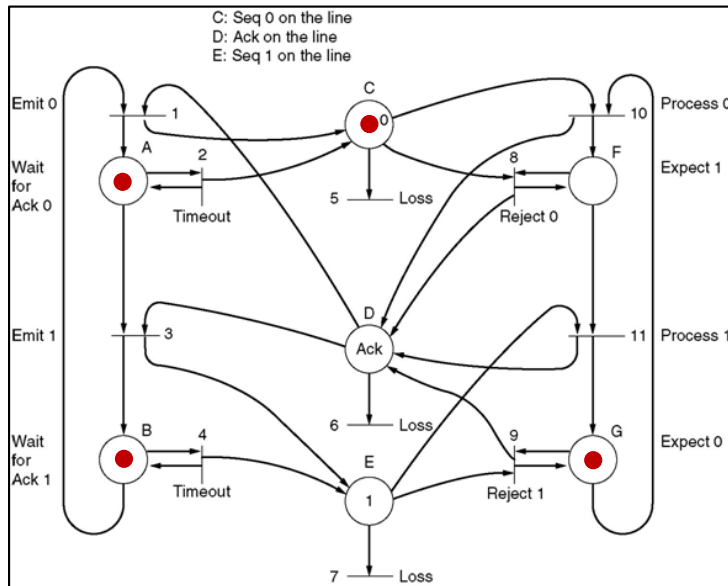
# Petri Net Models: Example Protocol 3



Transition 5



Transition 2



# Summary

---

- **The data link layer deals with networking of adjacent nodes**
- **The data link layer consists of two sublayers**
  - Logical Link Control (LLC)
  - Media Access Control (MAC)
- **Its main task is to provide services to the network layer**
- **This includes the delivery of error free frames, which contain payload from the network layer**
  - Transmission error
  - Flow control
- **Implementation of protocols is very complicated and weird**
- **Formal methods for their verification are needed**
  - Finite state machines
  - Petri nets