

Teorie Protocoale de Comunicatie (PCom)

Linkuri utile:

Laboratoare: <https://pcom.pages.upb.ro/labs/labs.html>

Enunt Tema 1: <https://pcom.pages.upb.ro/tema1/>

Enunt Tema 2:

https://curs.upb.ro/2023/pluginfile.php/270580/mod_folder/content/0/Enunt_Tema_2_Protocoale_2023_2024.pdf?forcedownload=1

Enunt Tema 3: <https://pcom.pages.upb.ro/tema3>

Enunt Tema 4: <https://pcom.pages.upb.ro/enunt-tema4/>

Enunt Proiect: https://curs.upb.ro/2023/pluginfile.php/323635/mod_folder/content/0/enunt.pdf?forcedownload=1

Abrevieri in Informatica:

- **ACL** = Access Control List
- **ACPI** = Advanced Configuration and Power Interface
- **AD** = Active Directory
- **ADC** = Analog to Digital Converter
- **AES** = Advanced Encryption Standard
- **API** = Application Programming Interface
- **APT** = Advanced Package Tool
- **ASCII** = American Standard Code for Information Interchange
- **ATM** = Automated Teller Machine
- **BCD** = Boot Configuration Data
- **BIOS** = Basic Input/Output System
- **BLE** = Bluetooth Low Energy
- **CA** = Certificate Authority
- **CAN** = Controller Area Network
- **CD-ROM** = Compact Disc - Read-Only Memory
- **CI** = Continuous Integration

- **CISC** = Complex Instruction Set Computing
- **CLI** = Command Line Interface
- **CMOS** = Complementary Metal-Oxide-Semiconductor
- **CPU** = Central Processing Unit
- **CSM** = Compatibility Support Module
- **CTF** = Capture The Flag
- **DAC** = Digital to Analog Converter
- **DHCP** = Dynamic Host Configuration Protocol
- **DMA** = Direct Memory Access
- **DNF** = Dignified YUM
- **DNS** = Domain Name System
- **DTB** = Device Tree Blob
- **DVD-ROM** = Digital Video Disc - Read-Only Memory
- **EDVAC** = Electronic Discrete Variable Automatic Computer
- **ELF** = Executable and Linking Format
- **ESP** = EFI System Partition
- **FAT32** = File Allocation Table 32
- **FSB** = Front-Side Bus
- **FTP** = File Transfer Protocol
- **GCC** = GNU Compiler Collection
- **GDB** = GNU Debbuger
- **GDPR** = General Data Protection Regulation
- **GNU** = GNU's not Unix
- **GNU GPL** – **GNU** General Public License
- **GNU LGPL** – **GNU** Lesser General Public License xiii
- **GPG** = GNU Privacy Guard
- **GPIO** = General Purpose Input Output
- **GPT** = GUID Partition Table

- **GPU** = Graphics Processing Unit
- **GRUB** = GRand Unified Bootloader
- **GUI** = Graphical User Interface
- **GUID** = Globally Unique Identifier
- **HDD** = Hard Disk Drive
- **HIG** = Human Interface Guidelines
- **HTTP** = Hypertext Transfer Protocol
- **HTTPS** = Hypertext Transfer Protocol secure
- **I/O** = Input/Output
- **IDE** = Integrated Development Environment
- **IFS** = Input Field Separator
- **IIC** = Inter-Integrated Circuit
- **IoT** = Internet of Things
- **IP** = Internet Protocol
- **IPC** = Inter-Process Communication
- **ISA** = Instruction Set Architecture
- **IT** = Information Technology
- **IT&C** = Information Technology and Communications
- **JAR** = Java Archive
- **JDK** = Java Development Kit
- **JIT** = just-in-time
- **JRE** = Java Runtime Environment
- **KVM** = Kernel Virtual Machine
- **LAN** = Local Area Network
- **LDAP** = Lightweight Directory Access Protocol
- **LED** = Light-Emitting Diode
- **LVM** = Logical Volume Manager
- **LXC** = Linux Containers

- **MAC** = Media Access Control
- **MBR** = Master Boot Record
- **MISO** = Master In Slave Out
- **MIT** = Massachusetts Institute of Technology
- **MOSI** = Master Out Slave In
- **MSI** = Microsoft Install
- **MU** = Memory Unit
- **NAS** = Network Attached Storage
- **NAT** = Network Address Translation
- **NIC** = Network Interface Card
- **NTFS** = New Technology File System
- **OS** = Operating System
- **OVA** = Open Virtualization Appliance
- **PC** = Personal Computer
- **PCI** = Peripheral Component Interconnect
- **PDF** = Portable Document Format
- **PGP** = Pretty Good Privacy
- **PHP** = PHP Hypertext Preprocessor
- **PID** = Process Id
- **PKI** = Public Key Infrastructure
- **POSIX** = Portable Operating System Interface
- **POST** = Power-On Self Test
- **PWC** = Pulse Width Modulation
- **PXE** = Preboot eXecution Environment
- **QEMU** = Quick Emulator
- **RAID** = Redundant Array of Independent / Inexpensive Disks
- **RAM** = Random Access Memory
- **RDP** = Remote Desktop Protocol

- **RFB** = Remote Frame Buffer
- **RISC** = Reduce Instruction Set Computing
- **ROM** = Read-Only Memory
- **RPM** = RPM Package Manager
- **RSA** = Rivest-Shamir-Adleman
- **SAM** = Security Account Manager
- **SAS** = Serial attached SCSI
- **SATA** = Serial Advanced Technology Attachment
- **SFP** = Small Form-factor Pluggable Transceiver
- **SPI** = Serial Peripheral Interface
- **SSD** = Solid State Drive
- **SSH** = Secure Shell
- **SSL** = Secure Sockets Layer
- **TCB** = Trusted Computing Base
- **TCP** = Transmission Control Protocol
- **TLS** = Transport Layer Security
- **TPM** = Trusted Platform sModule
- **UAC** = User Account Control
- **UEFI** = Unified Extensible Firmware Interface
- **UID** = User Id
- **URI** = Uniform Resource Identifier
- **URL** = Uniform Resource Locator
- **USB** = Universal Serial Bus
- **UUID** = Universally Unique Identifier
- **UX** = User Experience
- **VMM** = Virtual Machine Monitor
- **VNC** = Virtual Network Computing
- **WebUI** = Web User Interface

- **WIMP** = Window, Icon, Menu, Pointer
- **WLAN** = Wireless Area Network
- **YAML** = YAML Ain't Markup Language
- **YUM** = Yellowdog UPdater Modified
- **YUP** = Yellowdog Updater

Abrevieri in PCom:

- **ISO OSI** = International Organization for Standardization Open Systems Interconnections
 - **ISO** = International Organization for Standardization
 - **OSI** = Open Systems Interconnection
- **DLL** = Data Link Layer
 - **HDLC** = High Level Data Link Control
 - **BISYNC** = Binary Synchronous Communication
 - **SYN** = Synchronous Idle (Inceputul unui cadru)
 - **SOH** = Start of Heder
 - **STX** = Start of Text
 - **ETX** = End fo Text
 - **CRC** = Cyclic Redundancy Check
 - **DLE** = Data Link Escape (pt header stuffing)
 - **DDCMP** = Digital Data Communications Message Protocol
 - **SOF** = Start of Frame Delimiter
 - **FCS** = Frame Check Sequence
 - **MAC** = Media Access Control
 - **CSMA/CD** = Carrier Sense Multiple Access/Collision Detection
- **PPP** = Point to Point Protocol
- **PPPoE** = Point to Point Protocol over Etherne
- **MPLS** = Multi Protocol Label Switching
- **VPN** = Virtual Private Network
- **IP** = Internet Protocol
 - **CIDR** = Classes Inter Domain Routing
 - **MTU** = Maximum Transmission Unit
 - **DF** = Don't Fragment
 - **MF** = More Fragments
 - **ARP** = Address Resolution Protocol
- **NAT** = Netwrok Address Tranlation
- **ICMP** = Internet Control Message Protocol

- **PING** = Packet Internet or Inter-Network Groper
- **RIP** = Routing Information Protocol
- **TTL** = Time to live (of a packet)
- **LSP** = Link State Packet
- **BGP** = Border Gateway Protocol
- **AS** = Autonomous Systems
- **OSPF** = Open Shortest Path First
- **TCP** = Transmission Control Protocol
 - **SYN** = Synchronize
 - **RST** = Reset
 - **FIN** = Finish
 - **ACK** = Acknowledge
 - **NACK** = Not Acknowledge
 - **RTT** = Round Trip Time
 - **RTO** = Retransmission Timeout
 - **WIN** = Window Size (dimensiunea ferestrei de receptie)
 - **RWND** = Receive Window
 - **CWND** = Congestion Window
 - **BW** or **BNWD** = bandwidth
 - **AI** = Additive Increase (Crestere Liniară)
 - **SS** = Slow Start
 - **MD** = Multiplicative Decrease
- **UDP** = User Datagram Protocol
- **FTP** = File Transfer Protocol
- **DNS** = Domain Name System
 - **RR** = Resource Records

Teorie Laboratoare PCom

Link laboratoare: <https://pcom.pages.upb.ro/labs/labs.html>

Lab 01. Networking warmup

Link lab: <https://pcom.pages.upb.ro/labs/lab1/lecture.html>

Nivelul fizic

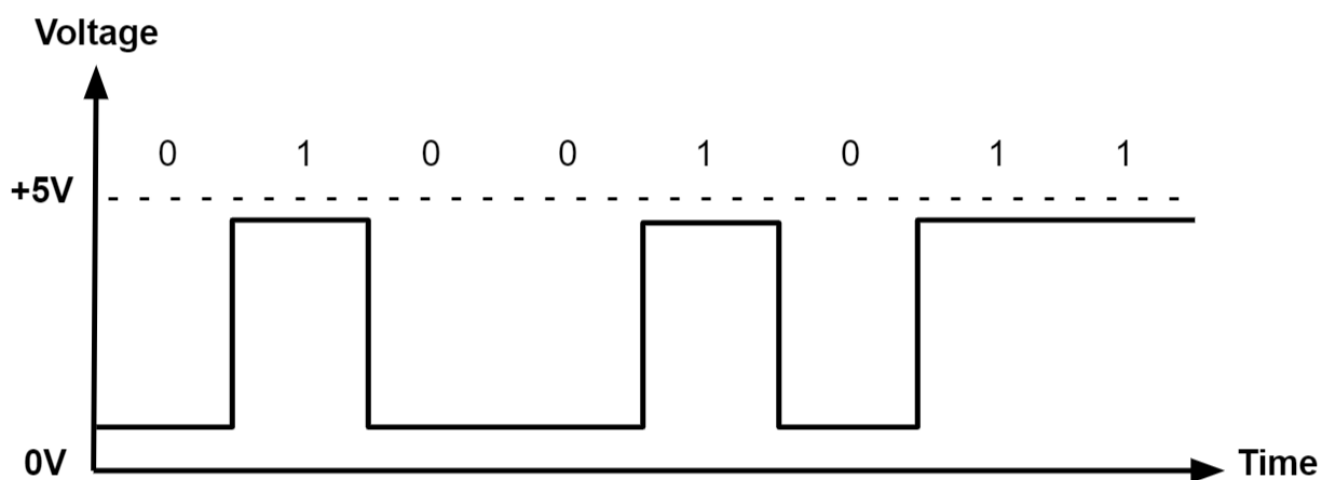
Nivelul fizic se referă la protocoalele și tehnicile utilizate pt a permite schimbul de informații. Schimbul de informații se face peste un **mediu de transmisie** (link).

Exemple de medii de transmisie:

- wireless
- cablu electric
- fibra optica
- semnale de fum

In cazul comunicatiei prin cablu, nivelul fizic se ocupa cu codificarea bitilor prin semnale electrice. Un exemplu de codificare este urmatoarea:

- sender: **la fiecare milisecunda**, cablul electric va fi conectat la 5V pt a transmite bitul 1 si la 0V pt a transmite bitul 0
- receiver: la fiecare milisecunda va masura tensiunea de pe fir



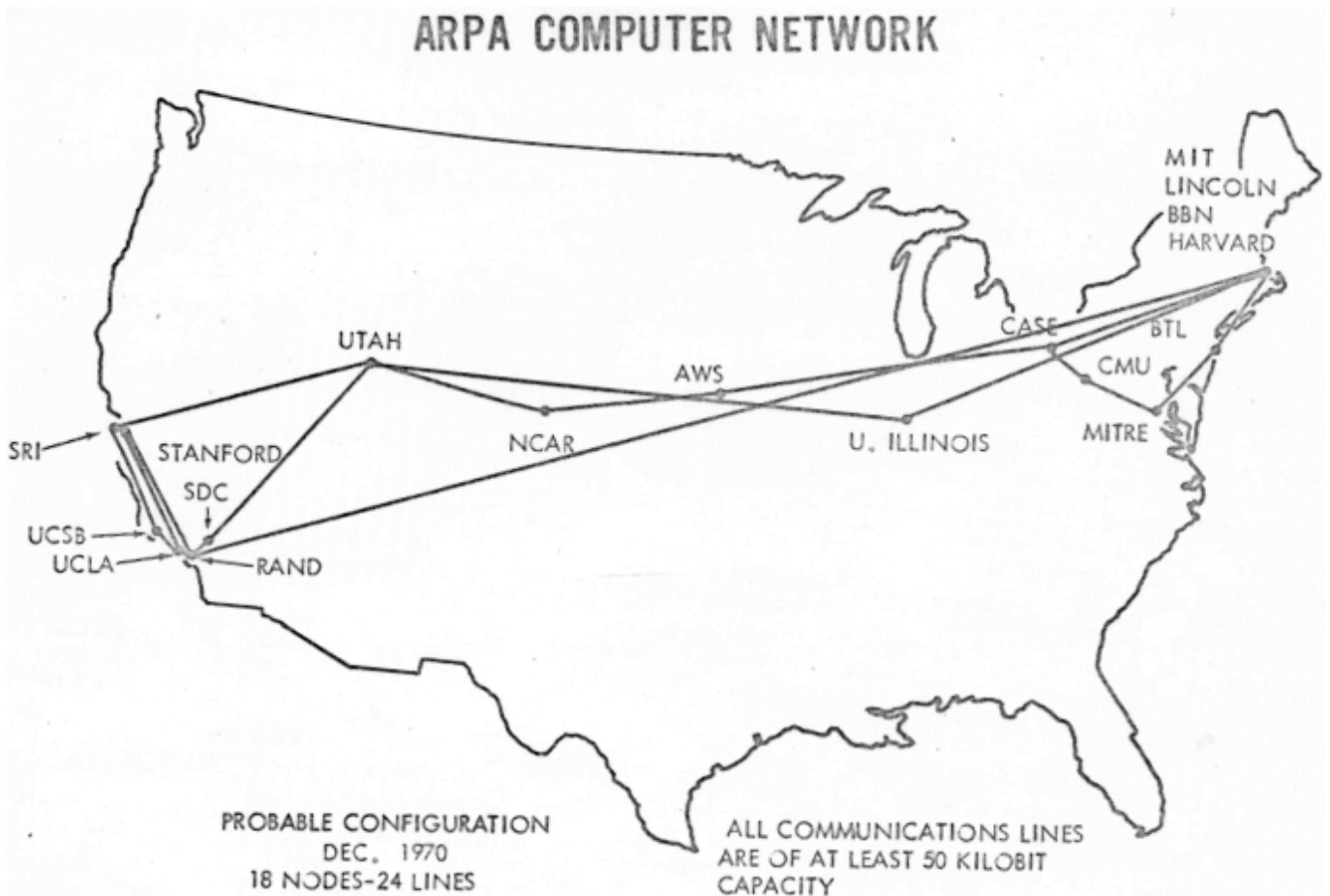
Rata de transmisie (bit rate) = nr de biti trimisi pe secunda

bit rate = nr bits / sec

In exemplul cu cadrul electric, rata de transmisie este de 1000 de biti pe secunda

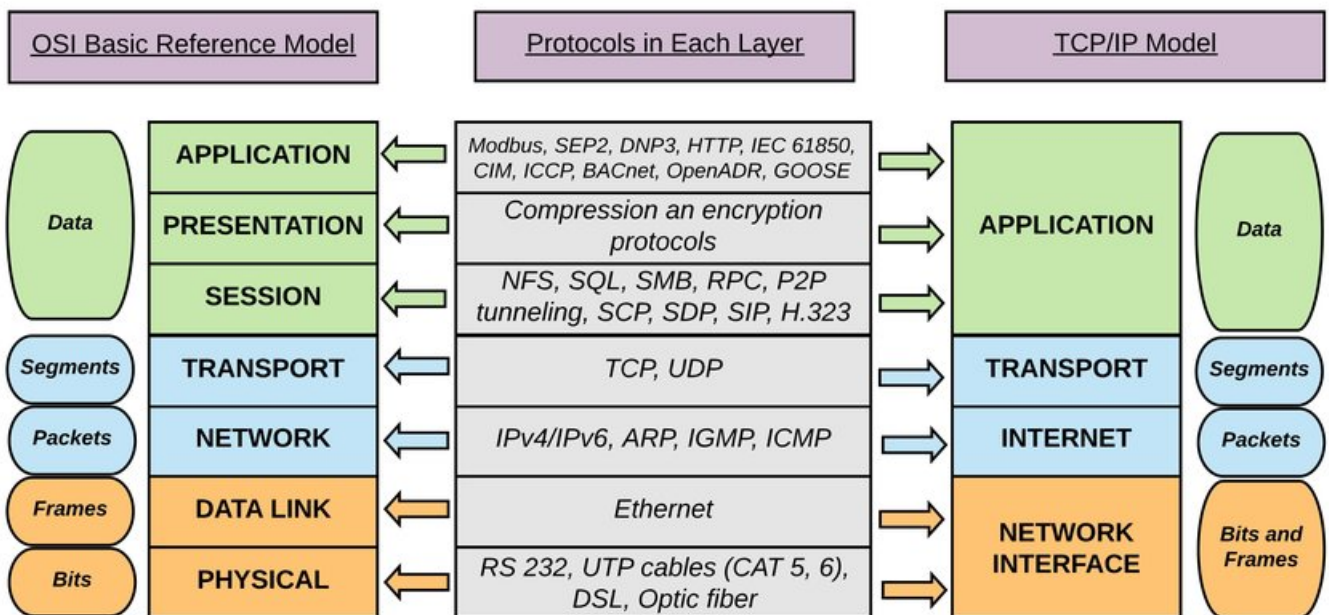
Internetul

La inceputul anilor 1970 internetul se rezuma la comunicarea peste un cablu intre doua dispozitive printr-un protocol simplu, dar in cativa ani complexitatea a crescut enorm. In figura de mai jos vedem precursorul internetului de astazi, **ARPANET**.



Pentru a modela cat mai usor arhitectura Internetului, cercetatorii de la acea vreme au propus diferite modele de referinta. De aceea, **Open Systems Interconnection (OSI)**, modelul propus de Huber Zimmerman, a fost cel mai influent.

Totusi, in practica, **modelul dominant de referinta** folosit este **TCP/IP**.



Lab 02. Datalink. Framing

Link lab: <https://pcom.pages.upb.ro/labs/lab2/lecture.html>

Framing

În general, nu suntem interesați în a lucra cu date la nivel de biți. Aplicațiile pe care le dezvoltăm lucrează cu mesaje, structuri sau fișiere complete. Nivelul fizic ne permite să transmitem un flux de biți de la un dispozitiv la altul, dar datele pe care le transmitem sunt structurate în **blocuri la nivel logic**.

Receptorul trebuie să știe să delimiteze între aceste blocuri pentru a extrage datele corecte. Cum **nivelul fizic nu este ideal**, pot apărea probleme precum desincronizări, astfel încât soluția naivă în care fiecare 8 biți reprezintă un frame nu este valabilă.

```
010 | 01000001 | 01000010 | 10101
      'A'         'B'
```

Unitatea de informație pe care o vom folosi la nivelul **Data Link** este **cadru (frame)** și reprezintă fluxul de biți care constituie un bloc logic de date.

NOTA: Problema pe care încercăm să o rezolvăm este:

Cum face sender-ul codificarea cadrelor (frames) a.i. receiver-ul să le poată extrage eficient din fluxul de biți pe care îl primește de la nivelul fizic

Bit stuffing

O posibilă metodă de framing o reprezintă **bit stuffing**.

Vom folosi **01111110** ca și delimitator de cadre.

De exemplu, dacă vrem să trimitem **0100**, atunci o să îl codăm ca și **01111110 | 0100 | 01111110**. Receiver-ul, doar după ce a primit **01111110** va începe să citească conținutul cadrului.

Ce facem în cazul în care vrem să trimitem 6 biți de 1, **111111**? Regula este simplă, după fiecare secvență de 5 biți de 1, **11111**, se inserează un 0. Astfel, delimitatorul **01111110** nu o să apară niciodată în conținutul unui cadru.

```
Sender
111111 -> 1111101

Receiver
1111101 -> 111111
1111100 -> 111110
```

Putem dezvolta astfel un protocol foarte simplu de nivel 2. Specificația acestui protocol conține structura și regula definită pentru a nu întâlni delimitatorul în datele pe care le vom transmite (payload).

```
DELIM | PAYLOAD | DELIM
```

Character stuffing in practica

Cum in software ne este mult mai usor sa lucram la nivel de byte (octet, 8 biti), decat bit, nivelul fizic ne ofera si un serviciu de trimitere de fluxuri de bytes.

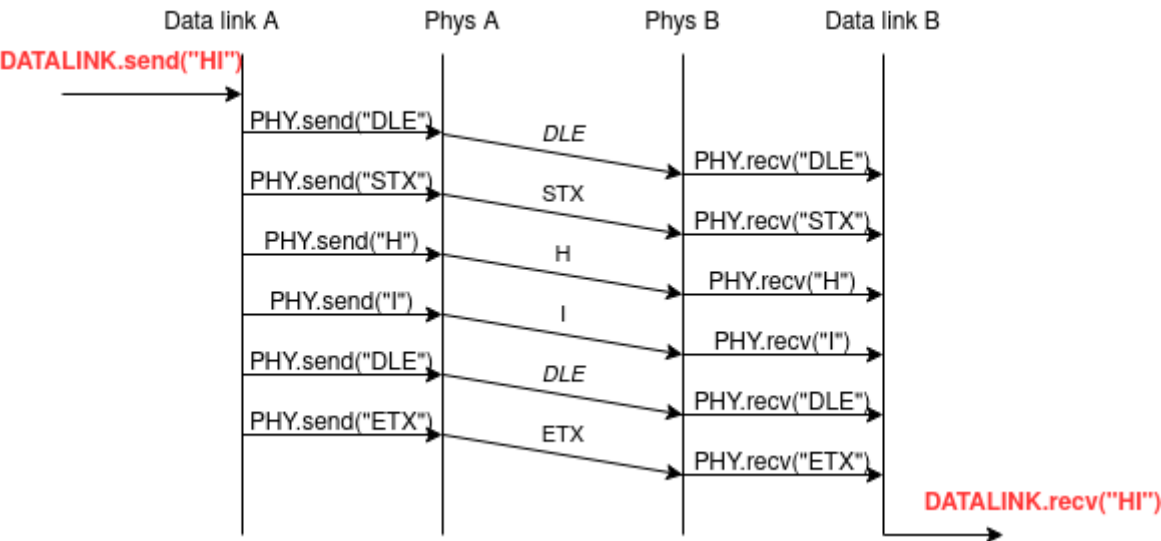
In mod similar cu bit stuffing, vom folosi mai multe caractere speciale pt a ne delimita frame-ul.

Vom folosi DLE, STX si ETX definiti in [table ASCII](#).

```
A B C => DLE STX A B C DLE ETX
```

```
A B C DLE STX D => DLE STX A B C DLE DLE STX D DLE ETX
```

Mai jos avem o diagrama care cuprinde transmisia de date folosind framing. Veem cum la nivelul **Data Link** folosindn protocolul nostru simplu cu bytes de separare putem oferi un serviciu de trimitere de frames.

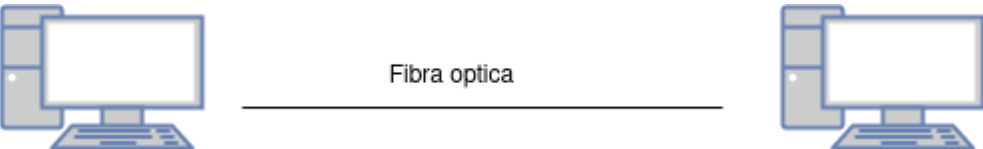


Tipuri de comunicatie

- Point-to-Point
- Point-to-Multipoint

Point-to-Point

Comunicarea Point-to-Point se intampla atunci cand avem doar doua dispozitive. In acest caz, dispozitivele nu trebuie sa specifice cui vor sa trimita frame-uri.



Exemple de protocoale de nivel 2 dezvoltate pt comunicarea Point-to-Point:

- **PPP** = Point-to-Point Protocol
- **HDLC** = High-Level Data Link Control

Point-to-Multipoint

Intr-o transmisie de tip Point-to-Multipoint, avem un transmitator si mai multi receptori. Cel mai popular mod de a identifica distanta este de a include un **camp de identificare in antetul protocolului** (de exemplu **adresa MAC** in protocolul **Ethernet**).

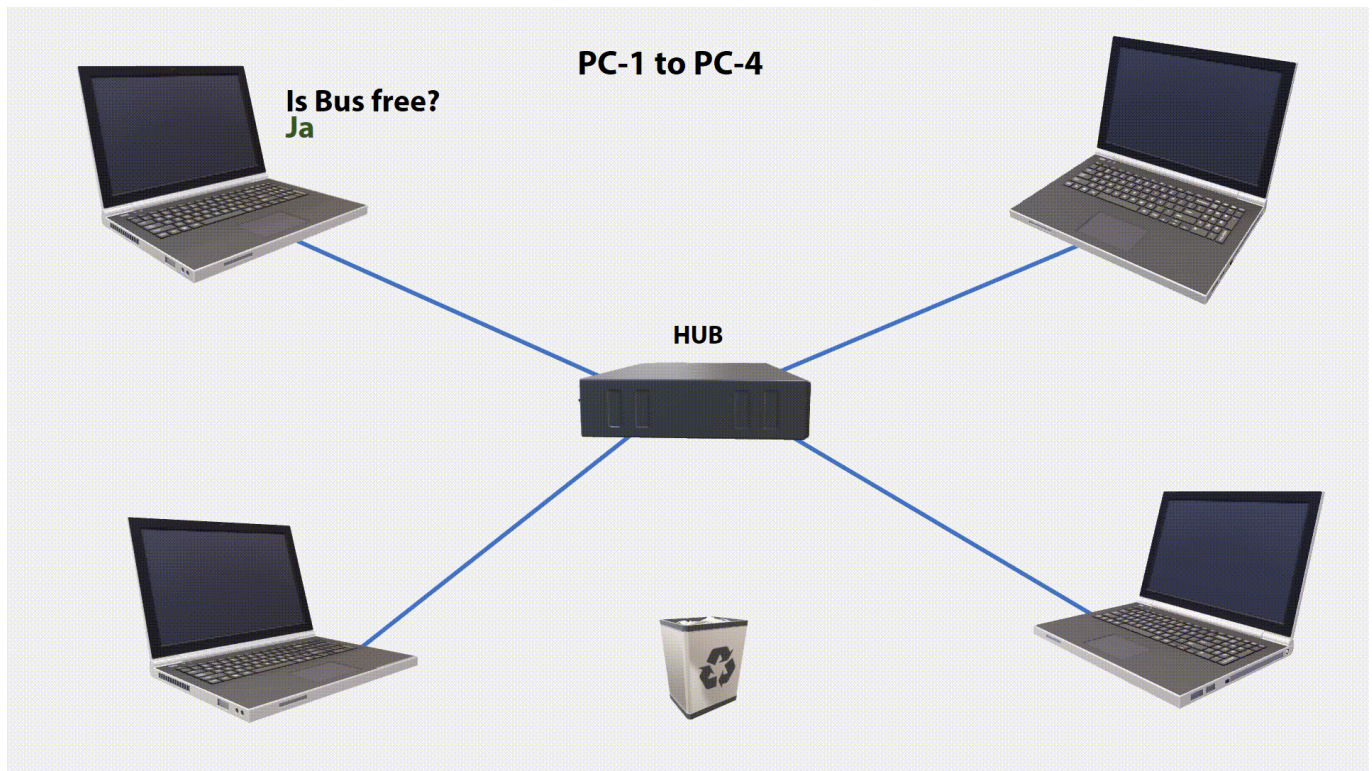
In imaginia de mai jos sunt exemple de comunicatii multipoint.

Comunicatie multipoint - fiecare dispozitiv poate masura voltajul pe fir



Comunicatie multipoint - exista un dispozitiv ce trimite mai departe cadrele





Metrici

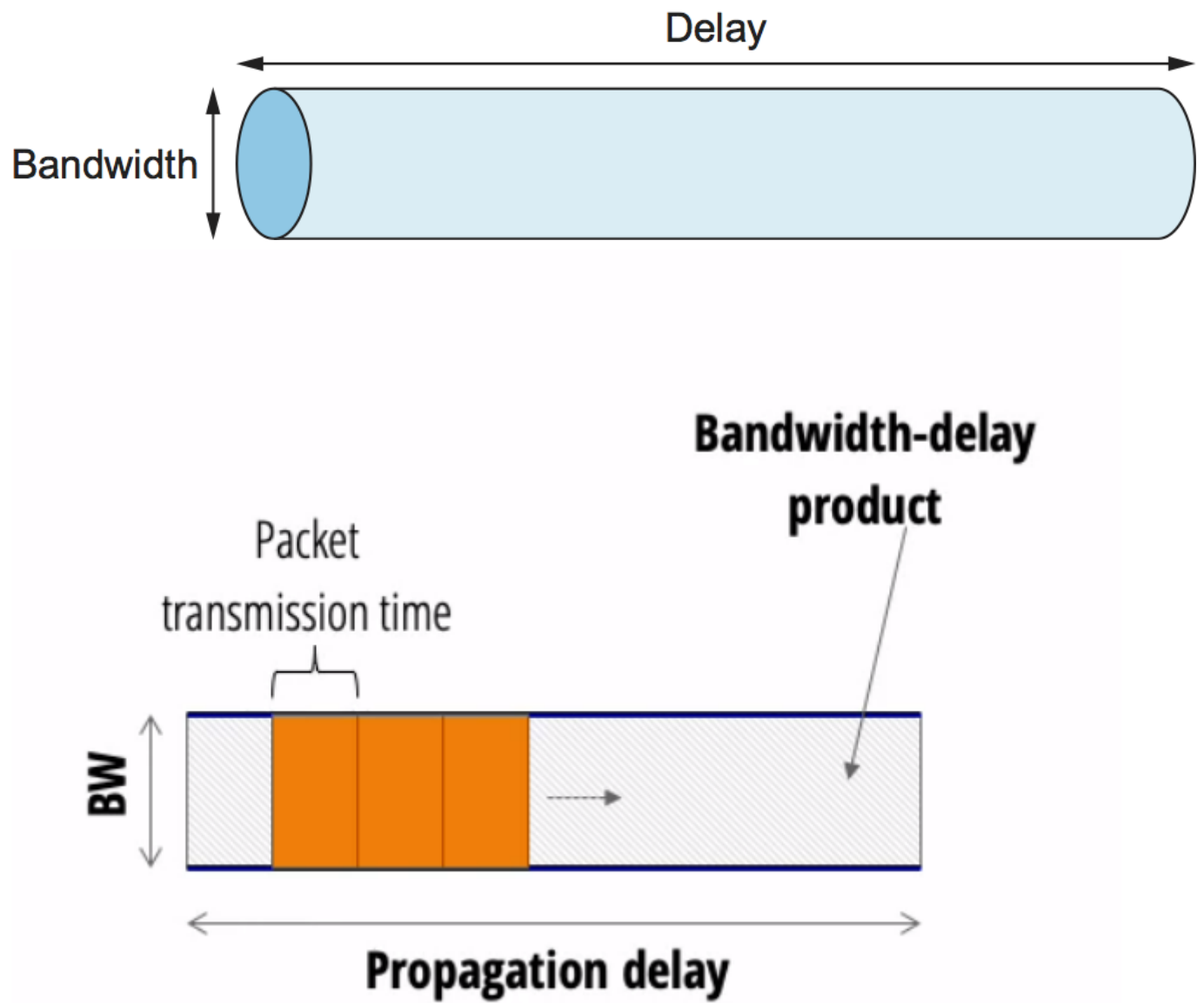
Pentru a putea studia **performanta** unui protocol de nivel **Data Link**, ne intereseaza urmatoarele metrici:

- **Bandwidth (BW)**
 - se masoara in **biti / secunda**
 - = **viteza de propagare**
 - reprezinta cantitatea de informatie care poate fi transmisa intr-o unitate de timp pe legatura de date
- **Delay**
 - se masoara in **secunde**
 - = **timpul de propagare**
 - reprezinta timpul care le ia unor date trimise printr-un mediu sa ajunga la destinatie
- **Bandwidth delay product (BDP)**
 - reprezinta numarul total de biti ce se pot afla pe un link la un anumit moment de timp
 - $BDP = \text{Bandwidth} * \text{Delay}$

Legatura de date poate fi asemanata cu un cilindru in care datele sunt introduse de catre transmitator si primite de catre receptor.

Aria sectiunii cilindrului reprezinta **viteza de transmisie (Bandwidth)**, iar **inaltimea** este **timpul de propagare (Delay)**.

Deci, cantitatea de informatie aflata pe link la un anumit moment de timp: $BDP = \text{Bandwidth} * \text{Delay}$



Tabelul de mai jos prezinta mai multe metrice pt link-uri existente:

Tip Link	Bandwith (BW)	One-Way Distance	Delay	Bandwidth * Delay (BDP)
Wireless LAN	54 Mbps	50 m	0.15 us	18 bits
Satellite	1 Gbps	35000 km	115 ms	230 Mb
Cross-country fiber	10 Gbps	4000 km	40 ms	400 Mb

Ce se foloseste in Internet? Ethernet

Pentru acest nivel din stiva de interent intalnim foarte des protocolul Ethernet. Atnet-ul (header-ul) este urmatorul:

802.3 Ethernet packet and frame structure

Layer	Preamble	Start frame delimiter (SFD)	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap (IPG)
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	42-1500 octets	4 octets	12 octets
Layer 2 Ethernet frame	(not part of the frame)		← 64-1522 octets →						(not part of the frame)
Layer 1 Ethernet packet & IPG	← 72-1530 octets →								← 12 octets →

IGP = Interior Gateway Protocols (o perioada de inactivitate)

Lab 03. Transfer de date peste un link imperfect

Link lab: <https://pcom.pages.upb.ro/labs/lab3/lecture.html>

De parcurs inainte de laborator:

- Reliable data transfer on top of an imperfect link

Materiale video optionale:

- How do CRCs work?

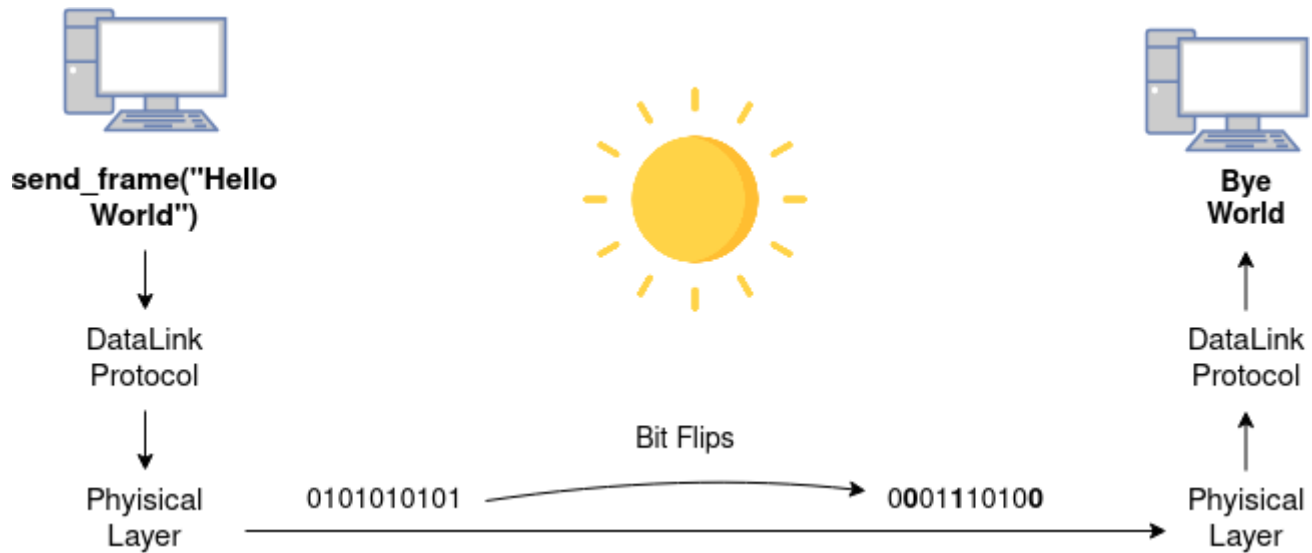
Detectarea erorilor de transmisie

In timpul transmiterii de date, pot aparea **erori**. Acestea se pot manifesta ca biti ale caror valor sunt schimbate intr-un cadru.

Intalnim doua tipuri de erori la nivelul legaturii de date:

- cadrele pot fi corupte
- cadrele pot fi pierdute sau pot aparea cadre neasteptate

De exemplu, daca trimitem sirul de biti 11111111 prin intermediul unui cabul, din cauza **interferentelor electromagnetice**, ultimul bit ar putea avea valoarea schimbata si receptorul ar primi 11111110.



În general, pentru a putea transmite date peste link imperfect, vom folosi una dintre următoarele abordări:

- detectarea erorilor și retransmisia (**CRC**, **Checksums**)
- corectarea erorilor (**distanța Hamming**)

În acest caz, apare o decizie de proiectare în dezvoltare protocolului. De exemplu, dacă știm că transmiterea se întâmplă peste un mediu cu **latență mare** și rată mare de corupere a datelor, cum ar fi comunicarea între Pământ și Marte (~ 29 de minute), ar avea sens să **folosim o metodă de corectare a erorilor**.

În schimb, dacă **latența este mică**, ar fi mult mai optim să realizăm o **retransmisie**.

Ethernet folosește un **camp CRC** pt **detectarea erorilor**.

Sume de control (**checksum**)

Adesea, atunci când transmitem date peste un link, este necesar ca receptorul să determine dacă cadrul primit a fost corupt.

Pentru a face acest lucru, **transmitatorul va include un nou camp** numit **checksum** în protocol, care este **rezultatul aplicării unei funcții pe continuul cadrului**.

Un exemplu simplu de funcție de **checksum** este suma tuturor octetilor din cadrul, **mod 256**.

```
uint8_t compute_checksum(const char *buff, size_t count)
{
    /* Ca input primim un buffer char *buf de dimensiune int count */
    uint32_t sum = 0;
    uint8_t checksum

    /* Adăugăm în sum fiecare byte din buffer */
    while (count > 0) {
        sum += ((uint8_t *) buff)
        buf += 1;
        count -= 1;
    }

    checksum = sum % 256;
```



```
    return checksum;
}
```

Ce facem daca am detectat o eroare? De cele mai multe ori, la detectia unei erori se va face o retransmisie de catre protocolul de nivel superior (**TCP** la nivel transport)

O **problema** a algoritmilor de **checksum** este simplitatea acestora ce poate cauza **coliziuni**.

PROBLEMA: Functia poate intoarce acelasi rezultat pentru input-uri diferite

```
Cadru           : 6 23 4
Cadru cu checksum : 6 23 4 33 (6 + 23 + 4 = 33 % mod 256 = 33)
Cadru la receptor : 8 20 5 33 (8 + 20 + 5 = 33 % mod 256 = 33)
```

In acest exemplu, chiar daca continutul **mesajului s-a schimbat, checksum-ul calculat a fost acelasi**, existand o sansa de 1/256 (256 - de la operatorul de modula) ca eroarea sa nu fie detectata.

Pentru a rezolva aceasta problema, s-a ales folosirea unor algoritmi precum **Cyclic Redundancy Codes CRC**.

NOTA Termenul de **checksum** a fost folosit initial pentru a descrie algoritmi de tipul sume, dar in ziua de azi cuprinde si algoritmi mai sofisticati, precum **CRC**.

Cyclic Redundancy Codes (CRCs)

Daca reprezentam datele transmise ca pe un numar, atunci **restul impartirii** este **valoarea** pe care o putem introduce in **header**, iar receptorul poate verifica daca datele primite au acelasi rest.

In practica, nu folosim numere, ci **polinoame**, printre altele fiind mult mai usor de lucrat cu ele (nu o sa avem carry).

Cyclic Redundancy Codes (CRC) reprezinta **restul impartirii polinomiale modulo 2 a datelor pe care vrem sa le trimitem**.

Putem vedea payload-ul ca si reprezentarea unui polinom.

```
PAYLOAD=      'H'      'i'      '!'
              01001000 01101001 00100001
```

Cu reprezentarea matematica:

$$x^{22} + x^{19} + x^{14} + x^{13} + x^{11} + x^8 + x^5 + x^0.$$

De ce **modulo 2** (inelul claselor de resturi modulo 2)? Deoarece vrem ca indicii in urma calculelor sa fie 1 sau 0, atunci cand facem impartirea, vom ajunge la valori reale, iar noi, putem folosi doar valori binare.

Pentru **optimizari**, operatiile in acest inel sunt echivalent cu **XOR**. In functie de polinomul la care o sa ne raportam, avem diferite implementari de **CRC**.

CRC 32 foloseste umratorul polinom:

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

Cum avem doar 32 de biti, nu ne intereseaza indicele lui x^{32} . Polinomul a fost ales astfel incat sa functioneze bine în cazul erorilor în rafala.

Pentru string-ul **123456789**, valoarea CRC32 este **0xCBF43926**.

Un exemplu de impartire de polinoame modulo 2 este acesta:

```

10011 ) 11010110110000 = Bits of payload
=Poly  10011,.,.,.,.
-----,.,.,.,.
10011,.,.,.,. (operatia de xor cand primul bit e 1)
10011,.,.,.,.
-----,.,.,.,.
00001,.,.,.,. (cand primul bit e zero, doar face un shift stanga
00000,.,.,.,. pentru a lua urmatorul indice de exponent)
-----,.,.,.,.
00010,.,.,.,.
00000,.,.,.,.
-----,.,.,.,.
00101,.,.,.,.
00000,.,.,.,.
-----,.,.,.,.
01011,.,.,.,.
00000,.,.,.,.
-----,.,.,.,.
10110,.,.,.,.
10011,.,.,.,.
-----,.,.,.,.
01010,.,.,.,.
00000,.,.,.,.
-----,.,.,.,.
10100,.,.,.,.
10011,.,.,.,.
-----,.,.,.,.
01110,.,.,.,.
00000,.,.,.,.
-----,.,.,.,.
1110 = Remainder = The CRC!

```

O posibila implementare a algoritmului **CRC 32** este urmatoarea:

```

uint32_t compute_crc32(const char *buffer, size_t len)
{
    /* unsigned char *buffer contine payload-ul, len este lungimea acestuia
    */
    /* Prin conventie crc-ul initial are toti bitii setati pe 1 */
    uint32_t crc = ~0; // 0xffffffff
    const uint32_t POLY = 0xEDB88320;

    /* Parcurgem fiecare byte din buffer */
    while(len--)
    {
        /* crc contine restul impartirii la fiecare etapa */
        /* nu ne intereseaza catul */
        /* adunam urmatorii 8 bytes din buffer */
        crc = crc ^ *buffer++;
        for( int bit = 0; bit < 8; bit++ )
        {
            /* 10011 ) 11010110110000 = Bytes of payload
               =Poly  10011,,,,,,,,
                       -----,
                       10011,,,,,,,, (operatia de xor cand primul bit e
1)
                       10011,,,,,,,,
                       -----,
                       00001,,,,,,,, (asta e noua valoare a lui crc)

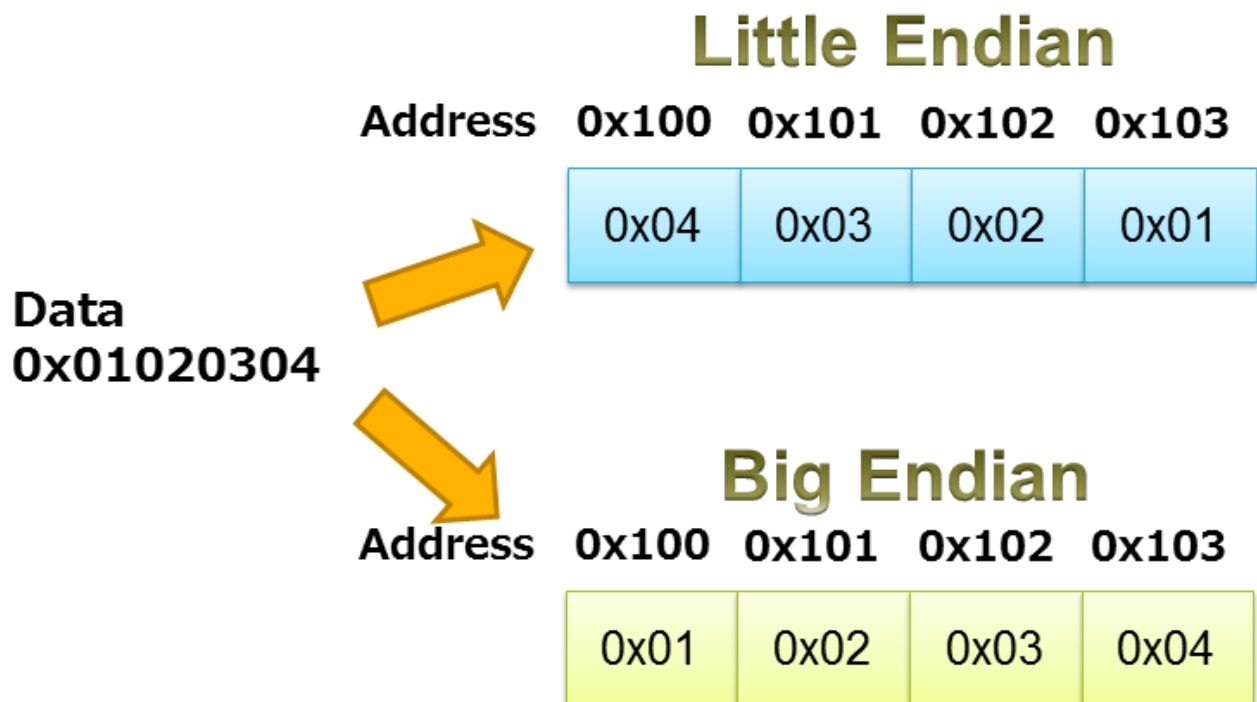
(crc >> 1) ^ POLY
    */
            if( crc & 1 )
                crc = (crc >> 1) ^ POLY;
            else
                /* 10011 ) 11010110110000 = Bytes of payload
                   =Poly  10011,,,,,,,,
                           -----,
                           10011,,,,,,,,
                           10011,,,,,,,,
                           -----,
                           00001,,,,,,,, primul bit e 0,
                           00000,,,,,,,,
                           -----,
                           00010,,,,,,,, am facut shift la dreapta,
pentru ca suntem pe **little endian**
    */
                crc = (crc >> 1);
        }
    }
    /* Prin conventie, o sa facem flip la toti bitii
    crc = ~crc;
    return crc;
}

```

In functie de ordinea in care un sir de octeti este **stocat in memorie**, avem doua interpretari:

- Little Endian
- Big Endian

Reprezentarea cu care suntem cel mai bine obisnuiti este **Big Endian**, asa cum reprezentam datele pe foaia, cel mai **semnificativ byte** este **primul**.



In general, procesoarele moderne folosesc Little Endian.

Totusi, placie de retea folosesc Big Endian.

O sa intalnim denumirea **Network Order** (**Big Endian**) si **Host Order** (**Little Endian**).

In API-ul POSIX avem mai multe functii care se pot folosi pentru a face trecerea Host Order <-> Network Order

```
#include <arpa/inet.h>

// host to network long
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);

// network to host long
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort)
```

Lab 04. Protocolul IP. Forrding

Link lab: <https://pcom.pages.upb.ro/labs/lab4/lecture.html>

Protocolul IP: https://youtu.be/rPoalUa4m8E?list=PLowKtXNTBypH19whXTVoG3oKSuOcw_XeW

Procesul de forwarding: <https://youtu.be/VWJ8GmYnjTs?>

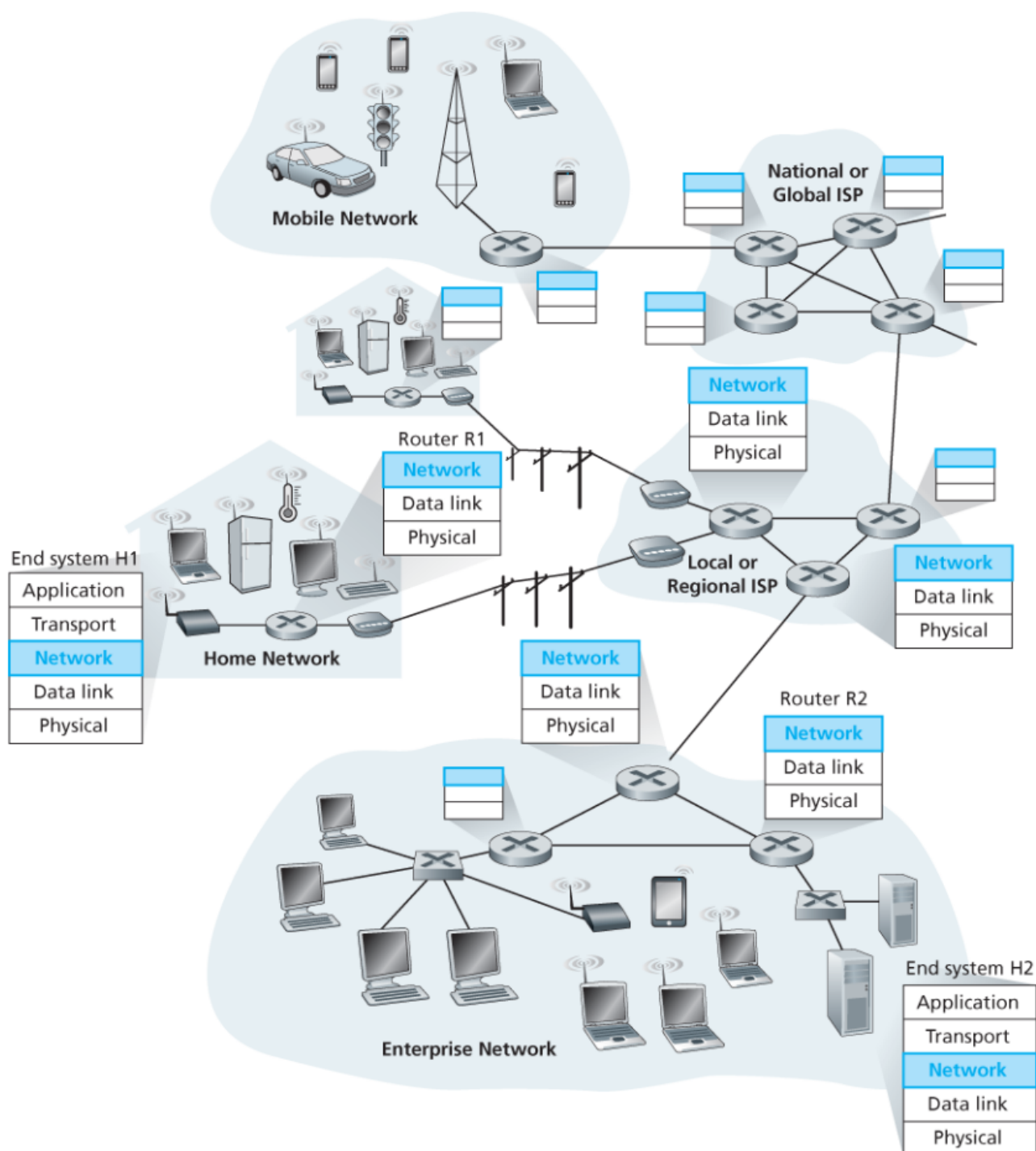
list=PLowKtXNTBypH19whXTVoG3oKSuOcw_XeW

Nivelul Retea

La ultimele laboratoare, am reusit sa dezvoltam o serie de protocoale pentru dispozitive direct conectate. Astazi, punem bazele unui protocol ce permite transferul de date intr dispozitive ce nu sunt direct conectate, fiecare dispozitiv afland0use intr-o retea diferita.

Numele protocolului este **Internet Protocol (IP)**, dezvoltat initial in anii 1970.

Mai jos gasim o reprezentare a unei topologii in care se afla mai multe retele.



Routing

O să considerăm acum următorul scenariu. În marile capitale ale Europei avem mai multe dispozitive. De exemplu, în Londra, avem 4 dispozitive conectate print **Ethernet** (3 calculatoare și un dispozitiv pe care îl vom numi **router**). La fel și în București. Dispozitive numite **router** sunt conectate printr-un protocol de nivel 2 de tip **Point-to-Point**.



Vrem să trimitem un cadru de la Host A, în Londra, la Host B, în București. Dacă Host A, trimite un cadru de nivel 2, în acest caz Ethernet cu adresa destinație MAC B, acesta nu ar fi considerat de niciunul dintre dispozitive pentru că nimeni din Londra nu are această adresă MAC. Dacă în schimb, am modifică aceste dispozitive numite routere să știe unde se află fiecare adresă MAC din toată Europa, cel din Londra ar primi un cadru Ethernet de la Host A cu destinația MAC B și ar trimite conținutul acestuia către Paris folosind protocolul de tip PPP dintre acestea. Totuși, între Londra și Paris este o conexiune de tip PPP, destinația se pierde între aceste conexiuni deoarece protocoalele de tip PPP nu folosesc o destinație.

Avem nevoie de un protocol peste nivelul DataLink care să se ocupe cu identificarea și transmiterea între ceea ce vom numi de acum rețele (e.g. rețeaua din București). În acest scop, a fost dezvoltat protocolul IP Protocol (IP) de nivel network. Astfel, o datagramă IP va fi encapsulată atât în protocolul Ethernet cât și în PPP, și routerele se vor ocupa de transmisie.

Protocoale utilizate:

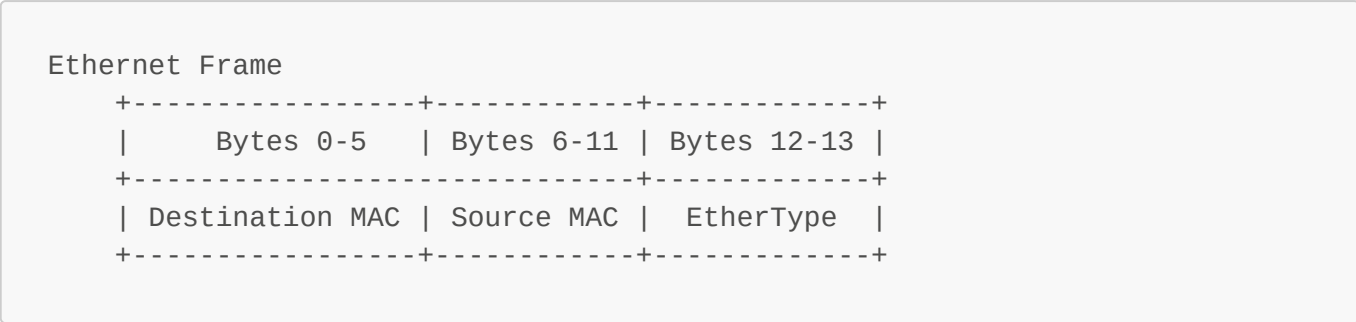
- Ethernet
- IP

Ethernet

Ethernet este echivalentul protocolului de **DataLink** pe care l-am implementat în primele laboratoare.

Noi vom lucra doar cu **cadre Ethernet** ce sunt transmise ca payload peste implementarea protocolului fizic **Ethernet**.

Cum CRC-ul este calculat in hardware, nu o sa il regasim in header. In acest caz, header-ul pe care il vom folosi este urmatorul:



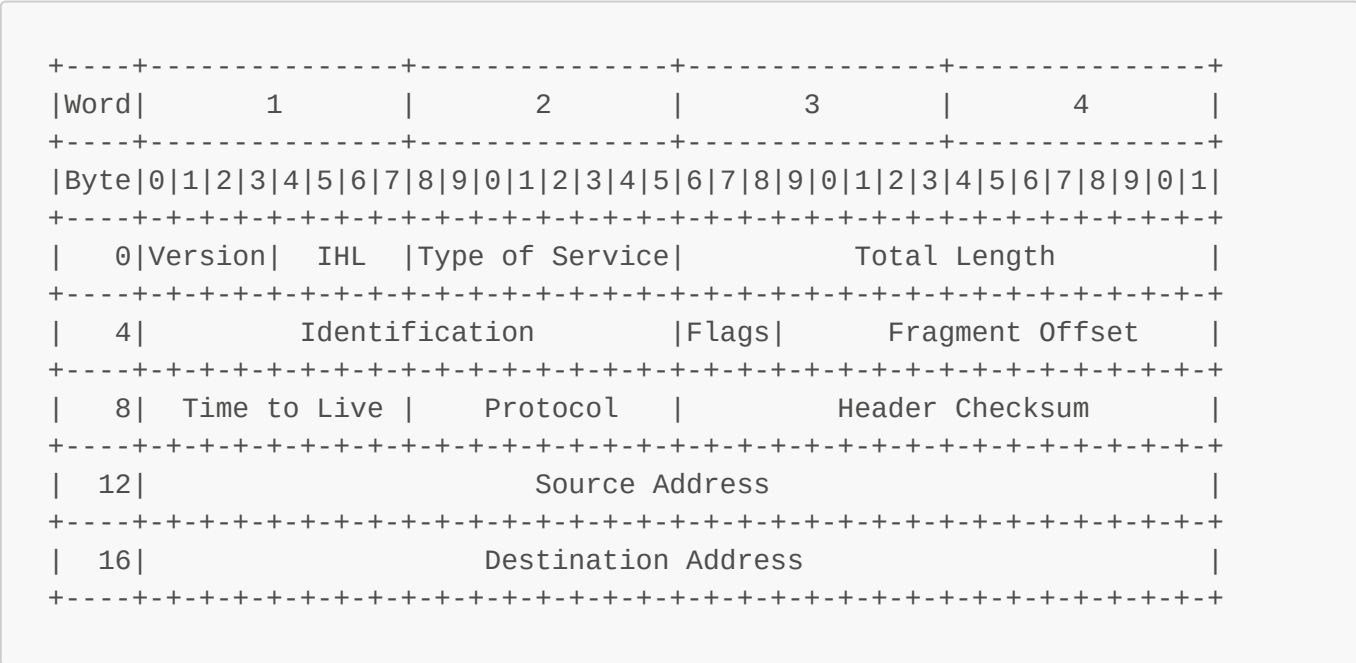
Adresa MAC Destinatie reprezinta identificatorul dispozitivului de nivel 2 catre care a fost trimis acest caddru.

In cadrul laboratorului putem folosi urmatoarea structura peste un cadrul Ethernet. Pentru campul EtherType ne intereseaza doar valoarea ETHERTYPE_IP (0x0800).

```
struct ether_header {
    uint8_t ether_dhost[6];
    uint8_t ether_shost[6];
    uint16_t ether_type;      // ETHERTYPE_IP
};
```

IPv4

Protocolul IP este utilizat pentru a permite dispozitivelor conectate in retele diferite sa schimbe informatii prin intermediul unui dispozitiv intermediar numit router. Hedaer-ul unui pachet (pachet) IP este urmatorul:



TTL = Time to Leave

Campul **Time to Leave** este un numar **decrementat** de fiecare router pentru a evita bucle. **checksum** este campul folosit pentru a **verifica integritatea header-ului IP**.

Destination Address este adresa IP a destinatiei.

Urmatoarea structura poate fi folosita pentru a reprezenta un pachet IPv4

```
struct iphdr {
    // The following syntax means that version has 4 bits and ihl 4 bits.
    uint8_t    ihl:4, version:4; // don't care
    uint8_t    tos;           // don't care
    uint16_t   tot_len;       // don't care
    uint16_t   id;           // don't care
    uint16_t   frag_off;      // don't care
    uint8_t    ttl;          // Time to Live -> to avoid loops, we will
decrement
    uint8_t    protocol;      // don't care
    uint16_t    check;        // checksum      -> Since we modify TTL,
// we need to recompute the checksum
    uint32_t    saddr;        // don't care
    uint32_t    daddr;        // the destination of the packet
};
```

Observam ca o adresa IP precum **10.30.4.2** poate fi reprezentata in memorie ca un integer pe 32 de biti, **uint32_t**.

Campul **checksum** este complementul fata de 1 al sumei tuturor cuvintelor de 16 biti din header. Totusi, cum noi modificam doar campul **TTL** si pentru ca checksum este o suma, exista o metoda mai rapida de a calcula noul checksum folosind formula:

Fie:

- HC - vechiul checksum din header
- C - complementul față de 1 al sumei campurilor din header
- HC' - noul checksum
- m - vechea valoare a câmpului de 16 biți (TTL în cazul nostru)
- m' - noua valoare a câmpului de 16 biți (TTL --)

$$\begin{aligned}
 HC' &= \sim(C + (-m) + m') \\
 &= \sim(\sim HC + \sim m + m')
 \end{aligned}$$

Avem astfel formula finala:

```
new_checksum = ~(~old_check + ~((uint16_t)old_ttl) + (uint16_t)ip_hdr->ttl) - 1;
```

Acel -1 de la final apare pentru a evita translatia din network order in host order pentru valorile de ttl pe 16 biti.

Adrese IP

În general, o adresă IP este de forma **10.20.30.40** și este reprezentată pe 32 de biți.

Cum avem foarte multe adrese IP, în general o să le structurăm în blucuri (**rețele**).

O rețea este identificată printr-un **prefix** și o **mască**. De exemplu, rețeaua din București în exemplul nostru este: **10.20.30.0/24**

Mască de rețea **/24** înseamnă că primii **24 de biți** sunt utilizați pentru identificarea **rețelei**, iar restul de **8 biți**, pentru identificarea dispozitivelor (**hosturilor**) din acea rețea.

Mască pe **/24** de biți =>

- primii **24** de biți vor fi **1**
- restul de **8** biți vor fi **0**

Network în București

Network: 10.20.30.0/24

Prefix: 10.20.30.0

Mask: 255.255.255.0 (24 = nr de biți de 1 de la stânga la dreapta)

Câte adrese IP sunt în rețeaua din București? Avem **256** de adrese IP disponibile, **10.20.30.0** - **10.20.30.255**. Adresele din acest bloc pot fi asignate dispozitivelor din București.

Măștile de subrețea sunt utilizate împreună cu notația **CIDR** pentru a defini dimensiunea unei rețele și numărul de dispozitive conectate la aceasta

CIDR = Classless Inter-Domain Routing

 CIDR addr

Procesul de forward (dirijare)

Un router, pentru a trimite un pachet către următorul dispozitiv (**hop**) va trebui să realizeze mai multe acțiuni (proces de forward). Procesul complet de forwarding este următorul

1. Pe una dintre interfețe este recepționat un pachet IP
2. Verifică checksumul. Dacă este greșit aruncă pachetul
3. Rulează algoritmul de **Longest Prefix Match (LPM)** în **tabela de rute** pentru a găsi următorul hop.
4. În cazul în care nicio intrare din tabelă nu face match, router-ul aruncă pachetul.
5. Routerul **decrementează** câmpul **TTL** din header-ul IP. În cazul în care **TTL** este 0, pachetul este aruncat
6. Recalculează **checksum**-ul
7. Router-ul face update la adresa **MAC** sursă a pachetului în adresa proprie și adresa **MAC** destinație a următorului HOP.
8. Pachetul este trimis către următorul hop identificat prin **LPM**

Tabela de rutare este populată de algoritmi de rutare și este structurată astfel:

Prefix	Next hop	Mask	Interface
192.168.0.0	192.168.0.2	255.255.255.0	0
192.168.1.0	192.168.1.2	255.255.255.0	1
192.168.2.0	192.168.2.2	255.255.255.0	2
192.168.3.0	192.168.3.2	255.255.255.0	3

Un dispozitiv are mai multe interfete pe care poate sa trimita pachete (ex.: din Londra are una pt Paris si una pt Berlin)

Longest Prefix Match (LPM)

Pentru a determina prefixul dintr-o adresa IP si o masca, putem folosi urmatoarea operatie pe biti: `ip & mask`.

Adresa IP	Mask	Prefix
10.20.30.4	& 255.255.255.0	= 10.20.30.0

Algoritmul are o specificatie relativ simpla:

- Router-ul cauta in tabela de rutare intrarile pt care: `(ip & mask) == prefix`
- Dintre toate rutele care fac match in etapa anterioara:
 - este aleasa ruta cea mai specifica (**prefixul** cel mai mare)
 - Daca doua rute au aceeasi specificitate, e va folosi ruta cu cel mai mic metric

Un exemplu este cel din rumatoarea imagine, in care `route 2` este cea mai specifica si urmatorul hop este conectat la interfata `S1`.

Bit by bit

desIP 172.16.2.1

172.16.000000100000001

route 1 172.16.1.0
255.255.255.0

172.16.00000001xxxxxx

route 2 172.16.2.0
255.255.255.0

172.16.00000010xxxxxx

route 3 172.16.0.0
255.255.0.0

172.16.xxxxxxxx,xxxxxx

prefix	out int
172.16.1.0/24	S0
172.16.2.0/24	S1
172.16.0.0/16	S2

OBS: `route 1` nu face match, deoarece `(destIp & mask) != prefix`.

O posibila implementare in `O(n)` a algoritmului este urmatoarea:

```
// Avem o tabela de rutare table {prefix, next_hop, mask, interface}

// tabela trebuie sortata descrescator prefix si masca
qsort((void *)table, table_len, sizeof(struct route_table_entry),
comparator);

for (int i = 0; i < table_len; i++) {

    /* Cum tabela este sortată, primul match este prefixul ce mai specific
    */
    if (table[i].prefix == (target_ip & mask)) {
        return &table[i];
    }
}
```

Lab 5. Protocolul UDP. Fereastra glisanta

Link lab: <https://pcom.pages.upb.ro/labs/lab5/lecture.html>

De parcurs inainte de laborator:

- [The User Datagram Protocol](#)
- [What is socket?](#)

Nivelul Retea

Protocoalele de nivel transport folosesc serviciile oferite de catre nivelul retea. In Internet, **nivelul retea** ofera un serviciu **fara conexiune**. Nivelul retea identifica fiecare host folosind o adresa IP. Nivelul retea poate transmite pachete ce au pana la 65KBytes de date catre orice destinatie cunoscuta din retea locala sau din Internet.

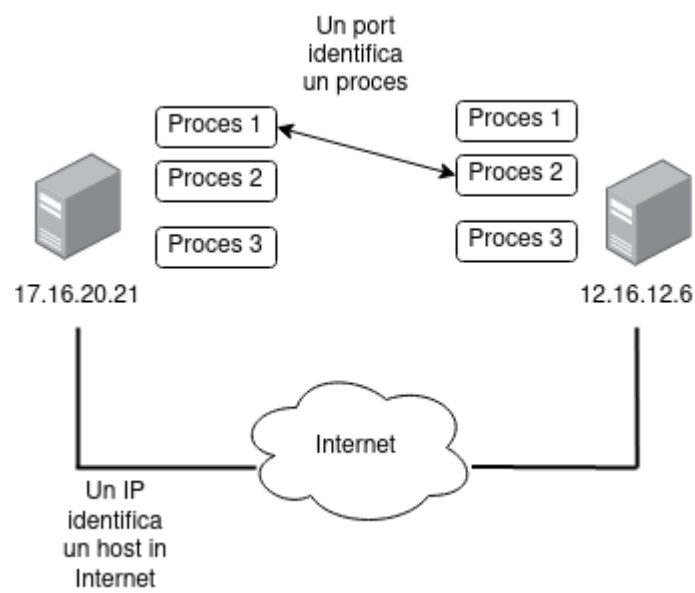
Nivelul retea nu garanteaza:

- transmiterea datelor
- nu poate detecta erori de transmisie
- nu pastreaza ordinea de transmitere

Nivelul Transport

Totate aceste lipsuri ale **nivelului retea** sunt rezolvate de catre protocoalele de **nivel transport**.

In general, implementarea protocoalelor de nivel transport se face in sistemul de operare.



Porturi

Porturile sunt concepute ce ne ajuta sa facem multiplexare intre aplicatii

In contextul retelelor de comunicatie, un **port** este un **numar** asociat unui **socket** dintr-un proces (nu unui host).

Daca un proces doreste sa comunice cu alte procese, acesta va expune un port, o locatie logica prin care accepta conexiuni sau prin care se realizeaza schimbul de date.

Aceste numere permit aplicatiilor sa partajeze concurent resursele de retea. Serverul de mail, de exemplu, nu asteapta terminarea altor procese ce implica retea (ex. web surfing) pt a putea trimite un mail la destinatie.

In antetul protocoalelor de nivel transport, portul este reprezentat pe 2 bytes: `uint16_t port;`.

Mai multe porturi au fost rezervate in procesul de standardizare. Astfel, in **RFC 1340** gasiti o lista de porturi care sunt considerate ca fiind rezervate (well-known) pt anumite protocoale. De exemplu, portul 21 este rezervat pt **File Transfer Protocol (FTP)**.

Port	Protocol	Use
20, 21	FTP	File Transfer
23	Telnet	Remote Login
25	SMTP	Email
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
90	HTTP	World Wide Web
110	POP_3	Remote email access
119	NNTP	USENET NEWS

UDP (User Datagram Protocol)

Serviciu **neorientat pe conexiune**: nu se stabileste o conexiune intre client si server. Asadar, serverul nu va astepta apeluri de conexiune, ci asteapta direct datagrame de la clienti. Acest tip de comunicare este intalnit in sistemele client-server in care se transmit putin mesaje si in general prea rar pentru a mentine o conexiune activa intre cele doua entitati.

UDP nu garanteaza:

- ordinea primirii mesajelor
- corectarea pierderilor pachetelor

UDP-ul se utilizeaza mai ales in retelele in care exista o pierdere foarte mica de pachete si in cadrul aplicatiilor pentru care peirderea unui pachet nu este foarte importnanta.

exemplu: aplicatiile de streaming video

Are un overhead foarte mic, in comparatie cu celelalte protocoale de transport.

header-ul UDP are 8 bytes

header-ul TCP are 20 bytes

Header UDP

Header-ul UDP are 8 bytes si are urmatoarea structura:

0	7	8	15	16	23	24	31
+-----+-----+-----+-----+							
Source				Destination			
Port				Port			
+-----+-----+-----+-----+							
Length				Checksum			
+-----+-----+-----+-----+							

Portul sursa este ales random de catre masina sursa a pacheteului dintre porturile libere existente pe acea masina.

Este un numar pe 16 biti, intre **0** si **65535**. Identifica procesul UDP care a trimis datagrama.

Portul destinatie este protul pe care masina destinatie poate reception pachete. Identifica socket-ul UDP care va procesa datele primite.

Length este lungimea in octeti (byte) a datagramei (header size + data size).

Checksum este valoarea sumei de verificare pentru datagrama.

Putem folosi urmatoarea structura pentru a repreentata header-ul UDP

```
struct udphdr
{
    uint16_t sport;      /* source port */
    uint16_t dport;      /* destination port */
    uint16_t ulen;       /* udp length */
    uint16_t sum;        /* udp checksum */
};
```

Sockets

In cadrul laboratorului nu vom implementa protocolul **UDP**, ci vom folosi implementarea existanta din Kernel-ul de Linux.

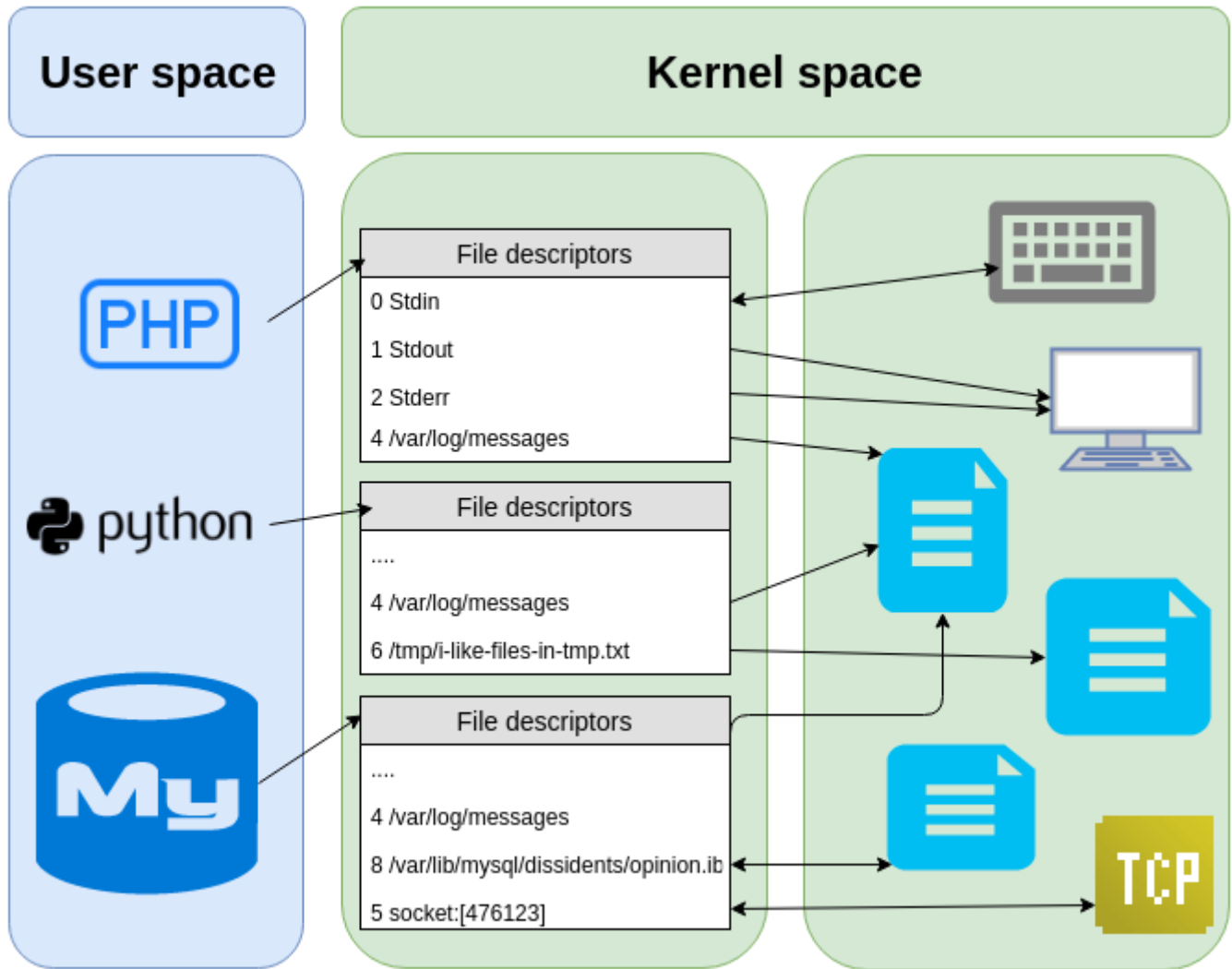
Acest lucru se realizeaza prin intermediul **API**-ului de sockets. Network stack-ul din Linux se ocupa de parsarea si interactiunea cu datagrame **UDP**, noua returnandu-ne doar continutul datagramei.

Un **socket** este un **canal gerenzat de comunicare intre procese**.

Un **socket** este reprezentat in Linux/UNIX printr-un **descriport de fisiere**.

Un **socket** ofera posibilitatea de **comunicare intre procese aflata pe masini diferite intr-o retea**

API-ul de sockets poate fi folosit si pentru **IPC** (Inter-Process Communication) intre procese ce ruleaza pe aceeasi calculator, prin specificarea adresei de loopback sau a unei interfete existente pe masina.



Funcții pt **socket**:

- `socket()`
- `bind()`
- `recvfrom()`
- `sendto()`
- `close()`
- `shutdown()`

NAME

`socket` - create an endpoint for communication

SYNOPSIS

```
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

NAME

`bind` - bind a name to a socket

SYNOPSIS

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

NAME

send, sendto, sendmsg - send a message on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

NAME

close - close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```



```
$ # Linux Programmer's Manual
$ man socket
$ man bind
$ man recvfrom
$ man sendto
$ man close
```

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Comunicare client-server UNIX

Intr-o arhitectura client-server, clientul trimite request-uri (cere resurse) catre server, iar acesta din urma trimite anapoi un raspuns (cu resursa).

Pasi urmati pentru a schimba mesaje folosind **UDP** la **nivelul Transport** folosind API-ul de sockets sunt urmatoarii:

1. **socket()** = Deschide un socket
2. **bind()** = Asociaza un **port** (si o **adresa IP**) pt un socket deschis
3. **recvfrom()** / **sendto()** = receptioneaza / trimite data
4. **close()** = inchide socket-ul

shutdown() = permite intreruperea comunicatiei selectiv, schimbând modul de utilizare a legaturii **full-duplex**



socket()

```
#include <sys/types.h>
#include <sys/socket.h>

/* creare socket in C */
/* int socket(int domain, int type, int protocol); */

/* pentru UDP, folosim un socket de tip SOCK_DGRAM */
int sockid = socket(PF_INET, SOCK_DGRAM, 0);

if (sockid == -1) {
    /* trateaza eroare */
}
```

Explicatii:

- **sockid** - file descriptor pentru socket. În caz de eroare se întoarce -1 si se seteaza variabila **errno**.
- **domain** - reprezintă familia de protocoale pe care urmează să le utilizăm în transferul informației. Vom folosi valorile PF_INET pentru IPv4 sau PF_INET6 pentru IPv6.
- **type** - reprezinta tipul socketului. Valori uzuale:
 - **SOCK_STREAM** - Indicata stabilirea unei comunicatii bazata pe construirea unei conexiuni între sursa si destinatie. Comunicatia este FIFO, fiabila si sigura, o vom folosi la laboratorul urmator cu TCP.
 - **SOCK_DGRAM** - Oferă un flux de date bidirectional, care nu promite sa fie sigur, in secventa sau neduplicat. Un proces care receptioneaza mesaje pe un socket datagrama, poate gasi mesaje duplicate si posibil intr-o ordine diferita fata de cea in care au fost trimise. protocol - specifica protocolul de transport utilizat. Vom seta pe valoarea 0, pentru a se alege protocolul corect in functie de type.

Pentru a afla mai multe informatii, putem accesa urmatorul capitol [5.2 socket\(\)—Get the File Descriptor!](#).

bind()

Utilizata in server pentru a lega un **socket** de un **port** si eventual de o anumita **adresa IP**.

Practic, **bind** este folosit pentru a indica implementarii de networking din Kernel sa lege acel socket la un anumit port si (optional) la o anumita **adresa IP**.

Atfel, stiva va trimite catre acel socket doar datagramele ce au ca port destinatie portul ales.

```
#include <sys/types.h>
#include <sys/socket.h>

/*int bind(int sockfd, struct sockaddr *my_addr, int addrlen)*/

struct sockaddr myaddr;
memset(&myaddr, 0, sizeof(servaddr));
myaddr.sin_family = AF_INET; // IPv4
/* INADDR_ANY = 0.0.0.0 as uint32 */
```

```
myaddr.sin_addr.s_addr = INADDR_ANY;
myaddr.sin_port = htons(atoi(8888));

int rs = bind(sockfd, myaddr, sizeof(servaddr));
/* in urma apelului, sockfd va avea adresa my_addr */
if (rs == -1) {
    /* trateaza eroare */
}
```

Explicatii:

- `sockfd` = Descriptorul de fisier returnat de `socket()`
- `my_addr` = Structura `sockaddr` ce contine informatii despre **adresa IP** si **port**
- `addr len` = lungimea structurii ce stocheaza adresa `my_addr`

`recvfrom()` / `sendto()`

Funcțiile sunt folosite pentru a primi/trimite o datagrama peste un socket.

```
#include <sys/types.h>
#include <sys/socket.h>

struct sockaddr to;
// Filling server information
memset(&to, 0, sizeof(servaddr));
to.sin_family = AF_INET;
to.sin_port = htons(8888);
int rc = inet_aton("127.0.0.1", &to.sin_addr);

int byteswrite = sendto(int sockfd, char *buff, int nbytes, int flags,
struct sockaddr *to, int addrlen);
if (byteswrite == -1) {
    /* trateaza eroare */
}

/* from va fi populata de apelul recvfrom si va contine informatii despre
cine a trimis datagrama catre noi */
struct sockaddr from;
int bytesread = recvfrom(int sockfd, char *buff, int nbytes, int flags,
struct sockaddr *from, int *addrlen);
if (bytesread == -1) {
    /* trateaza eroare */
}
```

Explicatii:

- `sockfd` = Descriptorul de fisier returnat de `socket()`
- `buff` = Bufferul unde se gasesc datele ce urmeaza a fi trimise/bufferul unde se vor receptiona datele
- `flags` = Precizeaza conditii de efectuare a transmisiei

- `to/from` = structura ce indica adresa unde se trimite/primesc date
- `addr len` = lungime structurii `to/from` in octeti

`close()` / `shutdown()`

Pentru a inchide un socket se foloseste functia de inchidere a unui descriptor de fisier din Unix:

```
#include <unistd.h>

int close(int fd);
```

Acest lucru va impiedica atat realizarea de alte citiri, cat si de scrieri din socket. Pentru mai mult control asupra socketului, se foloseste functia `shutdown`, care permite intreruperea comunicatiei selectiv, schimbând modul de utilizare a legaturii `full-duplex`.

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

NOTA `shutdown()` nu inchide un descriptor de fisier, ci doar schimba modul de utilizare

Resursele trebuie eliberate folosind `close()`

Stop-And-Wait peste UDP

TODO

Frereastra Glisanta peste UDP

TODO

Dimensiunea ferestrei glisante

TODO

Lab 6. Retransmisie peste UDP. Go-Back-N ARQ

TODO

Lab 7. Protocolul TCP. Multiplexare IO.

TODO

Lab 8. TCP Congestion Control

TODO

Lab 9. Protocolul HTTP

Link lab: <https://pcom.pages.upb.ro/labs/lab9/lecture.html>