FSO 2013: class of 21st of October

Communication between processes

- **Message passing. Client server-interaction**
- **Interprocess communication using *sockets***
- ***Mysocks* socket library**

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

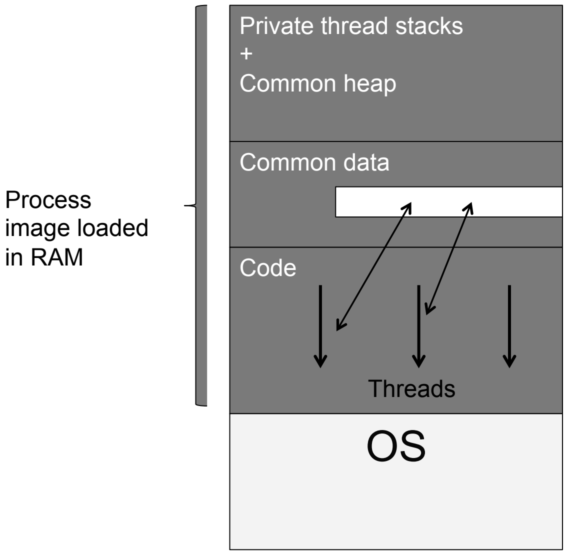
Cooperating Processes must

- **Exchange information:** processes transfer information between them. An example is the producer consumer problem
- **Synchronize actions:** action B in process P2 must happen only after action A in process P1. Example: mutual exclusion

Interprocess Communication (IPC)

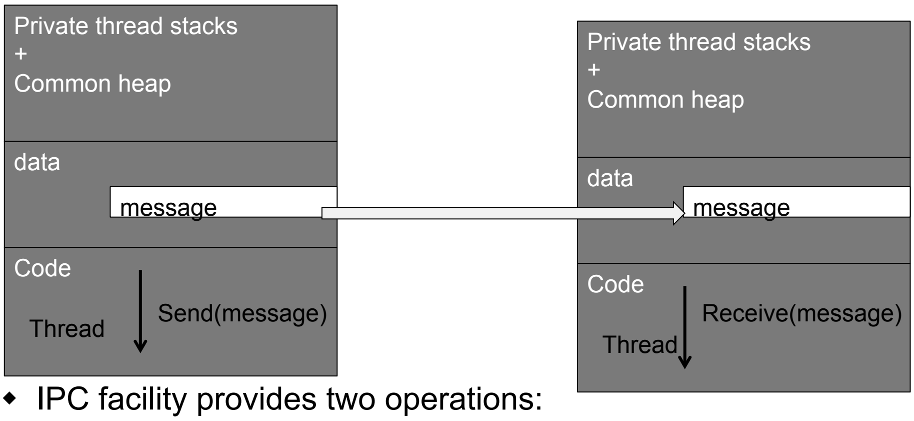
- Mechanism for processes to communicate and to synchronize their actions
- Information exchange between processes
  - Using shared memory Example: Pthreads reading and writing shared variables
  - Without shared memory
    - Example 1: a UNIX process created by a fork() does not share memory with other processes in the system
    - Example 2: a process running in machine H1 and another process running in machine H2

IPC with shared memory



- Threads communicating reading and writing shared variables
- No OS help is needed (fast)
- Reading and writing common variables need care (dangerous)

IPC without shared memory

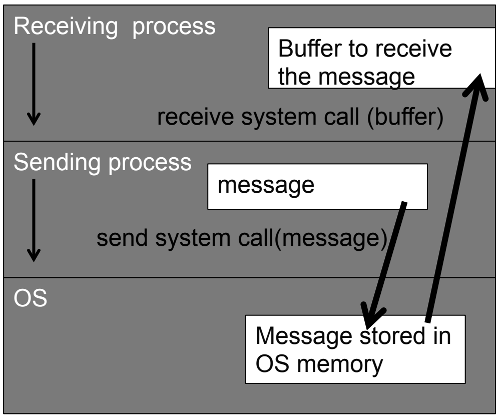


- IPC facility provides two operations:
  - **send** *message*
    space of the sending process
  - **receive**(*message*) – the bytes received are copied to the address space of the process that performs the operation

How to specify the receiver

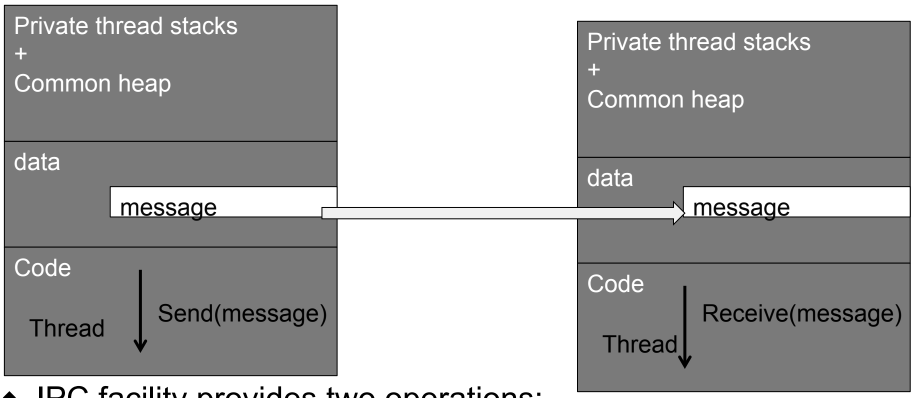
- Direct communication – the receiver of the message is specified by its process id
  - Processes must name each other explicitly:
    - **send** (*P, message*) – send a message to process P
    - **receive**(*Q, message*) – receive a message from process Q
- Indirect communication – Messages are directed and received from external entities (mailboxes or ports)
  - Each mailbox / port has a unique id. Operations are:
    - **send**(*A, message*) – send a message to mailbox /port A
    - **receive**(*A, message*) – receive a message from mailbox / port A

IPC without shared memory needs OS support


Receiving process
Buffer to receive the message
receive system call (buffer)
Sending process
message
send system call(message)
OS
Message stored in OS memory

- In the same machine, the OS must make the byte transfer as a process cannot access another process memory

IPC without shared memory



Private thread stacks
+
Common heap

data

message

Code

Thread

Send(message)

Private thread stacks
+
Common heap

data

message

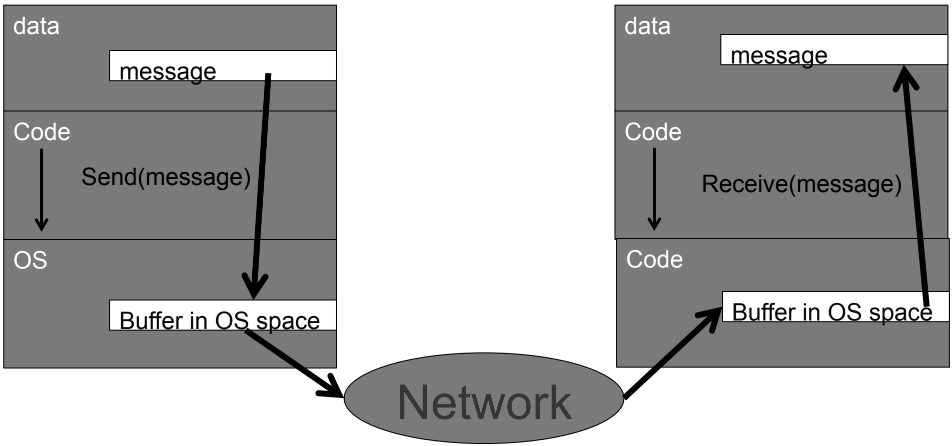
Code

Receive(message)

Thread

- IPC facility provides two operations:
  - **send**(*message*) – a sequence of bytes leaves the address space of the sending process
  - **receive**(*message*) – the bytes received are copied to the address space of the process that performs the operation

Extending IPC to two processes in distinct machines



data

message

Code

Send(message)

OS

Buffer in OS space

Network

data

message

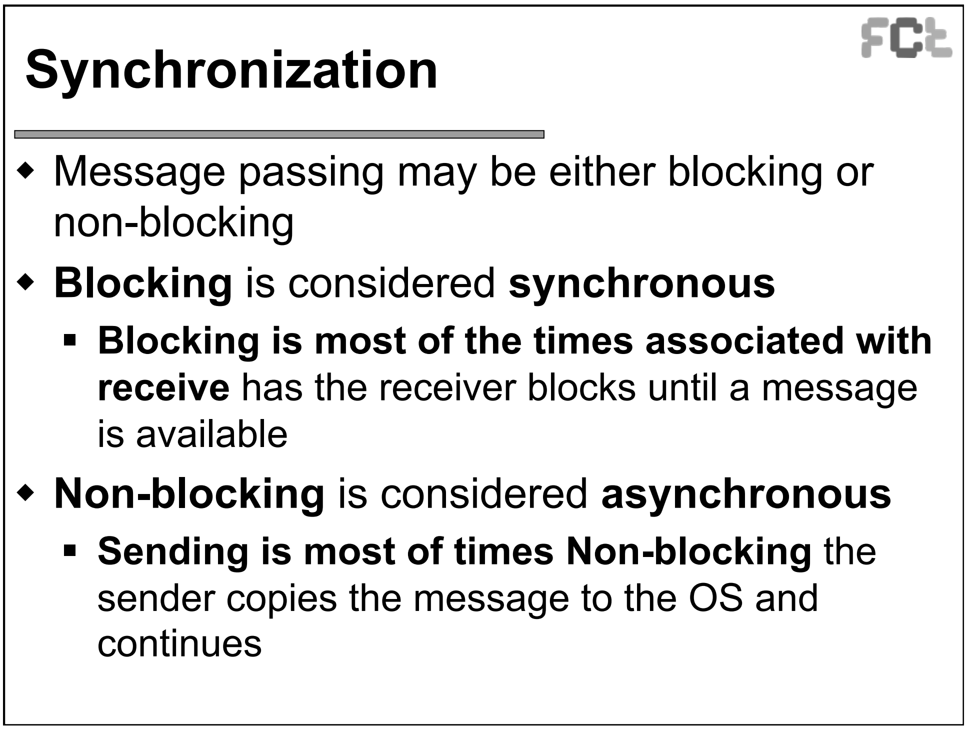
Code

Receive(message)

Code

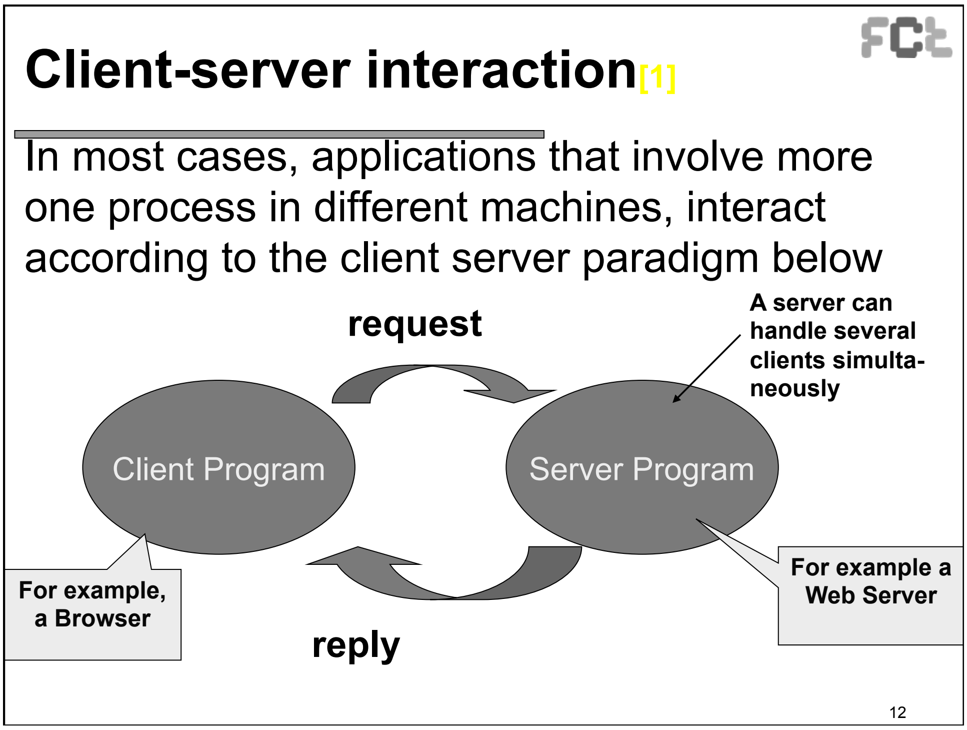
Buffer in OS space

- OS can deliver bytes to the network and receive bytes from the network
- Sender must specify: the address of the machine where to deliver the bytes, and a port / mailbox in the remote machine



Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking is most of the times associated with receive** has the receiver blocks until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Sending is most of times Non-blocking** the sender copies the message to the OS and continues

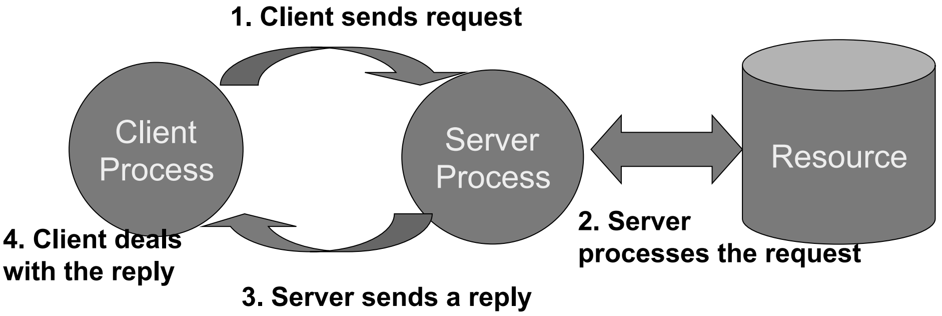


Client-server interaction[1]

In most cases, applications that involve more one process in different machines, interact according to the client server paradigm below

12

Client-Server interaction [2]



1. Client sends request

Client Process

Server Process

Resource

2. Server processes the request

3. Server sends a reply

4. Client deals with the reply

The **server** manages a resource and supplies a **service** to the clients that can demand operations over the resource

- Web Server: manages a set of files contain
  - (1) data: the service offered is the sending of the file contents
  - (2) programs: that the server executes on the behalf of a client
- FTP Servers and emails work the same way

Client-server interaction[3]

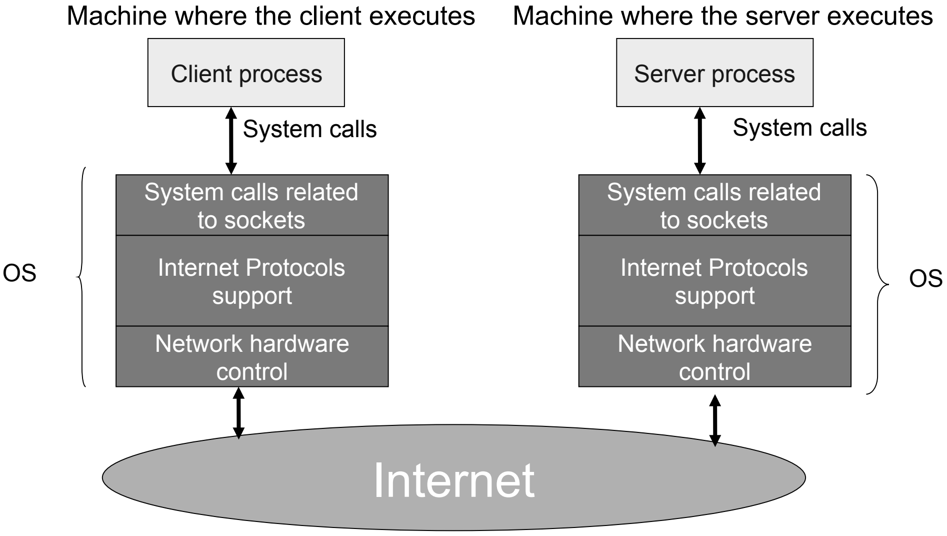
Clients and servers are processes and not computers: if the OS supports multi-programming

- In the same machine several servers and clients can be executing
- Client and server can execute in distinct machines or in the same

When building applications that include processes in separate machines, usually Internet Protocols are used: the layer in the Internet Protocols that support communication between processes residing in distinct machines is the Transport Layer

Operating systems have system calls that allow a process to use the transport layer of the Internet: the devices that allow access to these protocols are sockets.

Client-server interaction using Internet Protocols



Machine where the client executes

Machine where the server executes

Client process

Server process

System calls

System calls

OS

OS

System calls related to sockets

Internet Protocols support

Network hardware control

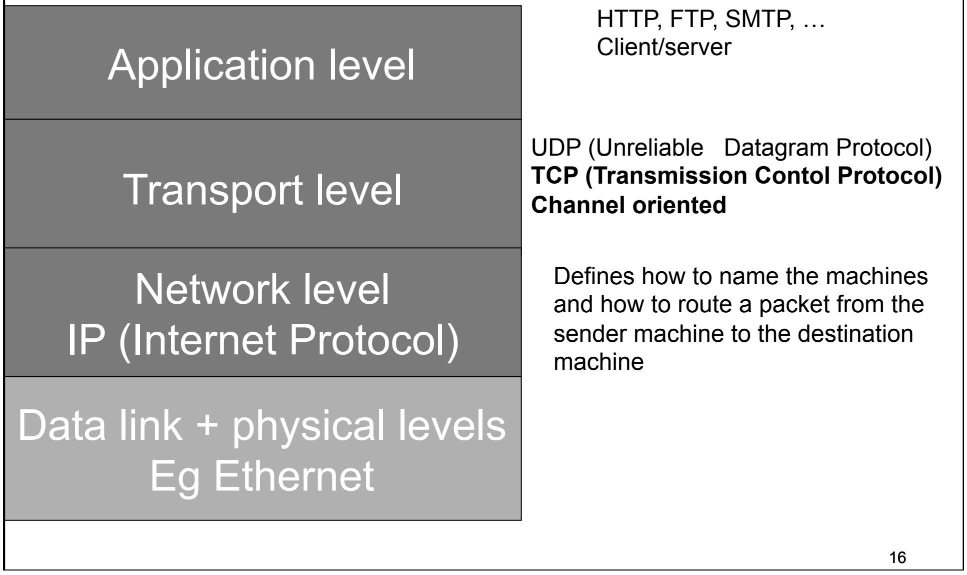
System calls related to sockets

Internet Protocols support

Network hardware control

Internet

TCP/IP protocol stack [1]



Application level

HTTP, FTP, SMTP, …
Client/server

Transport level

UDP (Unreliable Datagram Protocol)
TCP (Transmission Contol Protocol)
Channel oriented

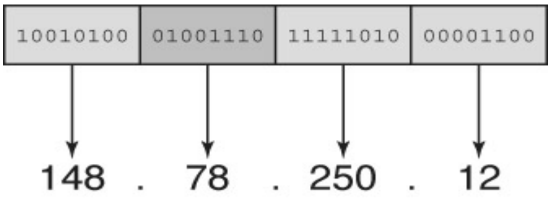
Network level
IP (Internet Protocol)

Defines how to name the machines and how to route a packet from the sender machine to the destination machine

Data link + physical levels
Eg Ethernet

Machine (node) address

- Unique in the Internet. Number with 32 bits;



IP address stored in 4 bytes

- there is also a human-friendly name that is a string. Example asc.di.fct.unl.pt

TCP and UDP ports

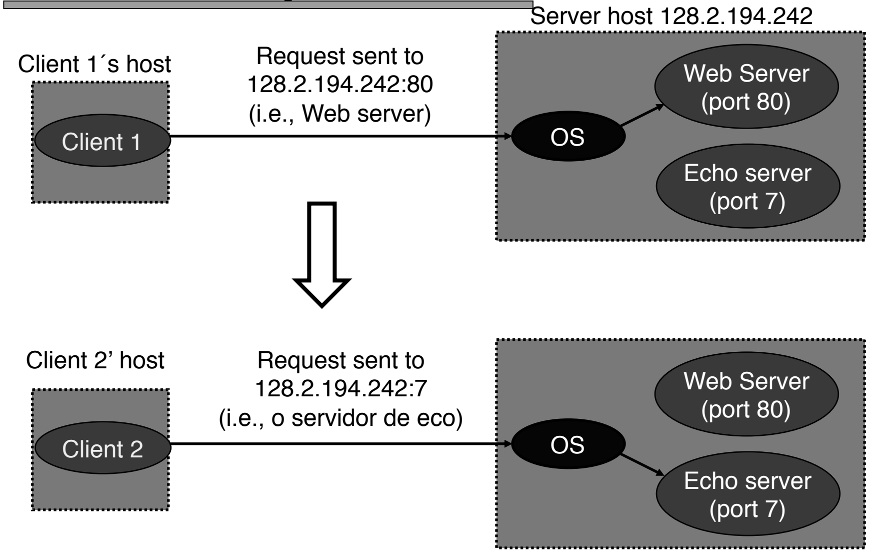
- To identify the addressee of a message one needs more than the IP address
- The Internet transport protocols support *Port Addresses or port numbers*
  - The port number is used to identify the service that is offered by a server process
  - If the IP address identifies a house (machine or host) the port identifies a floor where stays someone that offers a service
  - Example: port 80 is the default port for web servers

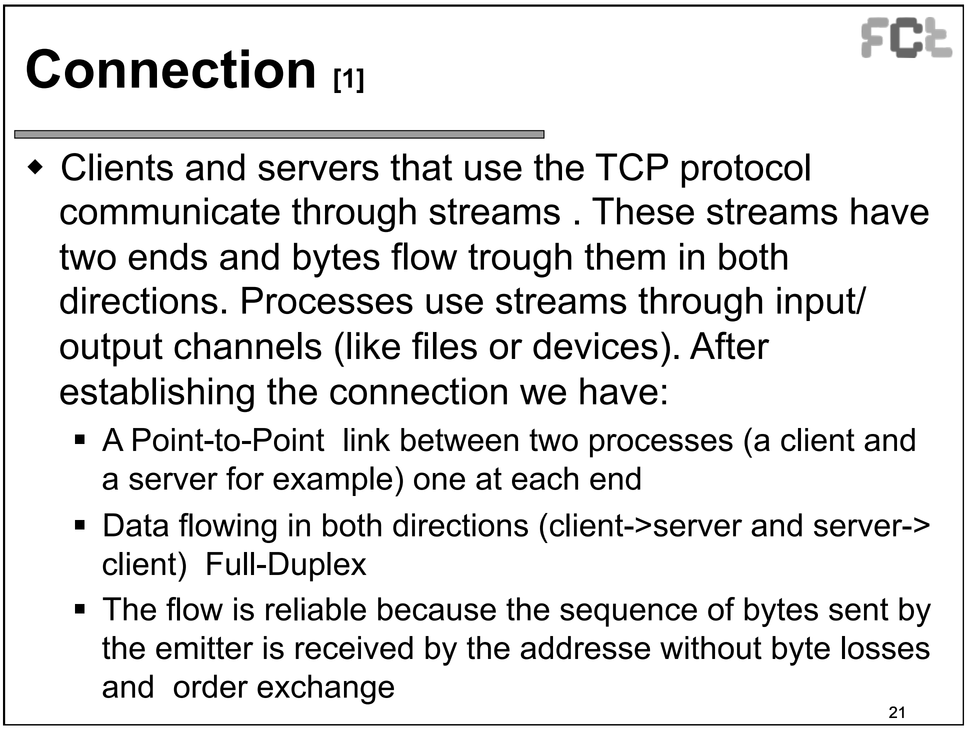
Standard ports and user ports

Application protocol	Port	Use
ftp	20	File transfer
ssh	22	Secure remote login
smtp	25	Email sending and receiveng
http	80	Web
pop3	110	Alternative email protocol

User ports: Ports between 1024 and 65535 can be freely utilized by user programs

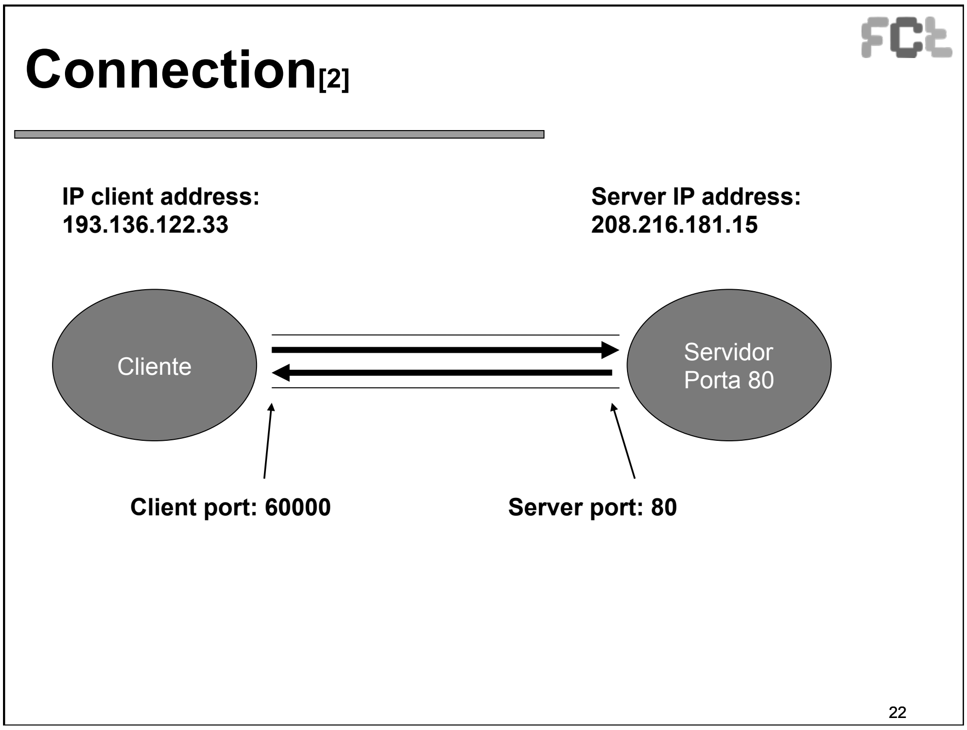
Use of Ports to identify the service required





Connection [1]

- Clients and servers that use the TCP protocol communicate through streams . These streams have two ends and bytes flow trough them in both directions. Processes use streams through input/ output channels (like files or devices). After establishing the connection we have:
  - A Point-to-Point link between two processes (a client and a server for example) one at each end
  - Data flowing in both directions (client->server and server-> client) Full-Duplex
  - The flow is reliable because the sequence of bytes sent by the emitter is received by the addresse without byte losses and order exchange



Connection[2]

IP client address:
193.136.122.33

Server IP address:
208.216.181.15

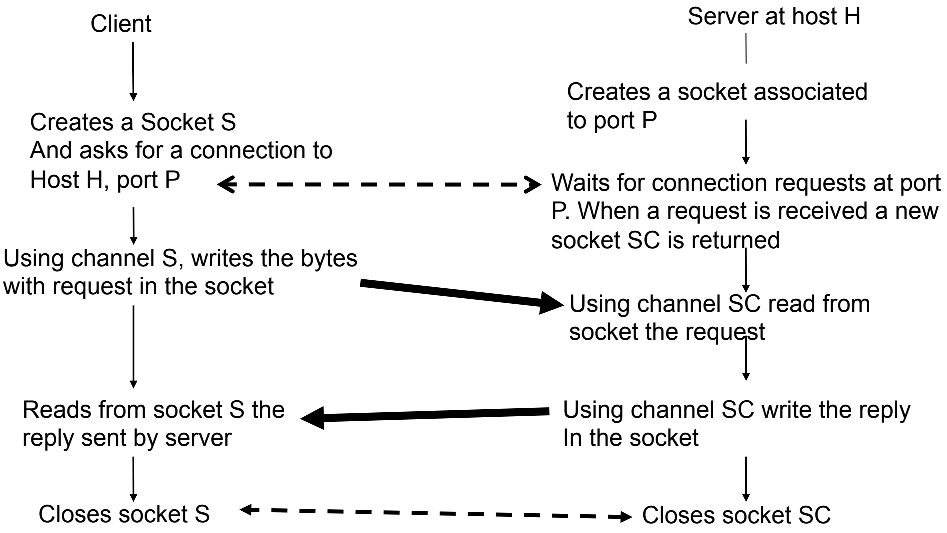
Client port: 60000

Server port: 80

TCP Sockets

- It´s a virtual device allowing the access to one of the endpoints of a TCP connection
- There is a set of system call that allow a process to build and destroy connections, and send and receive bytes through a connection
- Introduced in BSD UNIX (1982), all the modern operating systems support TCP *sockets*

Client /Server with Sockets



Client

Creates a Socket S
And asks for a connection to
Host H, port P

Using channel S, writes the bytes
with request in the socket

Reads from socket S the
reply sent by server

Closes socket S

Server at host H

Creates a socket associated
to port P

Waits for connection requests at port
P. When a request is received a new
socket SC is returned

Using channel SC read from
socket the request

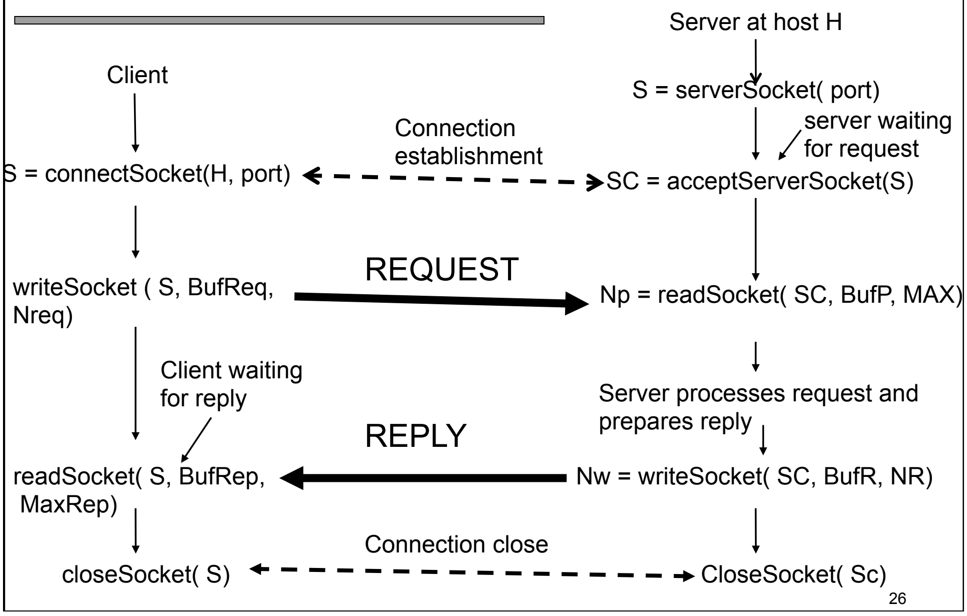
Using channel SC write the reply
In the socket

Closes socket SC

mysocks Socket Library

Operation	Input parameters	Return value
serverSocket (only server)	Server Port	> 0 I/O channel associated to the socket; -1 in case of error
acceptServerSocket (only server)	Channel returned by the ServerSocket operation se	> 0 I/O channel that allows reading and writing bytes from/to client
connectSocket (only client)	String with server´s host name, integer with server´s port	> 0 I/O channel that allows reading and writing from/to server
writeSocket (client and server)	I/O channel, address of bytes to write, number of bytes to write	> 0, number of bytes sent < 0 erro
readSocket (client and server)	I/O channel, address of buffer to receive bytes, number of bytes to write	> 0 number of received bytes < 0, error == 0, peer closed write channel
closeSocket (client)	I/O channel associated to socket	0 OK; -1 error

Client/Server with mySocks



Server at host H

Client

S = serverSocket(port)

Connection establishment

server waiting for request

S = connectSocket(H, port)

SC = acceptServerSocket(S)

REQUEST

writeSocket (S, BufReq, Nreq)

Np = readSocket(SC, BufP, MAX)

Client waiting for reply

Server processes request and prepares reply

REPLY

readSocket(S, BufRep, MaxRep)

Nw = writeSocket(SC, BufR, NR)

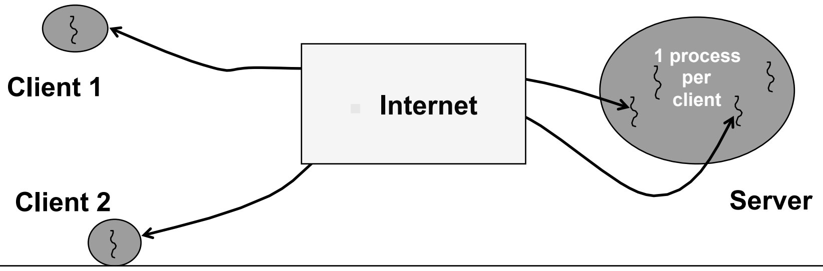
Connection close

closeSocket(S)

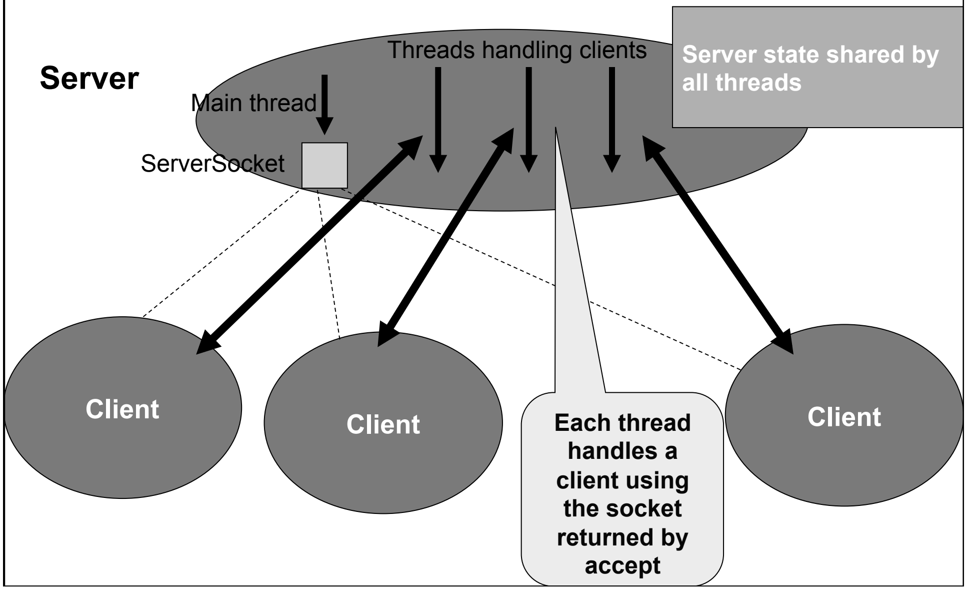
CloseSocket(Sc)

Server that several requests at the same time

- Servers need to handle new connections while replying to the current request
  - Servers that handle simultaneously more than one request are concurrent servers.
- When a new connection is made, the server acknowledges it and launches a new process to reply to the new client.



Server handling several clients simultaneously


Server
Threads handling clients
Server state shared by all threads
Main thread
ServerSocket
Client
Client
Each thread handles a client using the socket returned by accept
Client