

# Link Layer & Error Control

Roman Dunaytsev

The Bonch-Bruевич Saint-Petersburg  
State University of Telecommunications

[roman.dunaytsev@spbgut.ru](mailto:roman.dunaytsev@spbgut.ru)

Lecture № 4

## 1 Link layer

- Framing
- SLIP
- PPP

## 2 Error control

- Echoplex
- Single parity check
- 2D parity check
- Internet checksum
- CRC
- BER and FER
- Checksum offloading
- FEC
- ARQ

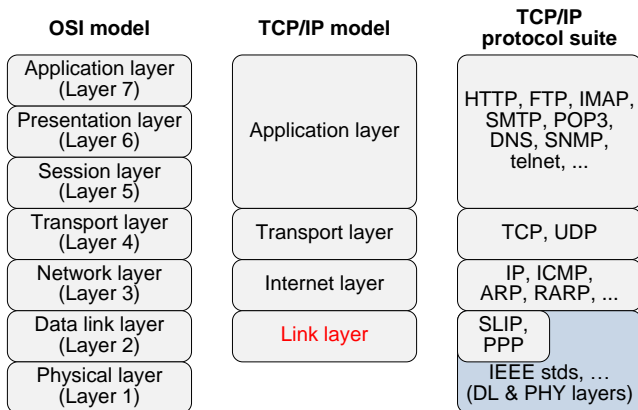
## 1 Link layer

- Framing
- SLIP
- PPP

## 2 Error control

- Echoplex
- Single parity check
- 2D parity check
- Internet checksum
- CRC
- BER and FER
- Checksum offloading
- FEC
- ARQ

- **The link layer** – the lowest layer in the TCP/IP model
  - Sometimes called **the data link layer**, **the network access layer**, or **the network interface layer**



## Link Layer (cont'd)

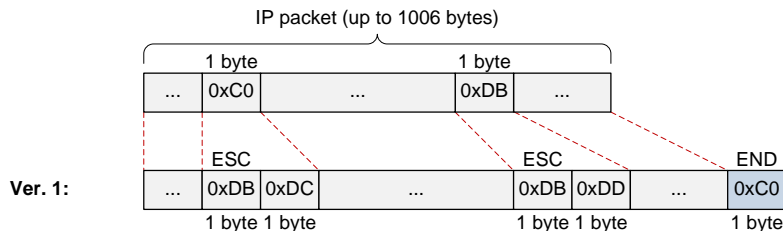
- The link layer transforms the physical layer, a raw transmission facility, to a communication link
- The link layer is responsible for communication between nodes that are **directly connected** by a physical medium
- **Functions of the link layer** :
  - **Framing** – inserts framing information into the transmitted stream to indicate the boundaries that define frames
  - **Error control** – ensures the integrity of the transmitted data
  - **Flow control** – prevents the sender from overloading the receiver
  - **Addressing** – inserts address or protocol type information to define the sender and/or receiver of the frame
  - **Media access control** – determines which device has control over the link at any given time
- Link layer protocols may include only some of these functions

## Link Layer (cont'd)

- The TCP/IP protocol suite supports many different link layer protocols, depending on the type of networking hardware being used:
  - IEEE 802.3: Ethernet
  - IEEE 802.11: Wi-Fi
  - IEEE 802.16: WiMAX
  - ANSI X3: Fiber Distributed Data Interface (FDDI)
  - etc.
- IETF standards cover only 2 link layer protocols:
  - RFC 1055 (June 1988): Serial Line Internet Protocol (SLIP)
  - RFC 1661 (July 1994): Point-to-Point Protocol (PPP)

- **Framing** is the process of sequencing data into standard patterns
- **3 types of data link protocols** :
  - Bit-oriented
  - Character-oriented
  - Byte count-oriented
- **Bit-oriented** – identify the beginning and end of a frame with a special sequence of bits called a **flag**
  - E.g., High-Level Data Link Control (HDLC), PPP
- **Character-oriented** – identify the beginning and end of a frame with special **characters**
  - E.g., Binary Synchronous Communications (Bisync or BSC), SLIP
- **Byte count-oriented** – identify the beginning of a frame with a special character followed by the precise **number of bytes** that the frame contains
  - E.g., Kermit (named after Kermit the Frog from The Muppets)

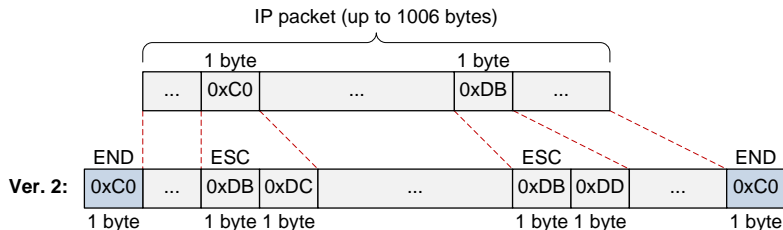
- **Serial Line Internet Protocol (SLIP)** – a character-oriented protocol for sending IP packets over point-to-point serial lines, either dedicated or dial-up
- SLIP provides just framing using **character stuffing**
- SLIP defines 2 special characters: **END** (0xC0) and **ESC** (0xDB)
  - Do not confuse the SLIP ESC character with ASCII ESC (0x1B)
  - The '0x...' notation is used to specify hexadecimal (HEX) values





# SLIP (cont'd)

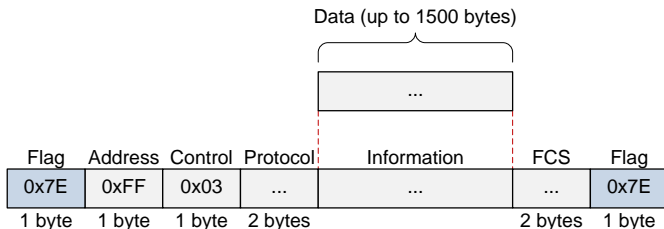
- SLIP works as follows:
  - 1 To send an IP packet, a SLIP sender simply starts sending the data in the packet
  - 2 If a data byte is the same as END, a 2-byte sequence of ESC and 0xDC is sent instead
  - 3 If a data byte is the same as ESC, a 2-byte sequence of ESC and 0xDD is sent instead
  - 4 When the last byte in the packet has been sent, END is then transmitted
- Most implementations transmit END at the beginning of packets too



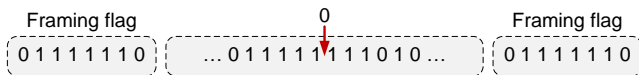
- **Point-to-Point Protocol (PPP)** – a bit-oriented protocol for sending IP and other types of packets over both asynchronous (e.g., dial-up) and synchronous (e.g., ISDN) serial lines
- PPP consists of 3 components:
  - A method for framing multi-protocol packets
  - A **Link Control Protocol (LCP)** for establishing, configuring, and testing the link connection
  - A family of **Network Control Protocols (NCPs)** for establishing and configuring different network layer protocols (e.g., RFC 1332: the Internet Protocol Control Protocol (IPCP))
- PPP uses the **High-Level Data Link Control (HDLC)** framing structure to encapsulate its data during transmission
  - This includes 2 fields that are needed in HDLC but not in PPP: **Address** and **Control**
  - They are maintained for backward compatibility but can be suppressed using **Address and Control Field Compression (ACFC)**

# PPP (cont'd)

- **Flag** – each frame begins and ends with 01111110
- **Address** – always 11111111 (means 'all stations')
- **Control** – always 00000011
- **Protocol** – identifies the data encapsulated in the information field of the frame (e.g., 0x0021 = IP packet, 0x002B = Novell IPX packet, 0xC021 = LCP data, 0x8021 = NCP data for IP)
- **FCS** – frame check sequence to detect errors in the frame



- Since 01111110 is the framing flag, PPP needs to escape this pattern when it appears in the information field
- On **asynchronous** links this is done by using **character stuffing**:
  - Similar to SLIP, PPP uses a special **escape character** (0x7D)
  - If a data byte is the same as the flag (0x7E), a 2-byte sequence of 0x7D and 0x5E is sent instead
  - If a data byte is the same as the escape (0x7D), a 2-byte sequence of 0x7D and 0x5D is sent instead
- On **synchronous** links this is done by using **bit stuffing**:
  - An extra '0' after every 5 '1's in the information field is inserted
  - The receiver can now correctly detect the structure enforced by the flags by inspecting the first bit after every 5 '1's: if it is a '0' it must be deleted, otherwise it is a true flag



- PPP works as follows:
  - ① In order to establish communications over a point-to-point link, each end of the PPP link must first send LCP frames to configure and test the link
  - ② After the link has been established and optional facilities have been negotiated as needed by LCP, PPP must send NCP frames to choose and configure one or more network layer protocols (e.g., IP, Novell IPX)
  - ③ Once each of the chosen network layer protocols has been configured, packets from each network layer protocol can be sent over the link
  - ④ The link will remain configured for communications until explicit LCP or NCP frames close the link down, or until some external event occurs (e.g., an inactivity timer expires)
- Due to LCP/NCP negotiations, PPP supports **full duplex** links only
- PPP replaces SLIP as the protocol of choice for point-to-point connections

# PPP (cont'd)

<b>Feature</b>	<b>SLIP</b>	<b>PPP</b>
Compression	None; Compressed SLIP (CSLIP) added later as an option	Automatically negotiated as a part of the connection setup
Connection configuration	Manual	Automatic; IP configuration is a part of the connection setup and is transparent to the user
Authentication	None	Supports the Password Authentication Protocol (PAP) and the Challenge Handshake Authentication Protocol (CHAP)
Protocol handling	IP only	Multi-protocol handling on a single serial connection
Error detection	None	Built-in
Industry support	None	Widespread industry support

## 1 Link layer

- Framing
- SLIP
- PPP

## 2 Error control

- Echoplex
- Single parity check
- 2D parity check
- Internet checksum
- CRC
- BER and FER
- Checksum offloading
- FEC
- ARQ

- Why error control?
- Computer applications require **precise** instructions to operate correctly
- However, due to unreliable networking **protocol data units (PDUs)** may be lost, duplicated, misrouted, excessively delayed, or delivered out of order
- Moreover, once data are sent over the transmission medium the characteristics of the medium affect the transmitted data in various ways so that the **signals** received at the remote end of a link differ from the transmitted signals
  - These adverse characteristics of a physical transmission medium are known as **transmission impairments**
  - In case of binary data, transmission impairments may lead to errors: '0's are transformed into '1's and vice versa



- **3 main transmission impairments** :
  - Attenuation
  - Distortion
  - Noise
- **Attenuation** – when a signal is transmitted through a communication channel, its amplitude decreases
- **Distortion** – when a signal is transmitted through a communication channel, its shape changes
  - **Amplitude distortion** – since the communication channel is limited to certain frequencies, the output amplitude is not a linear function of the input amplitude under certain conditions
  - **Delay distortion** – since different signal frequencies travel at different velocities, they arrive at the receiver at slightly different times
- **Noise** – the presence in a communication channel of random, unwanted signals

# Error Control (cont'd)

- The effect of transmission impairments on a data transmission is to introduce errors
- The effect of transmission impairments can be remedied (to a certain extent) with cable insulation, hardware compensation filters, etc.
- The errors that remain must be caught by communication protocols
  - Similarly to quality control on production lines



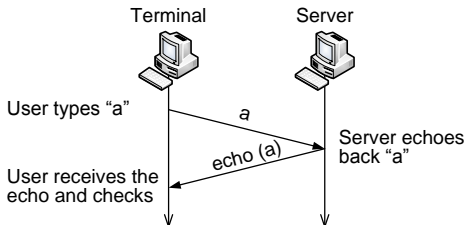
# Error Control (cont'd)

- Since errors are unavoidable in data communication systems, it is necessary to introduce some form of **error control**
- The first step in any form of error control is to **detect errors** if any
- Having detected the presence of errors, there are 2 strategies commonly used to **correct errors**:
  - **Feedback error control** – a message is returned to the sender indicating that errors have occurred and requesting a retransmission of the data
  - **Forward error correction** – further computations are carried out at the receiver to correct errors
- Note that some applications are relatively **error-tolerant** without degrading the end-user perception (e.g., VoIP)
- However, most Internet applications and services require error-free data delivery and proper sequencing (e.g., WWW)

# Error Control (cont'd)

- **Error detection** – a method that allows some bit errors to be detected
- The data are encoded so that the encoded data contain some **additional redundant information** about the data
  - If there is no redundancy, then it is impossible to distinguish between correct and incorrect messages
- The data are decoded so that the additional redundant information must match the original information
- 2 fundamental observations:
  - All error-detection schemes will **fail to detect some errors**
  - In all error-detection schemes, **redundancy must be added** to the message to enable faulty received messages to be distinguished from correct messages

- **Echoplex** (aka **echoing**) has been used in early systems with keyboard data entry
  - In such systems, a character displayed at a terminal is not obtained directly from the keyboard
  - Instead, each character is transmitted to the receiver, where it is sent back (echoed)
  - Error detection and correction is the responsibility of the user, who compares displayed characters with those entered (or supposedly entered)
- Echoplex requires a **full duplex** link to allow the echo to come back



## Echoplex (cont'd)

- If a full duplex configuration is not available, a device is usually switched to **local echo**, which sends the echo through the local modem back to the user device
  - However, local echo does not allow to check for errors across the link
- Although echoing is reasonable for error control in simple systems, it has serious **shortcomings**:
  - Echoplex relies heavily on human detection of errors
  - Echoing does not work well on long distance connections
  - A very low throughput

# Single Parity Check

- **Parity checking** – the parity bit is used to ensure that the transmitted message (which includes the parity bit) has either an odd or even number of '1's, depending on whether odd parity or even parity scheme has been selected
  - **Odd parity**: should be odd number of '1's
  - **Even parity**: should be even number of '1's
- The receiver must be set to the same parity scheme as the sender
- To correct the error, the message (e.g., 7 bits of an ASCII character plus the parity bit) must be sent again
  - The American Standard Code for Information Interchange (ASCII) is a character encoding based on the English alphabet
- **2 types of parity check** :
  - Single parity check
  - 2-dimensional parity check

# Single Parity Check (cont'd)

- **Single parity check** – adding a single parity bit to each string of bits that comprise a character
  - It is the simplest parity check scheme
- The parity bit is inserted at the sender, sent with each character in the message, and checked at the receiver to determine if each character has the correct parity
  - If transmission impairments caused a bit flip of '1' to '0' or '0' to '1', the parity check would indicate this
- The parity bit (PB) is calculated as the modulo 2 sum of the data bits:

$$PB_{\text{even}} \equiv d_1 + d_2 + \cdots + d_k = \sum_{i=1}^k d_i \pmod{2}$$

$$PB_{\text{odd}} \equiv PB_{\text{even}} + 1 \pmod{2}$$



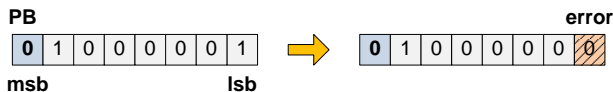
# Single Parity Check (cont'd)

- **Modulo 2 sum** is based on the binary addition with no carries (aka exclusive OR (XOR)):
  - $0 + 0 \equiv 0 \pmod{2}$
  - $0 + 1 \equiv 1 \pmod{2}$
  - $1 + 0 \equiv 1 \pmod{2}$
  - $1 + 1 \equiv 0 \pmod{2}$
- **Modulo arithmetic** (aka **clock arithmetic**) – a system of arithmetic for integers, where numbers 'wrap around' after they reach a certain value - **the modulus**
  - E.g., since the hour number starts over when it reaches 12, this is arithmetic modulo 12



# Single Parity Check (cont'd)

- Consider sending the ASCII character 'A' (1000001) using even parity
- The parity bit should be set to 0 to make the overall number of '1's an even number
- Setting the parity bit in **the most significant bit (msb)** position gives the 8-bit message
- Assume **the least significant bit (lsb)** is corrupted and received in error
- The number of '1's is now an odd so an error must have occurred





# Single Parity Check (cont'd)

- A **single** transmission error causes recomputed parity at the receiver to be invalid, so the error is detected
  - Which particular bit is wrong is not known by the receiver, so the character must be sent again to correct the error
- A **double** error, however, results in the same parity and cannot be detected
- More generally, **any odd number of errors can be detected** but **any even number cannot be detected**
  - This is true for either odd or even parity
- Single parity check will detect virtually all random, **independently occurring errors**
  - This is because in practice the probability of a single error, although very small, is generally much greater than the probability of 2 errors

# Single Parity Check (cont'd)

- However, there is another class of errors, called **burst errors**, in which the occurrence of errors is not independent
  - The burst starts with the first bit which is an error and ends with the last bit in error
  - Between the start and finish the bits will be random, so it can be expected that half of them will be correct
  - Since the number of errors in the burst is equally likely to be odd or even, single parity check will **only detect half of such burst errors**
- Consider the following cases:

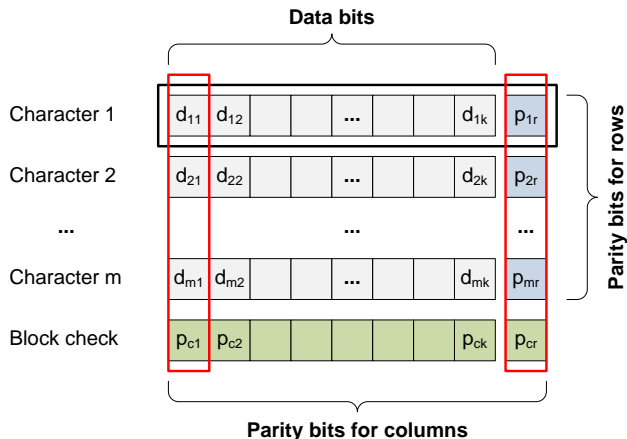
Original (even parity):	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	1	0	1	0	1	0	1	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1											
1	0	1	0	1	0	1	0											
Burst error (all "0" to "1"):																		
Result:	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	1	1	1	1	1	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1											
1	0	1	1	1	1	1	1											
Parity check:	Ok (even)	Fails (not even)																
Errors:	Not detected	Detected																

# Single Parity Check (cont'd)

- Single parity check is not very efficient for long messages
  - Since probability of multiple errors increases with the message length
- Nevertheless, single parity check provides a remarkable amount of error-detection capability
  - The addition of just 1 check bit results in making half of all possible error patterns detectable, regardless of the number of information bits
- Stronger error detection is possible with 2-dimensional parity check
  - 2-dimensional parity check increases the likelihood of detecting burst errors

# 2D Parity Check

- **2-dimensional parity check** – a refinement of single parity check
  - In addition to a parity bit on each character, it places a parity (odd or even) on a **block of characters**



## 2D Parity Check (cont'd)

- 2-dimensional parity check **can potentially correct single errors**, located by the intersection of a row and a column with invalid parity
- Double errors in any row or column can be only detected
- Even numbers of errors in each row and even numbers of errors in each column cannot be detected

1	0	0	1	0	0
0	0	0	0	0	1
1	0	0	1	0	0
1	1	0	1	1	0
1	0	0	1	1	1

Located 1  
out of 1 error

1	0	0	1	0	0
0	0	0	0	0	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	0	1	1	1

Detected 2  
out of 2 errors

1	0	0	1	0	0
0	0	0	1	0	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	0	1	1	1

Detected 2  
out of 3 errors

1	0	0	1	0	0
0	0	0	1	0	1
1	0	0	1	0	0
1	0	0	0	1	0
1	0	0	1	1	1

Detected 0  
out of 4 errors

- **Checksumming** – a simple error-detection scheme whereby each message is accompanied by a value based on the number of bits in the message
  - Checksums have no ability to correct errors, but only detect them
- The receiver then applies the same formula to the message and checks to make sure the value is the same
  - If the value matches, it is assumed that the message was received correctly
  - If not, the receiver can assume that the message was corrupted
- Checksums take on various forms
  - Usually 8, 16, or 32 bits in size



# Internet Checksum (cont'd)

- Several TCP/IP protocols (IPv4, ICMP, UDP, and TCP) use the **Internet checksum** algorithm to detect errors
  - It has no ability to correct errors, but only detect them
  - RFC 1071 'Computing the Internet checksum'
- Internet checksum works as follows:
  - 1 The checksum field itself is filled with '0's
  - 2 Adjacent bytes to be checksummed are paired to form 16-bit words
  - 3 The modulo ( $2^{16} - 1 = 65,535$  or  $0xFFFF$ ) sum is computed over these 16-bit words
  - 4 The 1's complement (i.e., bitwise NOT) of this sum is placed in the checksum field
  - 5 Then, to verify the checksum, the modulo ( $2^{16} - 1$ ) sum is computed over the same set of bytes, including the checksum field
  - 6 If the result is all '1's (i.e., '-0' in 1's complement arithmetic), the check succeeds
- **Bitwise NOT** = all '0's become '1's and vice versa

# Internet Checksum (cont'd)

- Consider the 4 bytes: 0x7B, 0xF7, 0xCD, 0x1C
- They can be paired to form 2 16-bit words as 0x7BF7 and 0xCD1C (i.e., 01111011 11110111 and 11001101 00011100)
- The modulo 0xFFFF sum is computed over these bytes
  - Any carry arising from the summing is added back to the total

$$\begin{array}{r} 0111101111110111 \\ \underline{1100110100011100} \\ 10100100100010011 \end{array} \quad \nearrow \quad \begin{array}{r} 0100100100010011 \\ \underline{0000000000000000} \\ 0100100100010100 \end{array}$$

- The 1's complement of this sum (bitwise NOT) is:

$$\begin{array}{r} \underline{0100100100010100} \\ 1011011011101011 \end{array}$$

- Thus, the checksum is 10110110 11101011 (or 0xB6EB):
  - $0x7BF7 + 0xCD1C + 0xB6EB \equiv 0xFFFF \pmod{0xFFFF}$

- **Cyclic Redundancy Check (CRC)** – one of the most common (and one of the most powerful) error-detection schemes
- In CRC, the informational symbols are represented by **polynomials** with binary coefficients
- I.e., the  $k$  information bits are used to form the informational polynomial of degree  $(k - 1)$ :

$$(b_{k-1}, b_{k-2}, \dots, b_1, b_0) \Rightarrow b_{k-1}x^{k-1} + b_{k-2}x^{k-2} + \dots + b_1x + b_0$$

- E.g., a 4-bit block gives an informational polynomial of degree 3:

$$(1, 0, 0, 1) \Rightarrow k = 4 \Rightarrow 1 * x^3 + 0 * x^2 + 0 * x^1 + 1 * x^0 = x^3 + 1$$

- CRC works as follows:
  - ① Given a  $k$ -bit block of bits, the sender generates a  $n$ -bit sequence, known as a **Frame Check Sequence (FCK)**, so that the resulting frame, consisting of  $(k + n)$  bits, is exactly divisible by some predefined **generator polynomial** of degree  $n$  (i.e.,  $(n + 1)$ -bit in size)
  - ② To do this, the original block is shifted to the left by  $n$  bits and is padded out with '0's
  - ③ Then the FCS is obtained by dividing this new sequence by the generator polynomial and using the remainder as the FCS, while the quotient is discarded
  - ④ The receiver then divides the incoming frame of  $(k + n)$  bits by the generator polynomial and if there is no remainder (the remainder is 0), assumes there was no error

# CRC (cont'd)

- All calculations are based on **polynomial arithmetic modulo 2**
  - We use 'modulo 2' because polynomial coefficients may be only 1 or 0
- Polynomial arithmetic modulo 2 uses binary addition with no carries (XOR)
  - Note that addition and subtraction give the same result in modulo 2

$$\begin{array}{r} 1001 \\ +0101 \\ \hline 1100 \end{array}$$

$$\begin{array}{r} 1001 \\ -0101 \\ \hline 1100 \end{array}$$

$$\begin{array}{r} 11001 \\ \times \quad 11 \\ \hline 11001 \\ +110010 \\ \hline 101011 \end{array}$$

# CRC (cont'd)

- 4-bit message  $M$ :  $(1, 1, 0, 0) \Rightarrow x^3 + x^2$ ,  $k = 4$
- Generator polynomial:  $(1, 0, 1, 1) \Rightarrow x^3 + x + 1$ ,  $n = 3$
- Message  $M$  is shifted to the left by  $n$  bits:  
 $2^n M \Rightarrow (1, 1, 0, 0, 0, 0, 0) \Rightarrow x^3(x^3 + x^2) = x^6 + x^5$

$$\begin{array}{r} x^6 + x^5 \quad | \underline{x^3 + x + 1} \\ \underline{x^6 + x^4 + x^3} \quad | x^3 + x^2 + x \\ x^5 + x^4 + x^3 \\ \underline{x^5 + x^3 + x^2} \\ x^4 + x^2 \\ \underline{x^4 + x^2 + x} \\ x = \text{FCS} \end{array}$$

- Frame to transmit ( $k + n = 7$  bits):  $(1, 1, 0, 0, 0, 1, 0) \Rightarrow x^6 + x^5 + x$

## CRC (cont'd)

- Frame received (7 bits):  $(1, 1, 0, 0, 0, 1, 0) \Rightarrow x^6 + x^5 + x$
- Generator polynomial:  $(1, 0, 1, 1) \Rightarrow x^3 + x + 1$
- Frame is divided by generator polynomial; remainder should be 0:

$$\begin{array}{r} x^6 + x^5 + x \quad | \underline{x^3 + x + 1} \\ \underline{x^6 + x^4 + x^3} \quad | x^3 + x^2 + x \\ x^5 + x^4 + x^3 + x \\ \underline{x^5 + x^3 + x^2} \\ x^4 + x^2 + x \\ \underline{x^4 + x^2 + x} \\ 0 = \text{remainder} \end{array}$$

- Check is Ok

# CRC (cont'd)

- 8-bit message  $M$ :  $(1, 0, 0, 1, 1, 0, 1, 0) \Rightarrow x^7 + x^4 + x^3 + x, \quad k = 8$
- Generator polynomial:  $(1, 1, 0, 1) \Rightarrow x^3 + x^2 + 1, \quad n = 3$
- $(1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0) \Rightarrow x^3(x^7 + x^4 + x^3 + x) = x^{10} + x^7 + x^6 + x^4$

```
10011010000 |1101
1101          |11111001
1001
1101
1000
1101
1011
1101
1100
1101
1000
1101
101 = FCS
```

- Frame to transmit ( $k + n = 11$  bits):  $(1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1)$



# CRC (cont'd)

- Frame received (11 bits), lsb in error: (1, 0, 0, 1, 1, 0, 1, 0, 1, 0, **0**)
- Generator polynomial: (1, 1, 0, 1)
- Frame is divided by generator polynomial; remainder should be 0:

```
10011010100 |1101
1101         |11111001
1001
1101
1000
1101
1011
1101
1100
1101
1100
1101
1100
1101
1 ⇒ remainder ≠ 0
```

- Check fails, error detected

# CRC (cont'd)

- Some standard generator polynomials:

- **CRC-8-CCITT**:  $x^8 + x^2 + x + 1$

- **CRC-16**:  $x^{16} + x^{15} + x^2 + 1$

- **CRC-CCITT**:  $x^{16} + x^{12} + x^5 + 1$

- **CRC-32 (aka CRC-CCITT)**:

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

- The CRC algorithm, while seemingly complex, is easily implemented in hardware using shift register and XOR gates
- When the generator polynomial is selected carefully, the probability that the CRC algorithm cannot detect an error is very low
  - E.g., CRC-12 detects 99.97% of errors with a length 12 or more

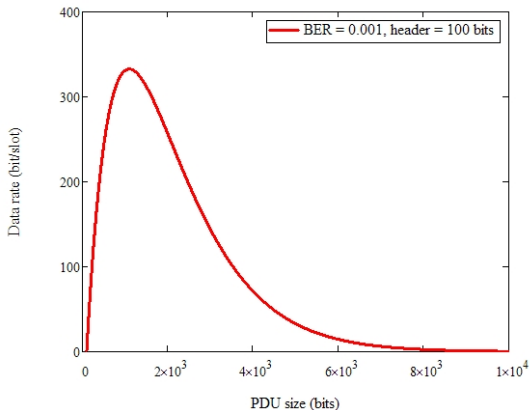
- In telecommunications, an **error rate** is the ratio of the number of bits/elements/characters/PDUs/etc. incorrectly received to the total number of bits/elements/characters/PDUs/etc. sent during a specified time interval
- **Bit Error Rate (BER)** = Erroneous bits / Total number of bits
- PDU loss rate = Lost PDUs / Total number of PDUs
- The number of errors caused by data transmission is typically orders of magnitude larger than the number of errors caused by hardware failures within a computer system
  - For optical fiber links, BER is usually below  $10^{-9}$
  - For internal circuits, BER is usually below  $10^{-15}$

- **Frame Error Rate (FER)** can be obtained from BER as follows:
  - Probability of a bit being error-free =  $(1 - \text{BER})$
  - Probability of a frame of length  $N$  being error-free =  $(1 - \text{BER})^N$
  - $\text{FER} = 1 - (1 - \text{BER})^N$
- What is the effect of altering the length of frames?
  - An increased frame size implies a smaller frame overhead
  - But larger frames are more vulnerable to bit errors
- Consider transmission of 1 PDU per time slot, no retransmissions,  $\text{BER} = 10^{-3}$ , header plus trailer (if any) length = 100 bits, total PDU length  $N = [101, \dots, 10000]$  bits

$$\text{Data rate}_{\text{effective}} = \frac{(N - \text{header})}{\text{slot}} * (1 - \text{BER})^N$$

# BER and FER (cont'd)

- The choice of an optimum PDU size is particularly important for long-range communications in the presence of a high BER



# Checksum Offloading

- The checksum calculation might be done in software or in hardware
- Higher-level protocols (e.g., TCP, UDP, IPv4) calculate checksums themselves and then pass the completed packet to the hardware
- Recent network adapters can provide advanced features such as Internet checksum calculation, also known as **checksum offloading**
- I.e., a higher-level protocol driver will not calculate the checksum itself but will simply pass an empty ('0's or garbage filled) checksum field to the hardware

## Checksum Offloading (cont'd)

- **Wireshark**, a free network protocol analyzer, validates the checksums of several protocols (TCP, UDP, IPv4, etc.)
  - [www.wireshark.org](http://www.wireshark.org)
- It does the same calculation as a 'normal receiver' would do, and shows the checksum fields in the packet details with a comment
  - E.g., [correct], [incorrect, should be 0xd326], etc.
- However, note:
  - The Ethernet transmitting hardware calculates the Ethernet CRC-32 checksum and the receiving hardware validates this checksum
  - If the received checksum is wrong, Wireshark will not even see the packet, as the Ethernet hardware internally discards the packet

## Checksum Offloading (cont'd)

- Checksum offloading often causes confusion as the network packets to be transmitted are handed over to Wireshark before the checksums are actually calculated
- Wireshark gets these 'empty' checksums and displays them as invalid, even though the packets will contain valid checksums when they leave the network hardware later
- Checksum offloading can be confusing and having a lot of 'invalid' messages on the screen can be quite annoying
- You can do 2 things to avoid this checksum offloading problem:
  - Turn off the checksum offloading in the network driver, if this option is available
  - Turn off checksum validation of the specific protocol in the Wireshark preferences (**Recommended**)



# Checksum Offloading (cont'd)

The image shows a Wireshark capture of a network packet. The packet list pane shows several TCP segments, with the selected one (Frame 18) having a [TCP CHECKSUM INCORRECT] warning. The packet details pane shows the following information:

- Frame 18 (54 bytes on wire, 54 bytes captured)
- Ethernet II, Src: Intel\_0a:87:8a (00:07:e9:0a:87:8a), Dst: ExtremeN\_19:82:d0 (00:04:96:19:82:d0)
- Internet Protocol, Src: 195.148.186.165 (195.148.186.165), Dst: 217.70.129.242 (217.70.129.242)
- Transmission Control Protocol, Src Port: opswagent (3976), Dst Port: http (80), Seq: 411, Ack: 11
- Source port: opswagent (3976)
- Destination port: http (80)
- Sequence number: 411 (relative sequence number)
- Acknowledgement number: 11681 (relative ack number)
- Header length: 20 bytes
- Flags: 0x10 (ACK)
- window size: 65535
- Checksum: 0xd98d [incorrect, should be 0xb0ee (maybe caused by "TCP checksum offload"?)]
- [SEQ/ACK analysis]

The packet bytes pane shows the following hex and ASCII data:

```
0000 00 04 96 19 82 d0 00 07 e9 0a 87 8a 08 00 45 00 .....E.
0010 00 28 01 93 40 00 80 06 1f ca c3 94 ba a5 d9 46 .(.@... ..F
0020 81 f2 0f 88 00 50 ab 62 95 d9 59 d1 7a 8d 50 10 .....P.b ..Y.Z.P.
0030 ff ff d9 8d 00 00 .....
```

- **Forward Error Correction (FEC)** – adding of redundant information embedded in the data set so the receiver can detect errors and **correct** them without requiring a retransmission
  - FEC imposes a greater bandwidth overhead than feedback error control, but is able to recover from errors more quickly
- Adding more check bits reduces the amount of available bandwidth, but also enables the receiver to correct more errors
  - BCH codes, bits: (total length; payload; correcting capability)
  - E.g., (255, 131, 18) and (255, 87, 26)
- FEC tends to be most useful when:
  - Errors are quite probable (satellite and mobile networks)
  - Retransmissions are impractical or simply impossible (satellite and multicast-based content distribution systems)
- FEC is not restricted to communications but can also be found in storage applications (CDs, DVDs, HDDs)

- Even very powerful error-correcting code may not be able to correct all errors that arise in a communication channel
- Consequently, many data communication links provide a further error-control mechanism – **feedback error control** – in which errors in data are detected and the data are **retransmitted**
- This is known as **Automatic Repeat reQuest (ARQ)** and it involves using redundant information embedded in the data to detect errors at the receiver and then returning a message to the sender requesting the retransmission of a PDU (or PDUs)
- In contrast to FEC, ARQ uses extra bandwidth mainly when PDUs are retransmitted but introduces variable delay (aka **jitter**) due to these retransmissions
- The shortcomings of FEC and ARQ could be overcome if they are used in combination (aka **hybrid ARQ (HARQ)**)

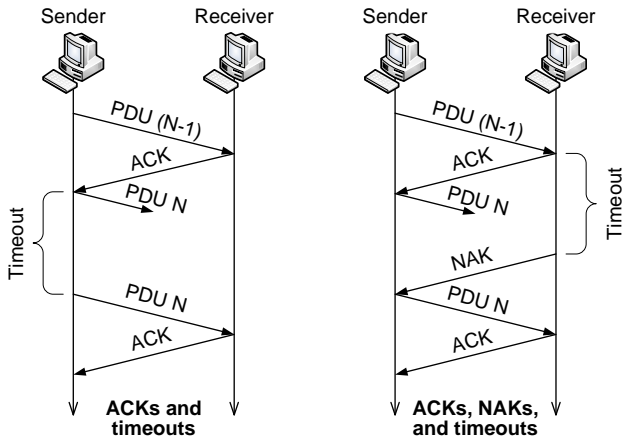
- ARQ schemes are based on some or all of the following components:
- **Error detection**
  - Using checksums and various forms of **sequencing** (to compensate for situations where PDUs have been received out of order or lost)
- **Positive acknowledgements** (aka **ACKs** or **RR (Receive Ready)**)
  - The receiver returns a positive acknowledgement to successfully received, error-free PDUs
- **Negative acknowledgements** (aka **NACKs** or **REJ (Reject)**)
  - The receiver returns a negative acknowledgement to PDUs in which an error is detected
- **Retransmission timeouts** (aka **RTO**)
  - The sender retransmits a PDU that has not been acknowledged after a predefined amount of time
  - The receiver sends a NAK if it does not receive the expected PDU within a predefined time interval

# ARQ (cont'd)

- **3 types of ARQ** :
  - Stop-and-Wait
  - Go-back-N
  - Selective-Repeat (aka Selective-Retry)
- **Stop-and-Wait ARQ** – the simplest ARQ scheme and ensures that each transmitted PDU is correctly received before sending the next
  - After transmitting a PDU, the sender waits for a reply before sending another PDU
  - Thus, PDU transmissions alternates with acknowledgements (positive or negative)
- Stop-and-Wait ARQ can be inefficient in its **utilization of the transmission link** due to the time spent in acknowledging every PDU
  - Efficient resource utilization is an important consideration in the design of a communication protocol

# ARQ (cont'd)

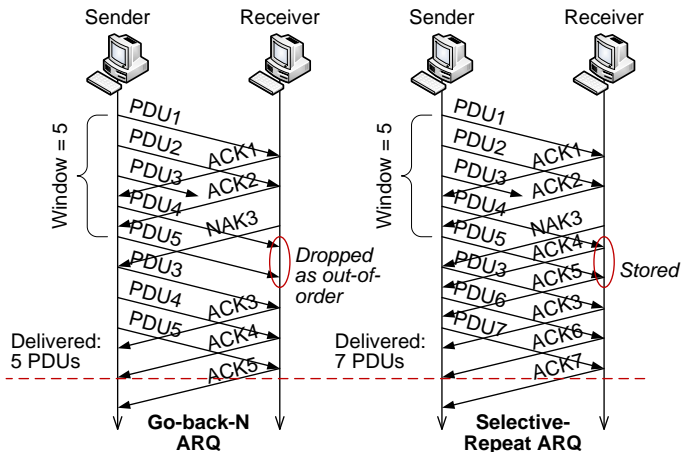
- Stop-and-Wait ARQ implementations



- The efficiency can be increased by transmitting a number of PDUs continuously without waiting for an immediate acknowledgement in **continuous ARQ**
  - In continuous ARQ, both PDU and ACK transmissions occur simultaneously
  - This form of error control is based on **sliding-window flow control**, where the sender may transmit a series of PDUs sequentially numbered
  - At any time, the number of unacknowledged outstanding PDUs should be no more than the window size
- When a NAK is received at the sender (or the retransmission timer expires), it may already have transmitted several PDUs following the one in error
- **2 variants of continuous ARQ** :
  - In **Go-back-N ARQ**, the erroneous PDU and all subsequent PDUs already transmitted should be retransmitted
  - In **Selective-Repeat ARQ**, only the erroneous PDU should be retransmitted

# ARQ (cont'd)

- Go-back-N ARQ simplifies the implementation by not requiring the receiver to reorder PDUs, while Selective-Repeat ARQ requires sufficient storage space to save all PDUs received out of order





- **ARQ persistence** is the willingness to retransmit lost PDUs to ensure reliable data delivery across the link
  - I.e., the maximum number of transmission attempts per PDU
- **3 levels of ARQ persistence** :
  - **Perfect persistence** (reliable) – repeat a lost or corrupted PDU an indefinite (and potentially infinite) number of times until the PDU is successfully received
  - **High persistence** (highly-reliable) – limit the number of times that ARQ may retransmit a particular PDU before the sender gives up on retransmission of the missing PDU and moves on to forwarding subsequent buffered in-sequence PDUs
  - **Low persistence** (partially-reliable) – just a few transmission attempts per PDU