

---

# *Computer Networks*

## **Lab 2 - Computer Networks**

*Manuel Ricardo, Filipe B. Teixeira, Eduardo Nuno Almeida*

*Universidade do Porto*

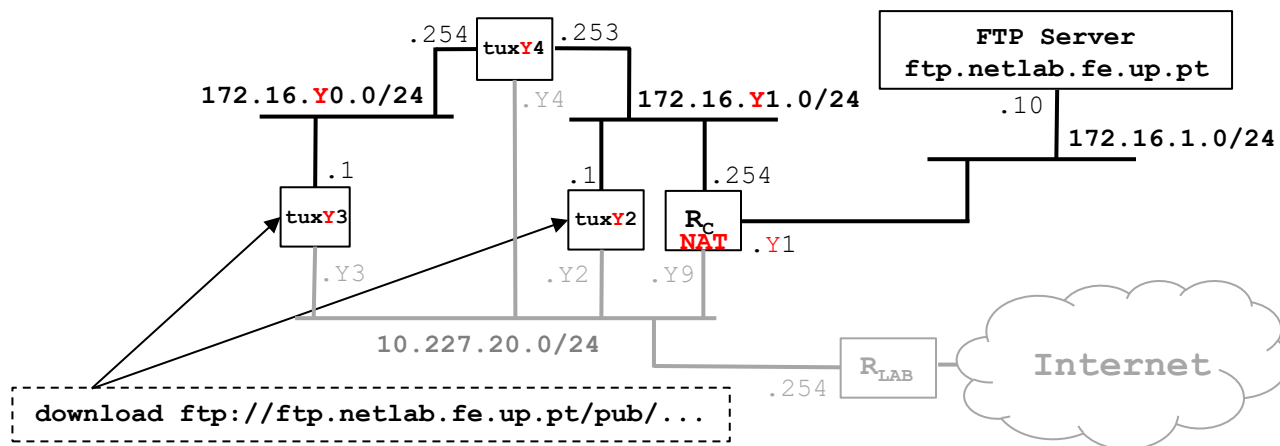
# *Lab Work - Two parts*

---

- ◆ Part 1 - Development of a download application

**download ftp://ftp.netlab.fe.up.pt/pub/...**

- ◆ Part 2 - Configuration and study of a computer network



# *Evaluation*

---

- ♦ Organization
  - » Groups of 2 students
- ♦ Evaluation criteria
  - » Participation during class (continuous evaluation)
  - » Presentation and demonstration of the work
  - » Individual 15-minute quiz to be answered in the classroom, on the last class before the presentation
  - » Final report
- ♦ Demonstration of the work
  - » Replicate the network topology described in Part 2 / Exp 6
  - » Using different IP addresses for the Bridges (announced before the demonstration starts)

---

## *Part 1 - Development of a download application*

# *Development of an Application*

---

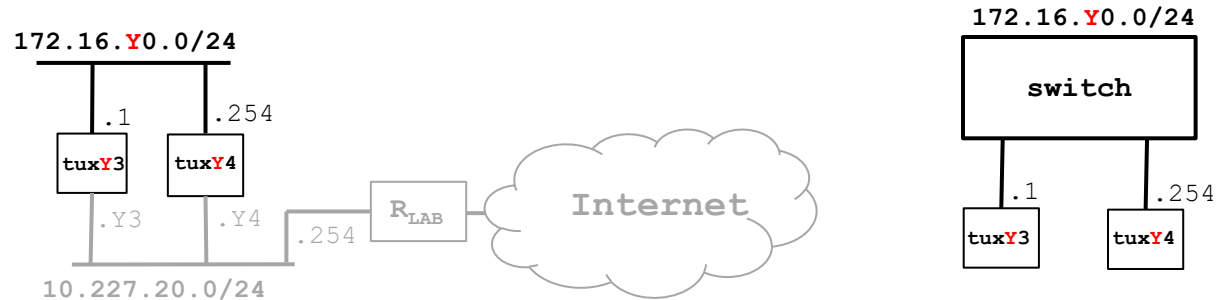
- ◆ Develop application `download ftp://ftp.netlab.fe.up.pt/pub/...`
  - » Application downloads a single file
  - » Implements FTP application protocol, as described in RFC959
  - » Adopts URL syntax, as described in RFC1738  
`ftp://[<user>:<password>@]<host>/<url-path>`
- ◆ Steps
  - » Experiments using Telnet application (Telnet, SMTP, POP, HTTP and FTP); focus on FTP
  - » Specification/design of a download application
    - unique use case: connect, login host, passive, get path, success (file saved in CWD) or un-success (indicating failing phase)
    - challenging programming aspects: gethostbyname, sockets, control connection, passive, data connection
  - » Implement a very simple FTP client at home
    - reuse existing programs: `clientTCP.c`, `getIP.c`
- ◆ Learning objectives
  - » Describe client - server concept and its peculiarities in TCP/IP
  - » Characterize application protocols in general, characterize URL, describe in detail the behaviour of FTP
  - » Locate and read RFCs
  - » Implement a simple FTP client in C language
  - » Use sockets and TCP in C language
  - » Understand service provided DNS and use it within a client program

---

## *Part 2 - Configuration and Study of a Network*

# Part 2 / Exp 1- Configure an IP Network

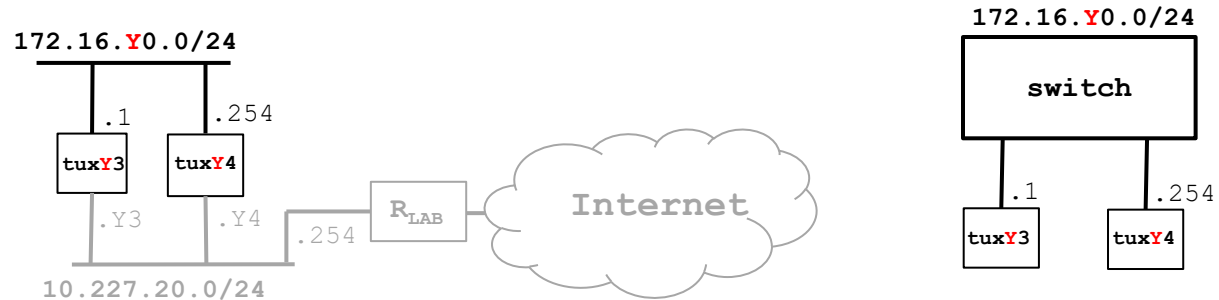
## Steps



1. Connect E1 of tuxY3 and E1 of tuxY4 to the switch
2. Configure eth1 interface of tuxY3 and eth1 interface of tuxY4 using **ifconfig** and **route** commands
3. Register the IP and MAC addresses of the network interfaces
4. Use **ping** command to verify connectivity between these computers
5. Inspect forwarding (**route -n**) and ARP (**arp -a**) tables
6. Delete ARP table entries in tuxY3 (**arp -d ipaddress**)
7. Start **Wireshark** in tuxY3.eth1 and start capturing packets
8. In tuxY3, **ping** tuxY4 for a few seconds
9. Stop capturing packets
10. Save the log and study it at home

# *Part 2 / Exp 1- Configure an IP Network*

---



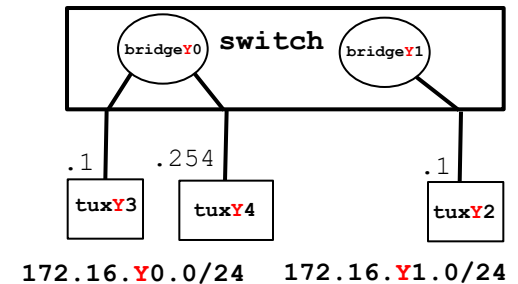
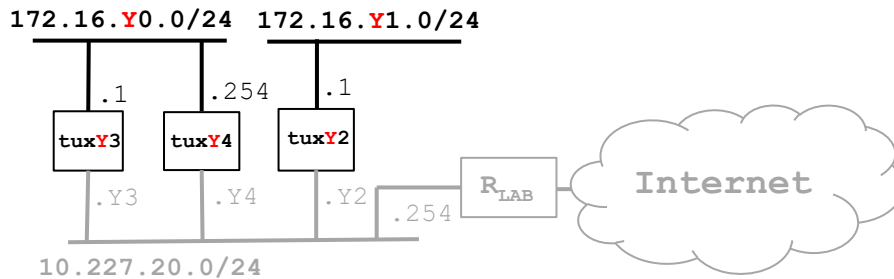
## Questions

- » What are the ARP packets and what are they used for?
- » What are the MAC and IP addresses of ARP packets and why?
- » What packets does the ping command generate?
- » What are the MAC and IP addresses of the ping packets?
- » How to determine if a receiving Ethernet frame is ARP, IP, ICMP?
- » How to determine the length of a receiving frame?
- » What is the loopback interface and why is it important?



# Part 2 / Exp 2 - Implement two bridges in a switch

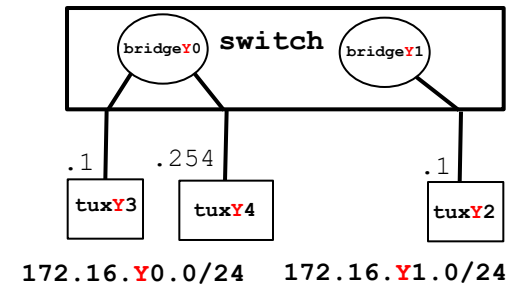
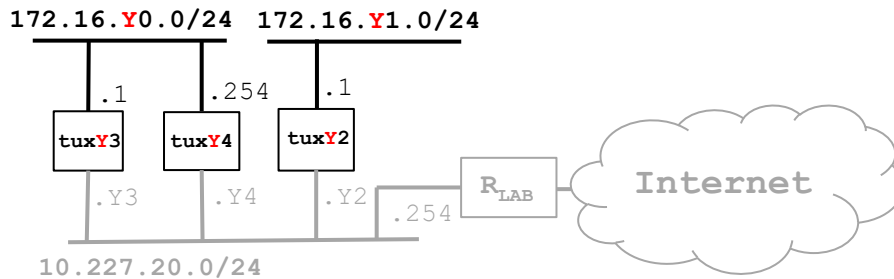
## Steps



1. Connect and configure E1 of tuxY2 and register its IP and MAC addresses
2. Create two **bridges** in the switch: bridgeY0 and bridgeY1
3. Remove the ports where tuxY3, tuxY4 and tuxY2 are connected from the default bridge (**bridge**) and add them the corresponding ports to bridgeY0 and bridgeY1
4. Start the capture at tuxY3.eth1
5. In tuxY3, ping tuxY4 and then ping tuxY2
6. Stop the capture and save the log
7. Start new captures in tuxY2.eth1, tuxY3.eth1, tuxY4.eth1
8. In tuxY3, do ping broadcast (**ping -b 172.16.Y0.255**) for a few seconds
9. Observe the results, stop the captures and save the logs
10. Repeat **steps 7, 8** and **9**, but now do
  - ping broadcast in tuxY2 (**ping -b 172.16.Y1.255**)

# Part 2 / Exp 2 - Implement two bridges in a switch

---

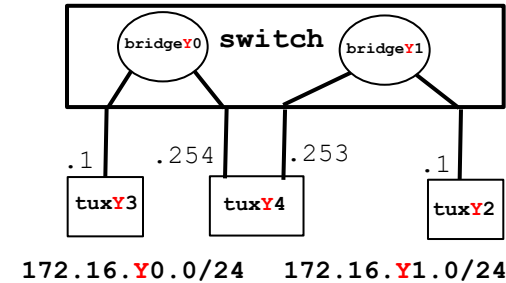
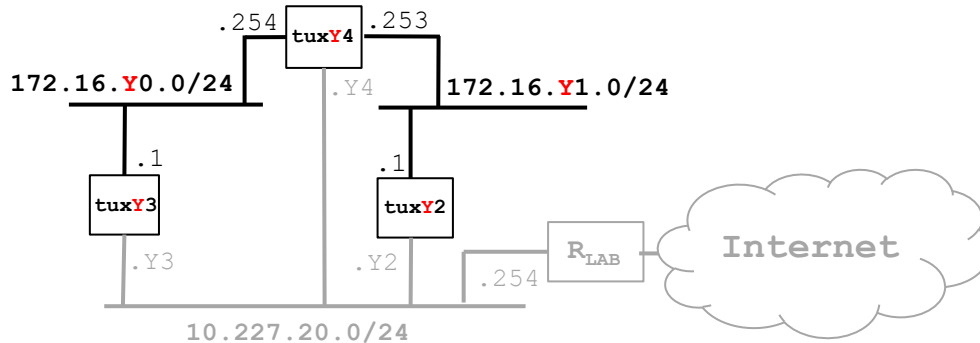


## Questions

- » How to configure bridgeY0?
- » How many broadcast domains are there? How can you conclude it from the logs?

# Part 2 / Exp 3 - Configure a Router in Linux

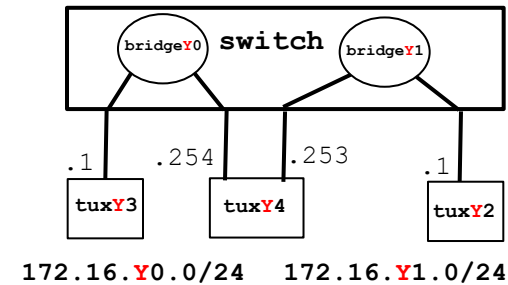
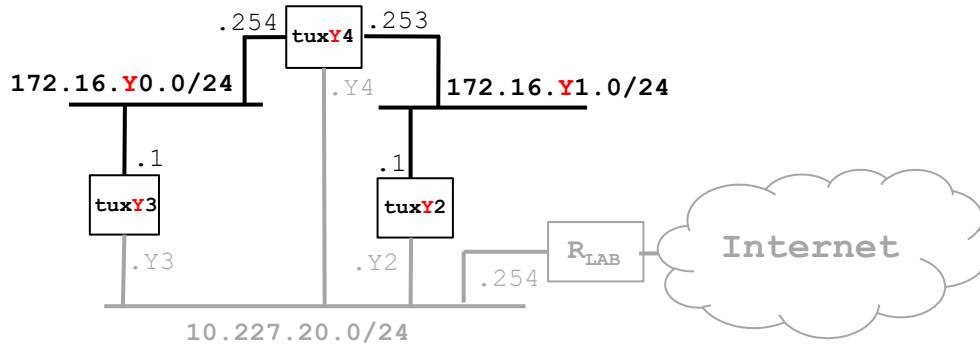
## Steps



1. Transform tuxY4 (Linux) into a router
  - Configure also eth2 interface of tuxY4 and add it to bridgeY1
  - Enable IP forwarding
  - Disable ICMP echo-ignore-broadcast
2. Observe MAC addresses and IP addresses in tuxY4.eth1 and tuxY4.eth2
3. Reconfigure tuxY3 and tuxY2 so that each of them can reach the other
4. Observe the routes available at the 3 tuxes (***route -n***)
5. Start capture at tuxY3
6. From tuxY3, ping the other network interfaces (172.16.Y0.254, 172.16.Y1.253, 172.16.Y1.1) and verify if there is connectivity
7. Stop the capture and save the logs

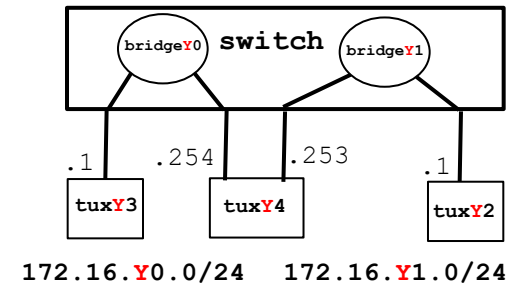
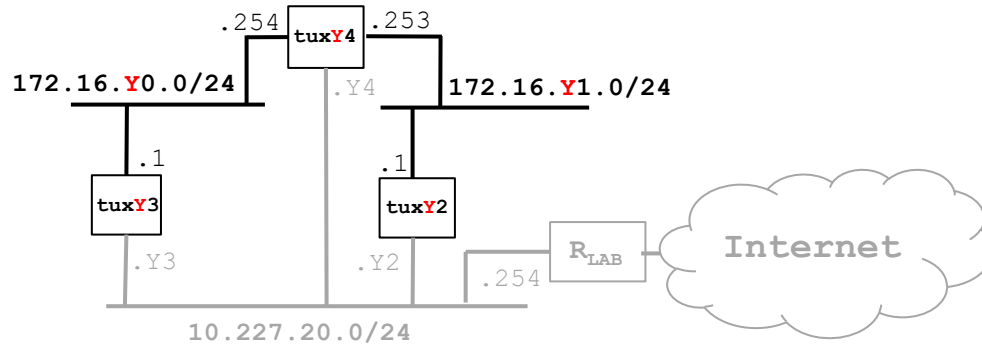
# Part 2 / Exp 3 - Configure a Router in Linux

## Steps



8. Start capture in tuxY4; use 2 instances of Wireshark, one per network interface
9. Clean the ARP tables in the 3 tuxes
10. In tuxY3, ping tuxY2 for a few seconds.
11. Stop captures in tuxY4 and save logs

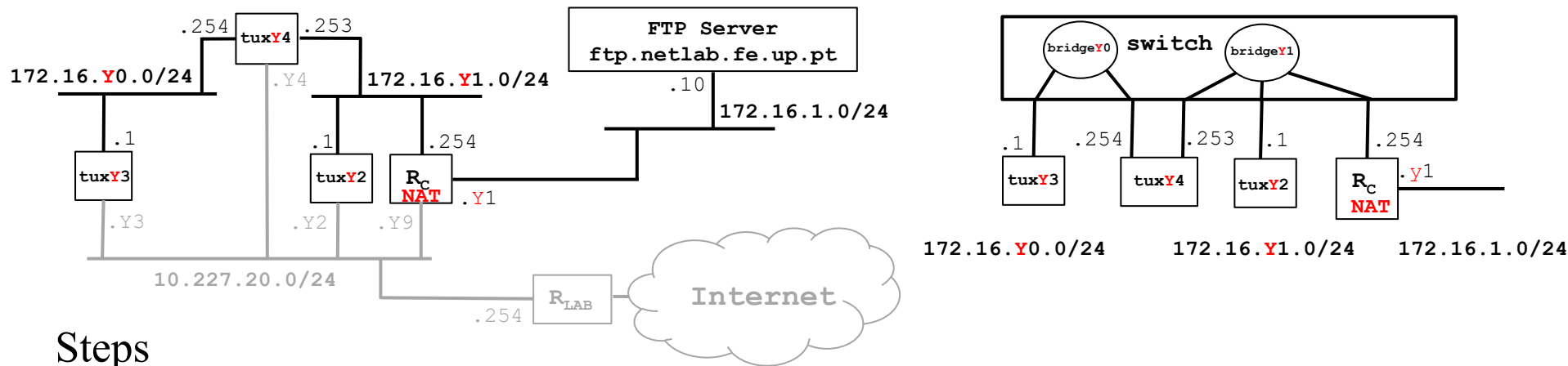
# Part 2 / Exp 3 - Configure a Router in Linux



## Questions

- » What routes are there in the tuxes? What are their meaning?
- » What information does an entry of the forwarding table contain?
- » What ARP messages, and associated MAC addresses, are observed and why?
- » What ICMP packets are observed and why?
- » What are the IP and MAC addresses associated to ICMP packets and why?

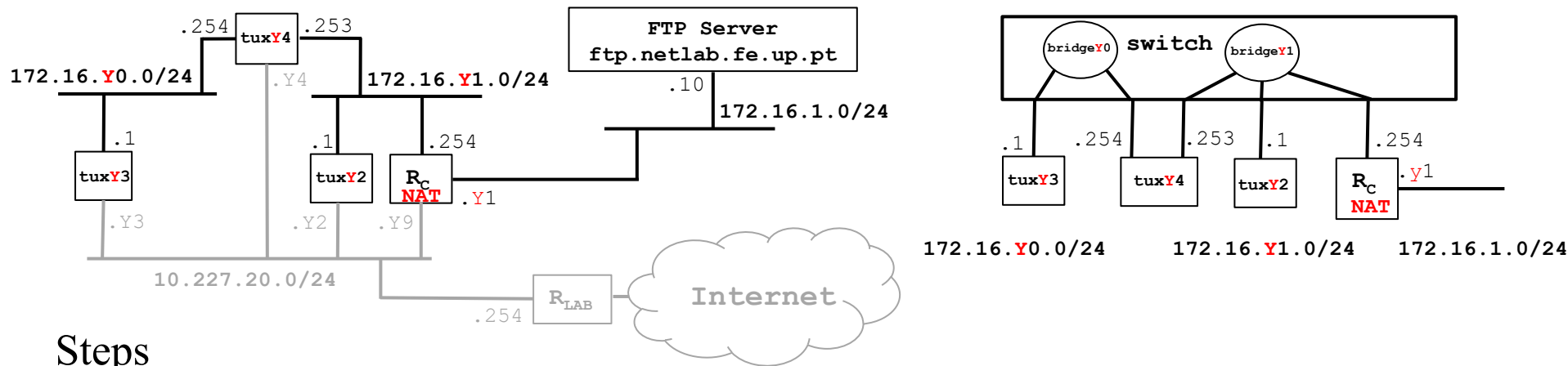
# Part 2 / Exp 4 - Configure a Commercial Router and Implement NAT



## Steps

1. Connect ether1 of **R<sub>C</sub>** to the *lab network* on **PY.12** (with NAT enabled by default) and ether2 of **R<sub>C</sub>** to a port on bridgeY1. Configure the IP addresses of **R<sub>C</sub>** through the router serial console
2. Verify routes
  - **tuxY4** as **default router** of **tuxY3**;
  - **R<sub>C</sub>** as **default router** for **tuxY2** and **tuxY4**
  - in **tuxY2** and **R<sub>C</sub>** add routes for 172.16.Y0.0/24
3. Using **ping** commands and **Wireshark**, verify if **tuxY3** can ping all the network interfaces of **tuxY2**, **tuxY4** and **R<sub>C</sub>**

# Part 2 / Exp 4 - Configure a Commercial Router and Implement NAT

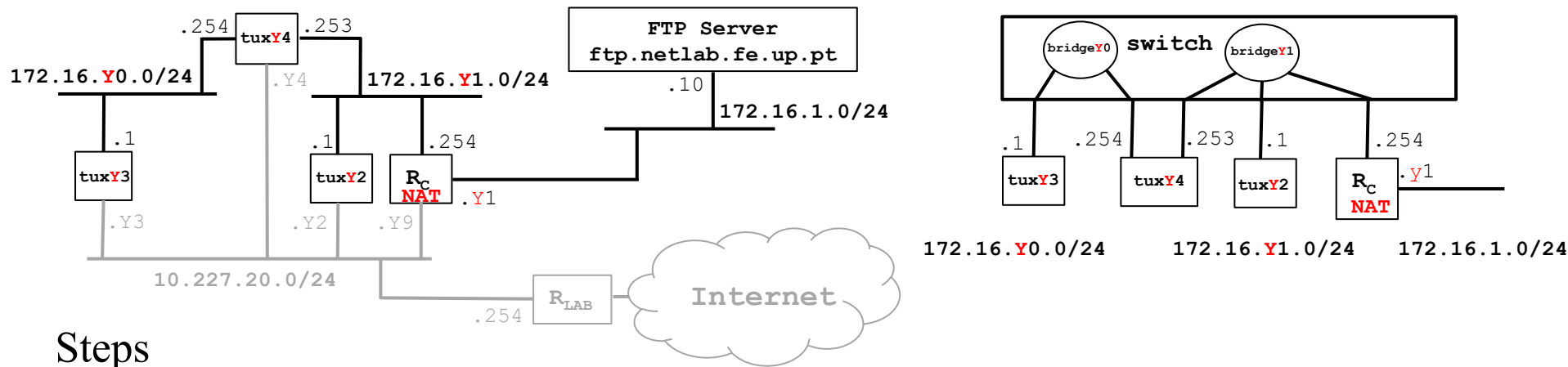


## Steps

### 4. In tuxY2

- Do the following:
  - `sysctl net.ipv4.conf.eth0.accept_redirects=0`
  - `sysctl net.ipv4.conf.all.accept_redirects=0`
- remove the route to `172.16.Y0.0/24` via tuxY4
- In tuxY2, ping tuxY3
- Using capture at tuxY2, try to understand the path followed by ICMP ECHO and ECHO-REPLY packets (look at MAC addresses)
- In tuxY2, do **traceroute tuxY3**
- In tuxY2, add again the route to `172.16.Y0.0/24` via tuxY4 and do **traceroute tuxY3**
- Activate the acceptance of ICMP redirect at tuxY2 when there is no route to `172.16.Y0.0/24` via tuxY4 and try to understand what happens

# Part 2 / Exp 4 - Configure a Commercial Router and Implement NAT

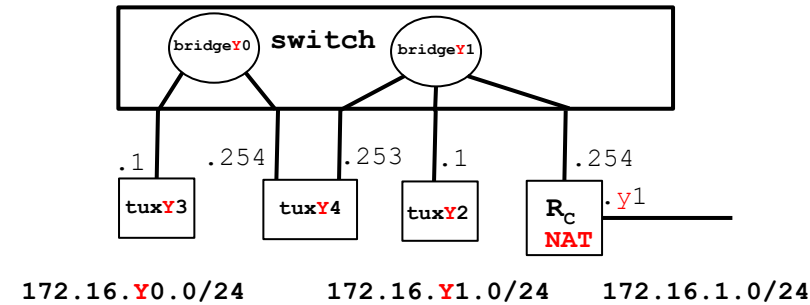
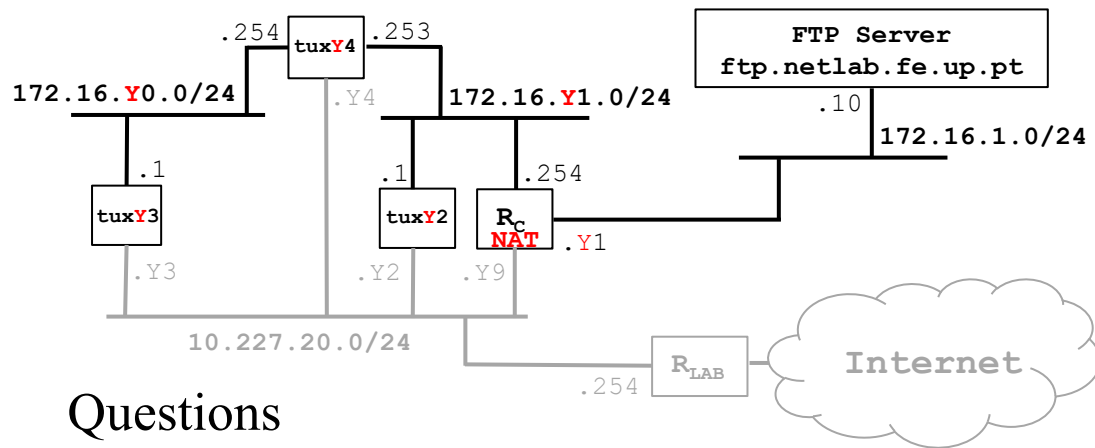


## Steps

5. In tuxY3, ping the FTP server (172.16.1.10) and try to understand what happens
6. Disable **NAT** functionality in router **R<sub>C</sub>**
7. In tuxY3 ping 172.16.1.10, verify if there is connectivity, and try to understand what happens



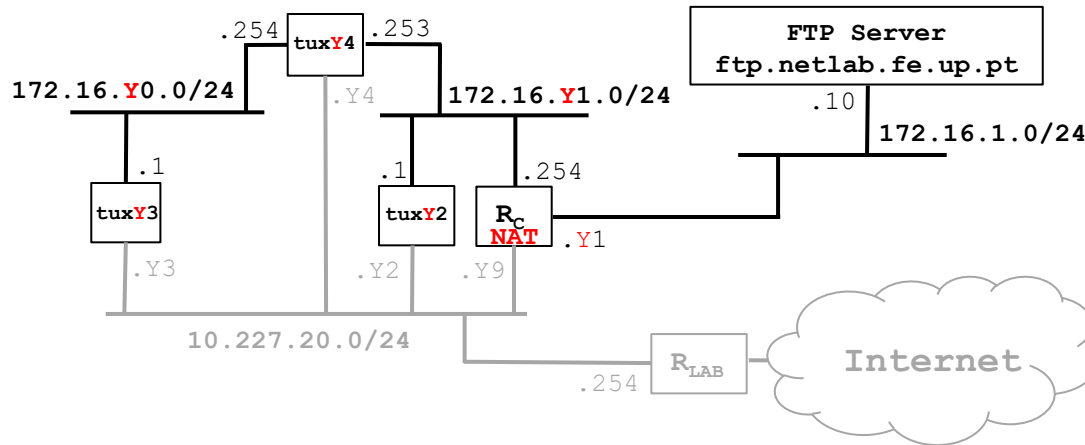
# Part 2 / Exp 4 - Configure a Commercial Router and Implement NAT



## Questions

- » How to configure a static route in a commercial router?
- » What are the paths followed by the packets, with and without ICMP redirect enabled, in the experiments carried out and why?
- » How to configure NAT in a commercial router?
- » What does NAT do?
- » What happens when tuxY3 pings the FTP server with the NAT disabled? Why?

## Part 2 / Exp 5 - DNS



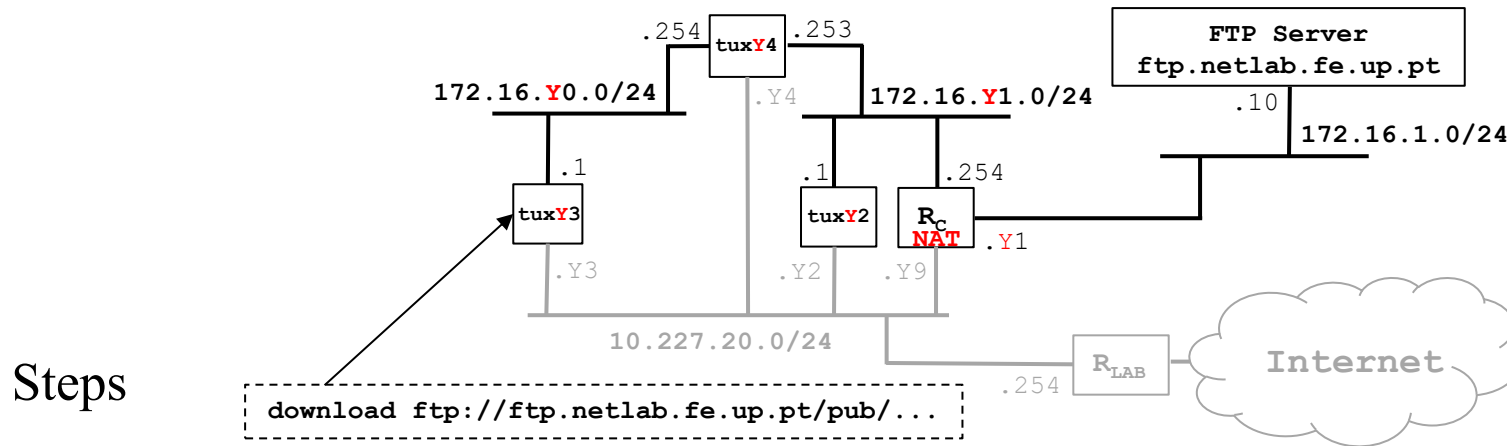
### Steps

1. Configure DNS at tuxY3, tuxY4, tuxY2 (use DNS server *services.netlab.fe.up.pt* (10.227.20.3))
2. Verify if names can be used in these hosts (e.g., **ping** hostname, use browser)
3. Execute **ping (new-hostname-in-the-Internet)**; observe DNS related packets in **Wireshark**

### Questions

- » How to configure the DNS service in a host?
- » What packets are exchanged by DNS and what information is transported

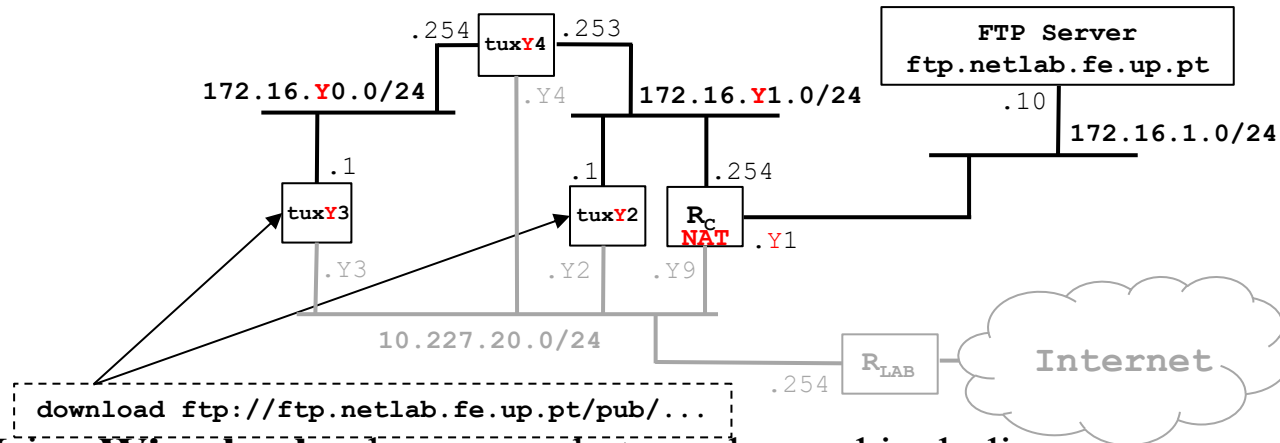
## Part 2 / Exp 6 - TCP connections



1. Compile your download application in tuxY3
2. In tuxY3, restart capturing with Wireshark and run your application
3. Verify if file has arrived correctly, stop capturing and save the log

## Part 2 / Exp 6 - TCP connections

### Steps



4. Using **Wireshark**, observe packets exchanged including:
  - TCP control and data connections, and its phases (establishment, data, termination)
  - Data transferred through the FTP control connection
  - TCP ARQ mechanism
  - TCP congestion control mechanism in action
  - Note: use also the Wireshark Statistics tools (menu) to study the TCP phases, ARQ and congestion control mechanism
5. Repeat the download in tuxY3 but now, in the middle of the transfer, start a new download in tuxY2
  - Use the Wireshark statistics tools to understand how the throughput of a TCP connection varies along the time

## *Part 2 / Exp 6 - TCP connections*

---

### Questions

- » How many TCP connections are opened by your FTP application?
- » In what connection is transported the FTP control information?
- » What are the phases of a TCP connection?
- » How does the ARQ TCP mechanism work? What are the relevant TCP fields? What relevant information can be observed in the logs?
- » How does the TCP congestion control mechanism work? What are the relevant fields. How did the throughput of the data connection evolve along the time? Is it according to the TCP congestion control mechanism?
- » Is the throughput of a TCP data connections disturbed by the appearance of a second TCP connection? How?

# *Workplan*

---

- ♦ 1st class - Part 1
  - » experiments using Telnet, focus on FTP
  - » usage of gethostbyname and socket functions
  - » client architecture; main use case
- ♦ 2nd class - Part 2: Steps 1 e 2
  - » linux, lab, cables, ipconfig, route, arp, wireshark, capture and log analysis
- ♦ Next classes - Part 2: Steps 2, 3, 4, 5 e 6
- ♦ Last week
  - » Demonstration of download application, downloading a file from netlab FTP server (<ftp.netlab.fe.up.pt>)
  - » Delivery of report (use moodle for upload)

# *Final Report*

---

- ◆ Report must contain
  - » Title, Authors, Summary
  - » Introduction
  - » Part 1 - Download application
    - Architecture of the download application
    - Report of a successful download, including print-screen of Wireshark logs showing the FTP packets
  - » Part 2 - Network configuration and analysis
    - For each experiment (1 to 6)
      - Network architecture, experiment objectives, main configuration commands, relevant logs
      - **Analysis of the logs captured that are relevant for the learning objectives**
  - » Conclusions
  - » References
  - » Annexes: code of the download application, configuration commands, logs captured
- ◆ Maximum length of 8 pages A4, font 11pt
- ◆ Upload through Moodle

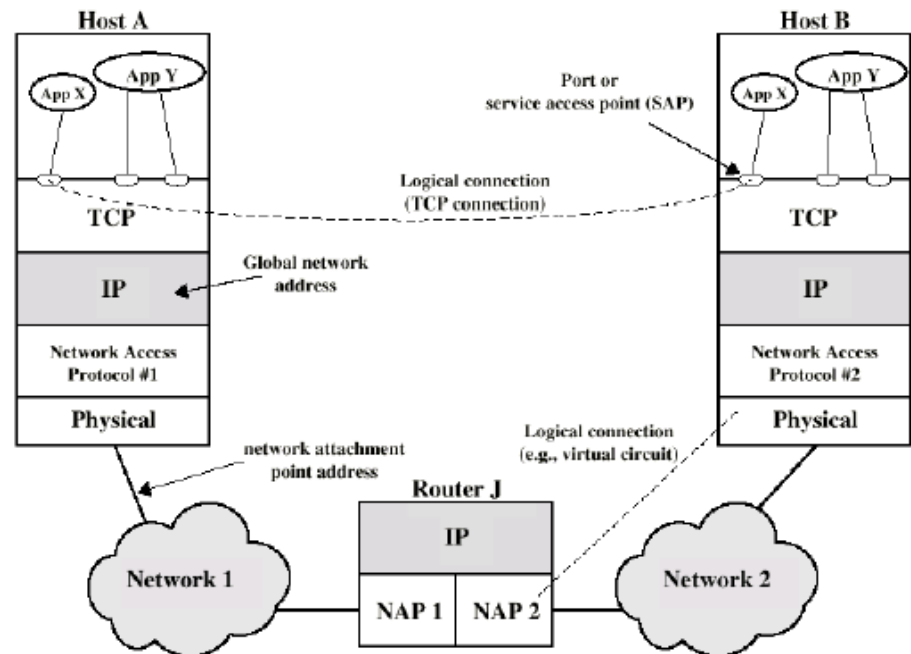
---

# *TCP/IP and Application Protocols*



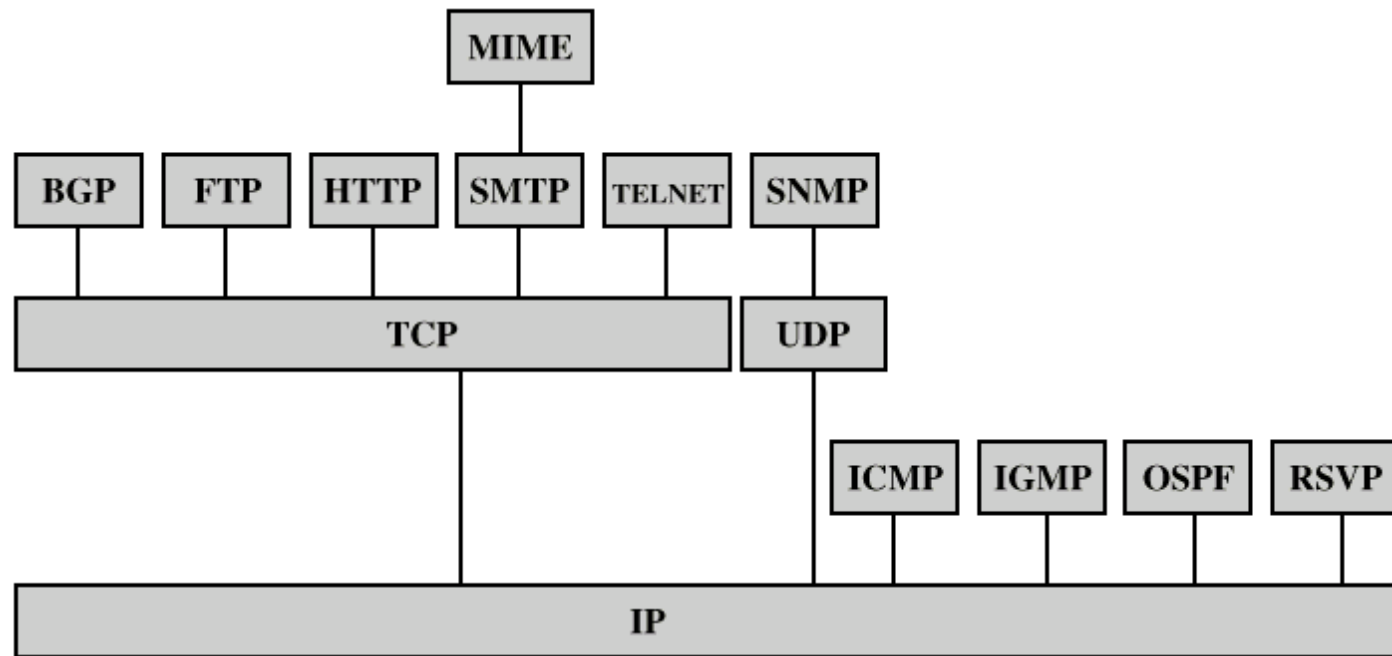
# Some characteristics of TCP/IP

- ◆ IP (*Internet Protocol*) is implemented on all computers (*hosts*) and routers
- ◆ Each computer has a unique IP address on each subnet it belongs to
- ◆ Each process on a computer has a unique address (port)



# *TCP/IP Protocols*

---



**BGP** = Border Gateway Protocol

**FTP** = File Transfer Protocol

**HTTP** = Hypertext Transfer Protocol

**ICMP** = Internet Control Message Protocol

**IGMP** = Internet Group Management Protocol

**IP** = Internet Protocol

**MIME** = Multi-Purpose Internet Mail Extension

**OSPF** = Open Shortest Path First

**RSVP** = Resource ReSerVation Protocol

**SMTP** = Simple Mail Transfer Protocol

**SNMP** = Simple Network Management Protocol

**TCP** = Transmission Control Protocol

**UDP** = User Datagram Protocol

# Demultiplexing

## » TCP/UDP header (port)

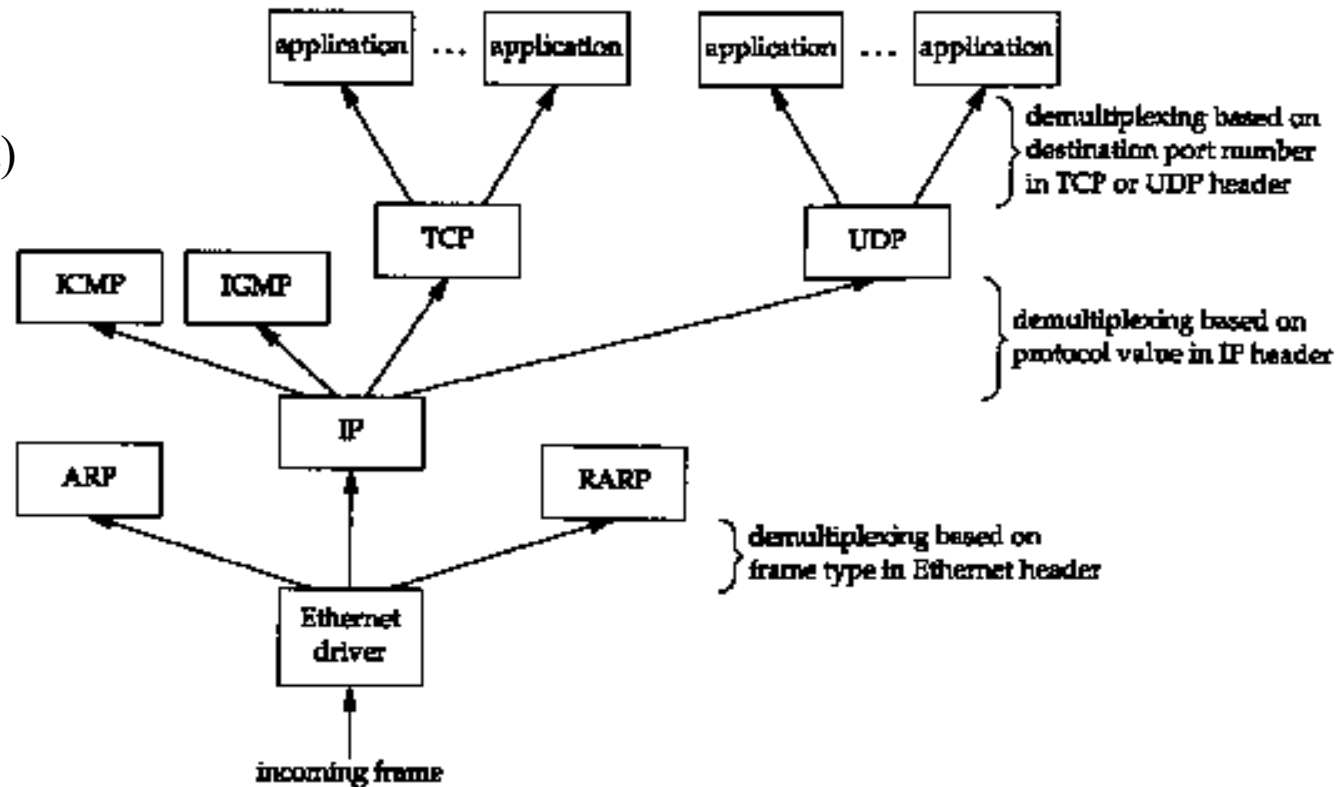
- FTP → 21
- Telnet → 23
- ...

## » IP header (protocol)

- ICMP → 1
- IGMP → 2
- TCP → 6
- UDP → 17

## » Ethernet header (type)

- IP → 0x0800
- ARP → 0x0806
- RARP → 0x8053



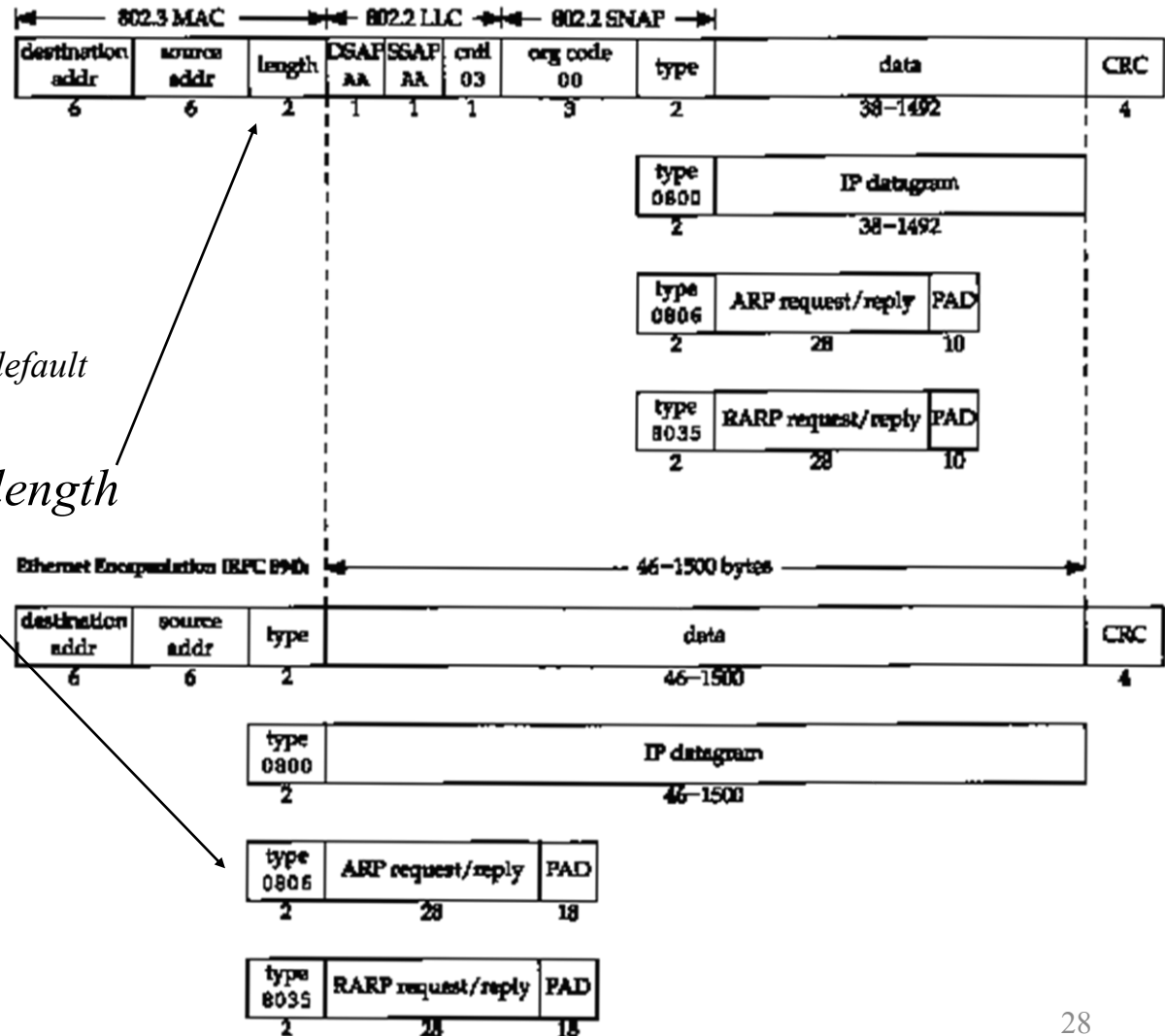
# Ethernet encapsulation

## ◆ Ethernet cards

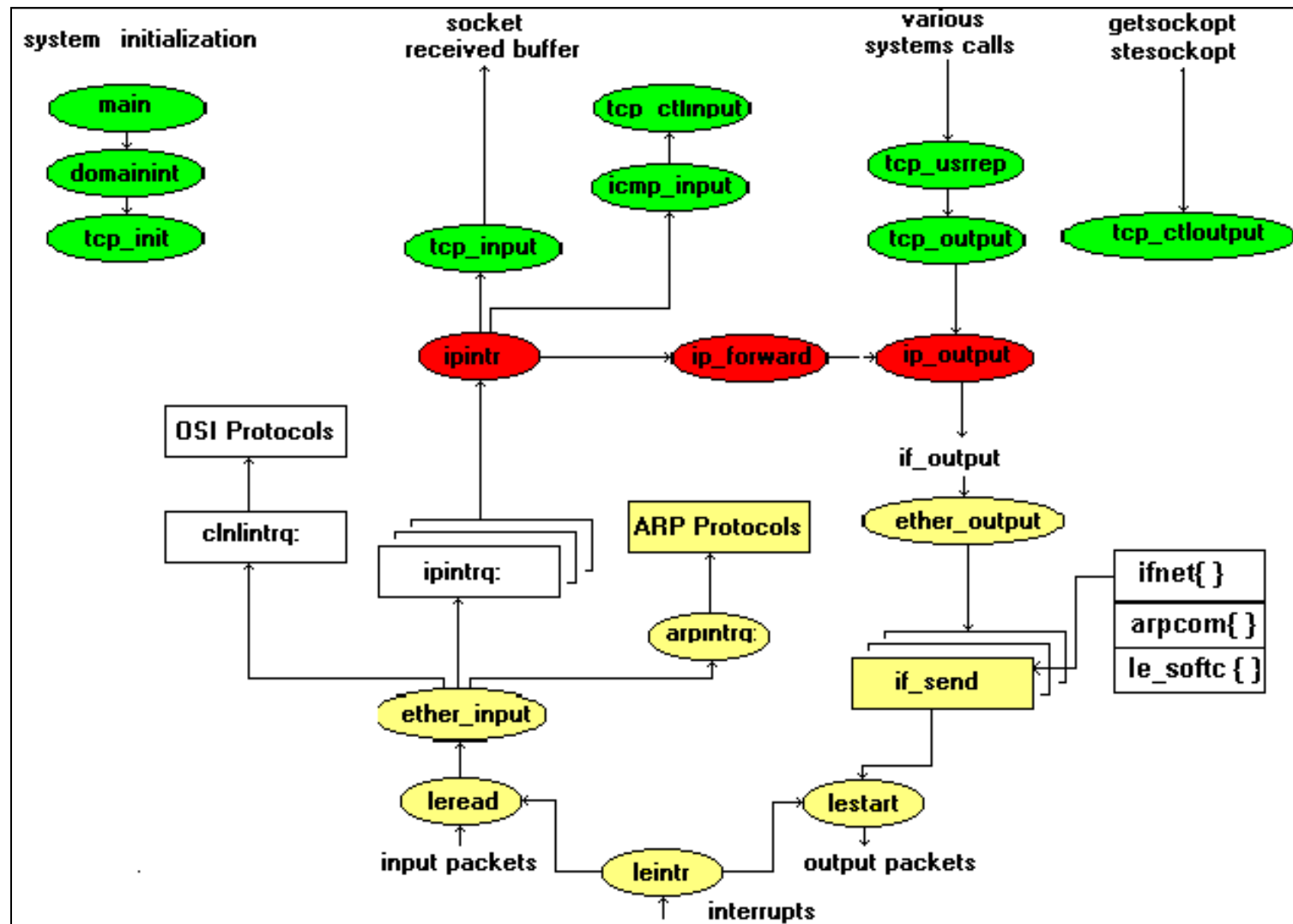
- » Must receive
  - IEEE 802 encapsulation
  - Ethernet encapsulation
- » If able to send the 2 types
  - Ethernet encapsulation → *default*

## ◆ Valid values for *IEEE 802 length*

- » Different from valid type
  - E.g., 0800 = 2048

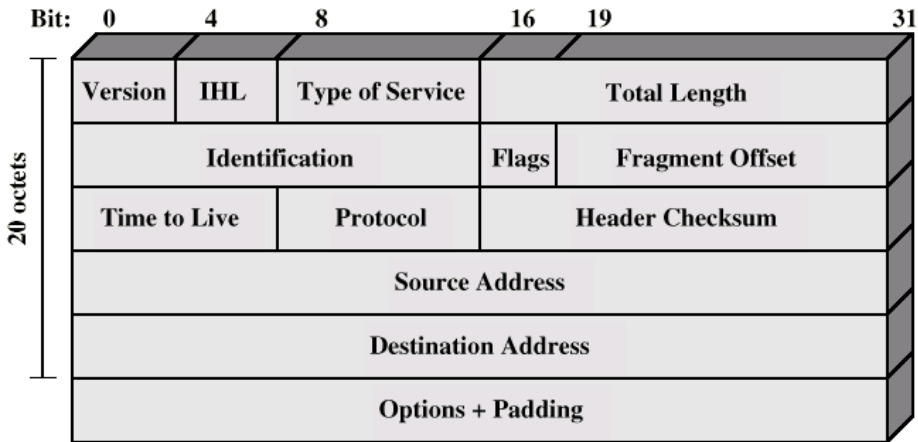
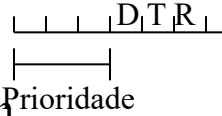


## TCP/IP Stack Functions



# IP Protocol

- » **Version** - protocol version (v4)
- » **IHL** - header length (in 32-bit words); 20..60 octets
- » **Type of Service** - type of service to be provided over the network
- » **Total Length** - total datagram length (max. 65535 octets)
- » **Identification** - identifier common to all fragments of a datagram
- » **DF** - *Don't Fragment*
- » **MF** - *More Fragments*
- » **Fragment Offset**
- » **Time To Live (TTL)** - limits the life of a packet; decremented each time it passes through a router; when it reaches 0 the packet is eliminated



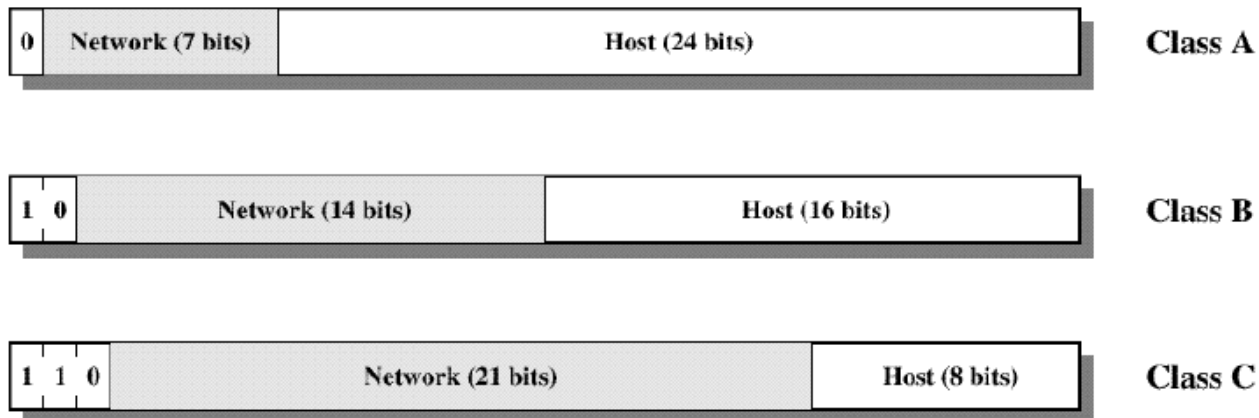
- » Protocol - encapsulated transport layer protocol (e.g., TCP, UDP)
- » Source Address - sender address
- » Destination address - destination address
- » Options - 1 octet identifies the option; 1 octet contains the length (optional); e.g: Record Route

# IP - Addresses

---

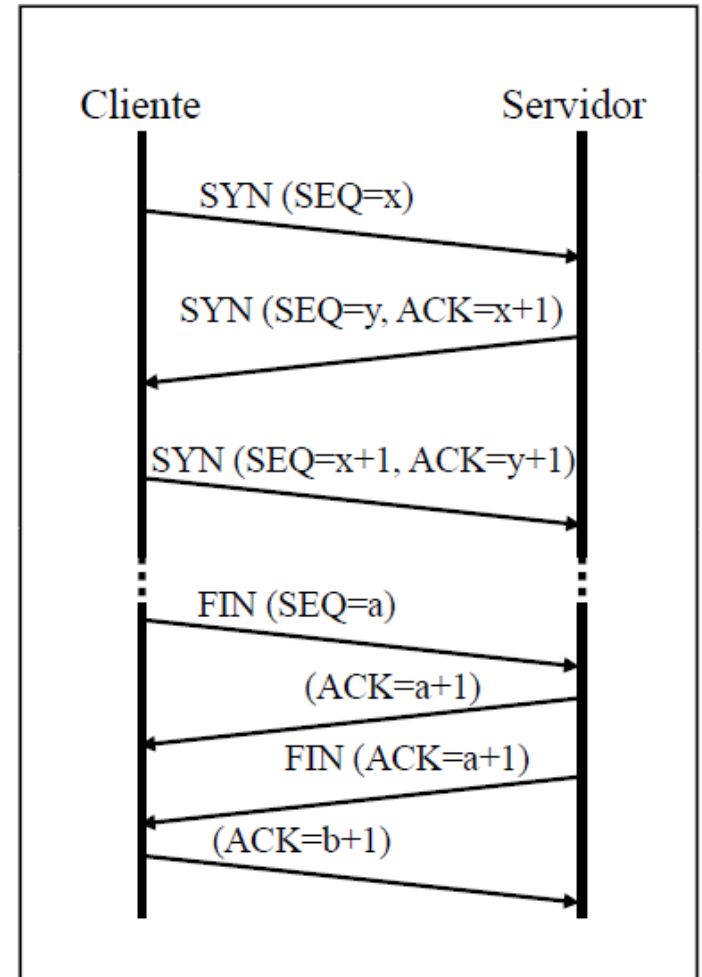
- ♦ 32-bit global addresses, structured in two parts: network (netid) and host (hostid)
  - » Originally addresses were based on classes (A, B, C, D E)
    - Fixed-length network prefix
  - » Classless addresses (CIDR)
    - Variable length network prefix

Classe	Valores
A	0.0.0.0 → 127.255.255.255
B	128.0.0.0 → 191.255.255.255
C	192.0.0.0 → 223.255.255.255
D	224.0.0.0 → 239.255.255.255
E	240.0.0.0 → 247.255.255.255



# TCP - Transmission Control Protocol

- ♦ RFC 793
- ♦ Characteristics
  - » Ensures reliable end-to-end octet flow over unreliable support
  - » Connection oriented protocol
  - » *Full-duplex* connections
  - » Positive Acknowledgment (ACK)
  - » Recovers from losses and errors (retransmissions) after *time-out*
  - » Orderly delivery of data to the application
  - » Flow and congestion control
  - » Multiplexing multiple TCP connections over the same IP address
- ♦ TCP connection establishment
  - » *3-way handshake*
  - » Client-server model





# *TCP - Transmission Control Protocol*

**Source Port** - port of origin

**Destination Port** - port of destination

**Sequence Number** - identifies, in the sender stream, the sequence of octets sent

**Acknowledgement Number** - corresponds to the octet number expected to receive

**HLEN** - the length of the TCP header (in 32-bit words)

**URG** - informs whether the Urgent Pointer field should be interpreted

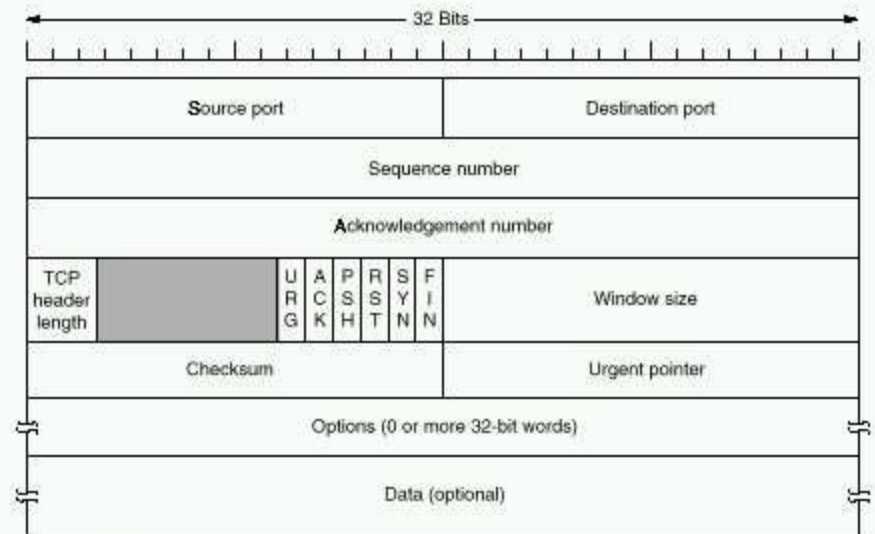
**ACK** - informs whether the Ack number field is valid

**PSH** - allows to disable buffering

**RST** - used to reset a connection

**SYN** - allows to establish a connection

**FIN** - allows to close a connection



The TCP header.

**Window Size** - number of bytes the communication peer can send without acknowledgment (flow control)

**Checksum** - covers header, data, and pseudo-header

# *Berkeley Sockets*

---

- ♦ API - Application Programming Interface
  - » operating system: UNIX
  - » programming language: C
  - » communication protocols
    - TCP/IP
    - UNIX
    - XNS
  - » Address data structures
  - » Primitives: `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, `recvfrom()`, `sendto()`, `close()`
  - » Association - *socket* pair

# Berkeley Sockets

---

## ◆ Address data structures

### » BSD

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

```
struct sockaddr {  
    u_short    sa_family;    /*Address family - ex: AF_INET*/  
    char       sa_data[14]; /*Protocol address*/  
};
```

### » Internet

```
<netinet/in.h>
```

```
struct in_addr {  
    u_long      s_addr;  
};  
  
struct sockaddr_in {  
    short       sin_family;    /*AF_INET*/  
    u_short     sin_port;      /*Port number*/  
    struct in_addr sin_addr;    /*32 bit netid/hosdtid*/  
    char        sin_zero[8];    /*unused*/  
};
```

# *Berkeley Sockets*

---

```
➔ int socket(int family, int type, int protocol)
```

family: AF\_INET, AF\_UNIX

type: SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW

protocol: protocol to use (with the value 0 is determined by the system)

» Returns

- *socket* descriptor
- -1 in case of error

```
➔ int bind(int sockfd, struct sockaddr* myaddr, int addrlen)
```

sockfd: *socket* descriptor

myaddr: local address (IP + port)

addrlen: myaddr structure length

» Returns

- 0 in case of success
- -1 in case of error

» This primitive associates the *socket* with the local address myaddr

# Berkeley Sockets

---

➔ `int connect(int sockfd, struct sockaddr* serveraddr, int addrlen)`

serveraddr: remote server address (IP + port)

» Returns

- 0 in case of success
- -1 in case of error

» TCP: remote server connection establishment

» UDP: *serveraddr* address storage

➔ `int listen(int sockfd, int backlog)`

backlog: number of queued call requests

» Returns

- 0 in case of success
- -1 in case of error

» Primitive specifies the maximum number of queued calls

# Berkeley Sockets

---

➔ `int accept(int sockfd, struct sockaddr* peeraddr, int* addrlen)`

peeraddr: structure used to store the client address (IP + port)

addrlen: pointer to the length of the peeraddr structure

» Returns

- accepted socket descriptor, client address and length
- -1 in case of error

» Primitive listens for connection requests and creates another socket with the same properties as *sockfd*

➔ `int send(int sockfd, const void* buf, int len, unsigned int flags)`

➔ `int recv(int sockfd, void* buf, int len, unsigned int flags)`

buf: pointer to the memory location that contains/will contain the data

flags: MSG\_OOB, MSG\_PEEK, MSG\_DONTROUTE

» Returns

- number of octets written/read
- 0 in case the connection was closed
- -1 in case of error em caso de erro

» These primitives allow sending and receiving data from the network

# Berkeley Sockets

---

```
➔ int sendto(int sockfd, const void* buf, int len,  
             unsigned int flags,  
             struct sockaddr* to, int tolen)  
  
➔ int recvfrom(int sockfd, void* buf, int len,  
              unsigned int flags,  
              struct sockaddr* from, int* fromlen)
```

» to: packet destination address

» from: sender address present in received packet

» these primitives are similar to *send()/recv()* but additionally allow the sending of messages in connectionless (UDP) scenarios, without any connection establishment

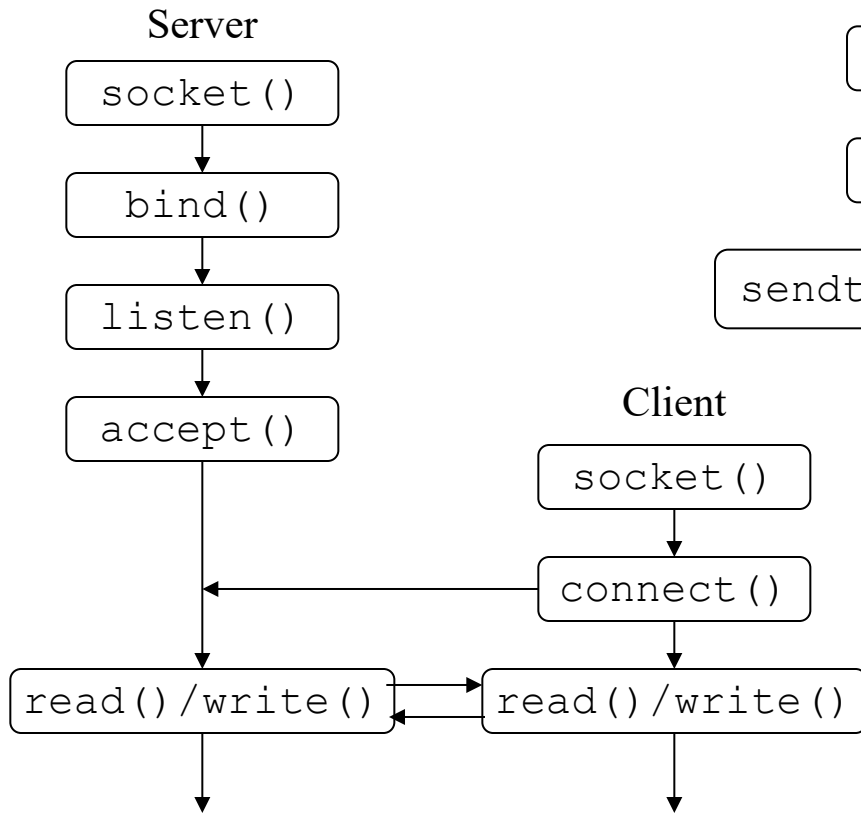
```
➔ int close(int sockfd)
```

» this primitive is used to close the socket

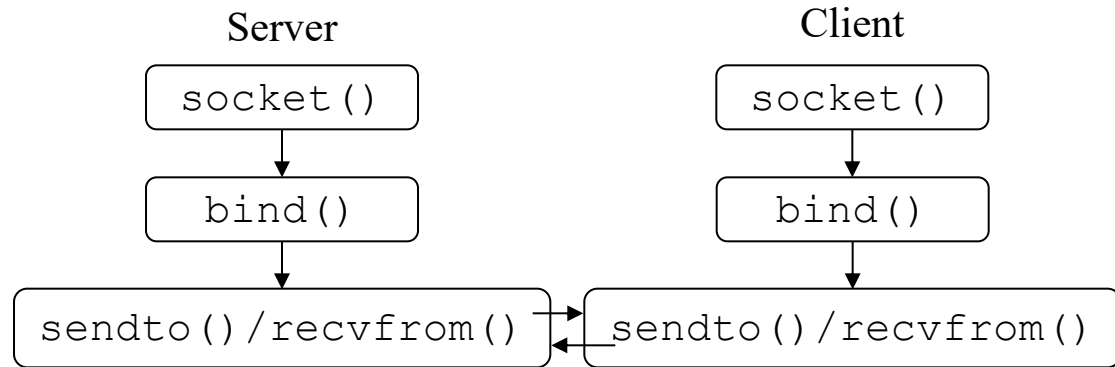
# Berkeley Sockets

---

connection-oriented protocol



Non-connection-oriented protocol



Note: the client of a TCP connection can call the `bind()` primitive before establishing the connection



# Berkeley Sockets

---

- ♦ Ordering of octets
  - » varies with architecture (e.g., Intel is *little endian*, Motorola is *big endian*)
    - Little endian → little end first;      Big endian → big end first
  - » network byte order → *big endian*
  - » conversion primitives (long - 32 bits, short - 16 bits):
    - ➔ `u_long htonl(u_long hostlong)`
    - ➔ `u_short htons(u_short hostshort)`
    - ➔ `u_long ntohl(u_long netlong)`
    - ➔ `u_short ntohs(u_short netshort)`
- ♦ Conversion between address formats
  - » *dotted decimal notation* for 32-bit Internet address with network ordering
    - ➔ `unsigned long inet_addr(char * cp)`
  - » 32-bit Internet address with network ordering for *dotted decimal notation*
    - ➔ `char* inet_ntoa(struct in_addr in)`

# Berkeley Sockets

---

- ♦ *sockets* options

  - `setsockopt()`

  - `getsockopt()`

  - `fcntl()`

  - `ioctl()`

- ♦ Asynchronous  
input/output

  - » use of signs

- ♦ Input/Output  
Multiplexing

  - » `select()` routine

- ♦ *Domain Name Service*

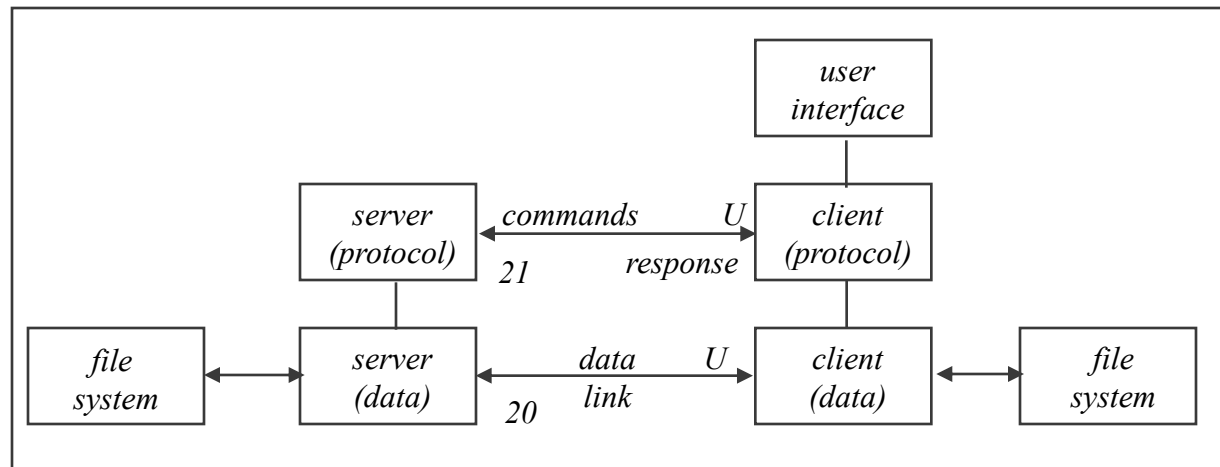
  - » allows obtaining the address of a host  
from its name

```
struct hostent*  
gethostbyname (const char* name);  
  
struct hostent{  
    char*  hname;           /*nome oficial*/  
    char** haliases;  
    int     h_addrtype; /*AF_INET*/  
    int     h_length;  
    char**  h_addr_list;  
};  
  
#define h_addr h_addr_list[0]
```

# *FTP - File Transfer Protocol*

---

- FTP - File Transfer Protocol
  - file transfer between computers (ASCII and binary)
- Client-Server Communication Model
  - independent TCP connections for connection control and data transfer
  - RFC 959



# *Example: Passive Mode FTP Session*

---

First “response” came split into two packets. { 220-Welcome to the University of Porto's mirror archive (mirrors.up.pt)  
220-----  
A response ends with a line containing the code, a space, optional text, and CRLF { 220-  
220-All connections and transfers are logged. The max number of connections is 200.  
220-  
220  
USER anonymous  
331 Please specify the password.  
PASS anonymous@  
230 Login successful.  
CWD debian  
250 Directory successfully changed.  
TYPE I  
200 Switching to Binary mode.  
SIZE README  
213 1328 ← File size in bytes  
PASV  
227 Entering Passive Mode (193,137,29,15,198,138). ← Server IP addr. Server IP port for file transfer =  $256 \times 198 + 138 = 50826$   
Client makes connection to 193.137.29.15:50826 { RETR README  
Server transfers file and closes data connection { 150 Opening BINARY mode data connection for README (1328 bytes). ← Preliminary reply (code 1xy)  
226 Transfer complete. ← Final reply (1<sup>st</sup> digit of code > 1)  
QUIT  
Server and client close control connection { 221 Goodbye.

NOTE: In a C string, CR is '\r' and LF is '\n'. For example, the QUIT command would be sent as "QUIT\r\n". Telnet does this by default.

---

## *Network Configuration Examples*

# *Network Configurations on Linux (examples)*

---

- ♦ Restart the networking service
  - » `systemctl restart networking`
- ♦ Configuration of tux
  - » Activate interface eth0
    - `ifconfig eth0 up`
  - » List current network interfaces configurations
    - `ifconfig`
  - » Configure eth0 with IP Address 192.168.0.1 and mask of 16 bits
    - `ifconfig eth0 192.168.0.1/16`
  - » Add a route to a subnetwork
    - `route add -net 192.168.1.0/24 gw 172.16.4.254`
  - » Add default route
    - `route add default gw 192.168.1.1`
  - » List current routes
    - `route -n`

# *Network Configurations on Linux (examples)*

---

- » Enable IP forwarding
  - `sysctl net.ipv4.ip_forward=1`
- » Disable ICMP echo ignore broadcast
  - `sysctl net.ipv4.icmp_echo_ignore_broadcasts=0`
- ♦ NAT configuration on Linux
  - » `iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE`  
`iptables -A FORWARD -i eth1 -m state --state NEW,INVALID -j DROP`  
`iptables -L`  
`iptables -t nat -L`
- ♦ Configuration of DNS
  - » Edit `/etc/resolv.conf`, clean the file and add the IP address of the DNS server  
`nameserver xxx.xxx.xxx.xxx`

# *Reset the Mikrotik switch*

---

- ◆ Connect to the switch
  - » Serial port **/dev/ttyS0** on one of the tux using o **GTKterm**
    - Connect S0 port to RS232 on the patch panel and from RS232 to the console port of the switch
    - Set the Baudrate to 115200 bps
    - Username - admin
    - Password - (blank)
  
- ◆ After login in with GTKterm
  - » `/system reset-configuration`
    - y
    - [ENTER]



# *Reset the Mikrotik router*

---

- ◆ Connect to the router
  - » Serial port **/dev/ttyS0** on one of the tux using o **GTKterm**
    - Connect S0 port to RS232 on the patch panel and from RS232 to the Router MTIK
    - Set the Baudrate to 115200 bps
    - Username - admin
    - Password - (blank)
  
- ◆ After login in with GTKterm
  - » `/system reset-configuration`
    - y
    - [ENTER]

# *Handling Bridges in Mikrotik Switch*

---

- ◆ Creating a bridge (bridgeY0)
  - » `/interface bridge add name=bridgeY0`
  - » `/interface bridge print`
- ◆ Remove a bridge (bridgeY0)
  - » `/interface bridge remove bridgeY0`
  - » `/interface bridge print`
- ◆ Add port1 to bridgeY0
  - » `/interface bridge port add bridge=bridgeY0 interface=ether1`
  - » `/interface bridge port print`
- ◆ Remove port1 from bridge
  - » `/interface bridge port remove [find interface =ether1]`
  - » `/interface bridge port print`
- ◆ Show bridges and ports
  - » `/interface bridge port print brief`

# *Configuring the Mikrotik router*

---

- ◆ Network Interfaces
  - » `/ip address add address=172.16.1.Y9/24 interface=ether1`
  - » `/ip address print`
- ◆ Routes
  - » Default gateway
    - `/ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254`
  - » Static Routes
    - `/ip route add dst-address=172.16.10.0/24 gateway=172.16.11.253`
  - » Show routes
    - `/ip route print`
- ◆ NAT
  - » Disable default nat
    - `/ip firewall nat disable 0`
  - » Add NAT rules
    - `/ip firewall nat add chain=srcnat action=masquerade out-interface=ether1`

# Connections in Bancada 1

