



# Introducción a sockets node.js

Juan Quemada, DIT - UPM

# sockets en node.js - Índice

1. <u>Primeras aplicaciones TCP/IP, modelo cliente-servidor e interfaz de sockets</u> .....	<u>3</u>
2. <u>Sockets TCP de Cliente</u> .....	<u>10</u>
3. <u>Sockets TCP de servidor</u> .....	<u>16</u>
4. <u>Sockets UDP</u> .....	<u>23</u>



# Primeras aplicaciones TCP/IP, modelo cliente-servidor e interfaz de sockets

Juan Quemada, DIT - UPM

# Internet y la arquitectura TCP/IP

## ◆ Internet se crea el 1 de Enero 1983

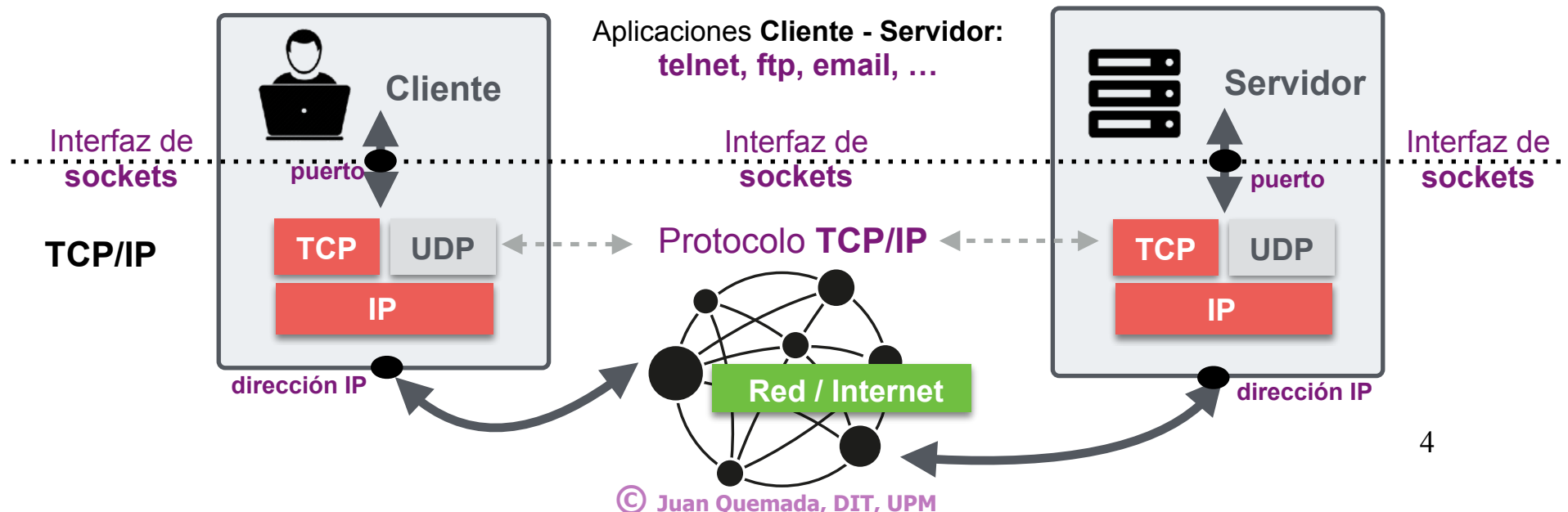
- Interconectando con la pila de **protocolos TCP/IP** redes relacionadas con ARPANET

## ◆ Interfaz de sockets

- Permite programar **aplicaciones cliente-servidor** conectadas con TCP/IP
  - ◆ La **dirección IP** y el **puerto** identifican el cliente o servidor con el que comunicar

## ◆ Las primeras **aplicaciones** cliente-servidor de Internet son

- telnet (terminal virtual), ftp (transferencia de ficheros), email (correo elec.), ..
  - ◆ La **Web**, la aplicación mas usada hoy en Internet, no aparece hasta principios de los noventa



# Aplicaciones cliente-servidor

- ◆ **Clientes:** dan acceso a la información y los servicios
  - Son **anónimos** y realizan solicitudes a los servidores
- ◆ **Servidores:** alojan la información y los servicios
  - Son **públicos** y tienen una **dirección conocida** en Internet
- ◆ **TCP/IP:**
  - Pila de protocolos que permite crear Internet
    - ◆ **IP** (Internet Protocol): protocolo de interconexión de redes heterogeneas
      - Es el que da el nombre a Internet
    - ◆ **TCP-UDP:** protocolos de transporte de información
      - Las aplicaciones utilizan el interfaz de **sockets** para utilizar **TCP-UDP**
  - ◆ Ver: [https://en.wikipedia.org/wiki/Internet\\_protocol\\_suite](https://en.wikipedia.org/wiki/Internet_protocol_suite)

# Máquina servidora



- ◆ La máquina servidora es el ordenador que alberga servidores (programas)
  - Una máquina servidora puede ser física (real)
  - Una máquina también puede ser virtual (simulada por software y alojada en la nube)
  
- ◆ Cada máquina tiene una **dirección** (o varias) de tipo **IP**
  - **IPv4**: versión 4 del protocolo IP (anterior) con dirección de **32 bits**
    - ◆ Por ejemplo: **192.9.0.144**, **127.0.0.1** (localhost - mi máquina), ...
  - **IPv6**: versión 6 del protocolo IP (última) con dirección de **128 bits**
    - ◆ Por ejemplo: **2001:db8:85a3::8a2e:370:7334**, ....
  
- ◆ Cada dirección IP puede asociarse a una dirección simbólica o de dominio
  - El servicio **DNS** gestiona estas direcciones, por ejemplo **upm.es**, **google.com**, ...
    - ◆ Las direcciones **simbólicas** (o de dominio) se suelen utilizar porque son mas fáciles de recordar

# Servidores y puertos

## ◆ Un **Servidor** es un programa

- Atiende un servicio **en un puerto** de la máquina servidora
  - ◆ Ejemplos de servidores: **Web, Correo Electrónico, Telnet, FTP, .....**

## ◆ **Puerto**

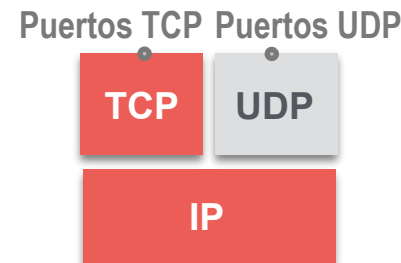
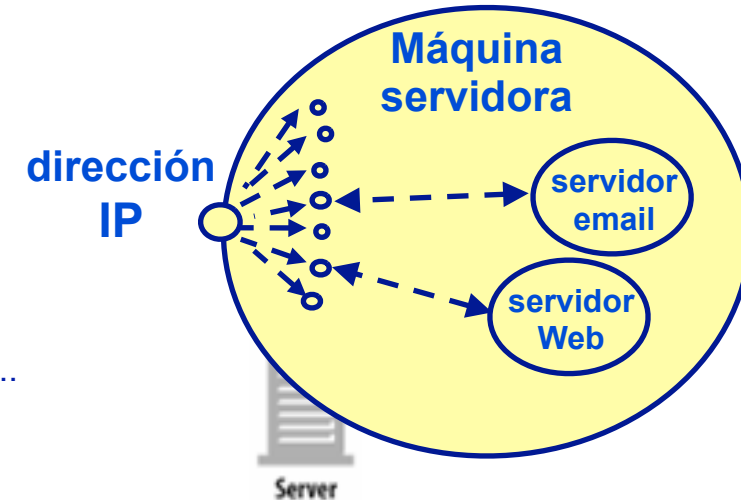
- **Sub-dirección de conexión a la red** de un programa, dentro de una **máquina**
  - ◆ Tiene 16 bits y permite albergar hasta  $2^{16}$  servidores (o servicios) diferentes
  - Los puertos 0 a 1023 están reservados para servicios del sistema y el resto para usuarios

## ◆ Una máquina servidora tiene 4 espacios de puertos diferentes de 16 bits

- Espacio de puertos **TCP IPv4**
- Espacio de puertos **TCP IPv6**
- Espacio de puertos **UDP IPv4**
- Espacio de puertos **UDP IPv6**

## ◆ **netstat**

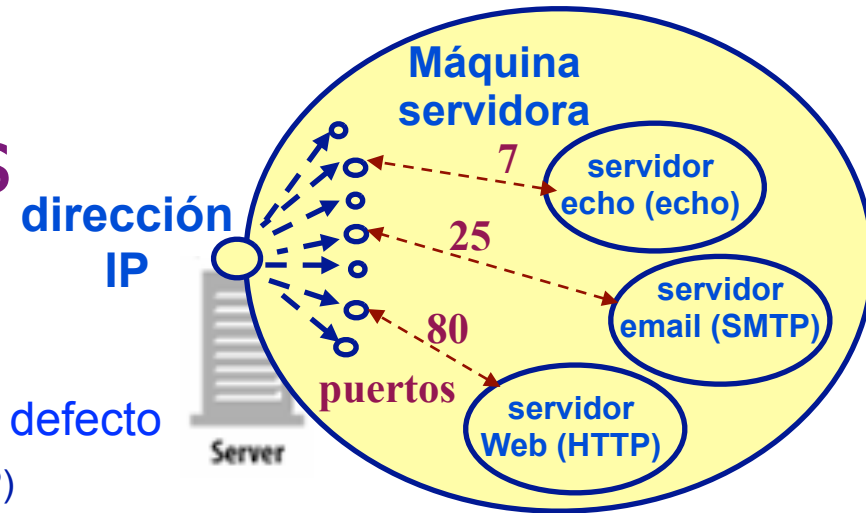
- Comando UNIX que muestra el estado de los puertos (sockets) de una máquina
  - ◆ **\$ netstat -p tcp** muestra el estado de los puertos TCP de la máquina
  - ◆ **\$ netstat -p udp** muestra el estado de los puertos UDP de la máquina



# Servicios, protocolos y URLs

## ◆ Servicios (o servidores)

- Cada RFC le asocia un protocolo y un puerto por defecto
  - ♦ **Web seguro:** protocolo HTTPS (puerto 443 de TCP)
  - ♦ **Web:** protocolo HTTP (puerto 80 de TCP)
  - ♦ **Email:** protocolo SMTP, .. (puertos 25 de TCP, ...)
  - ♦ **Telnet:** protocolo Telnet (puerto 23 de TCP)
  - ♦ **Daytime:** protocolo Daytime (puerto 13 de TCP o UDP)
  - ♦ **Echo:** protocolo Echo (puerto 7 de TCP o de UDP)



## ◆ Un **URL** identifica un servicio en una máquina servidora

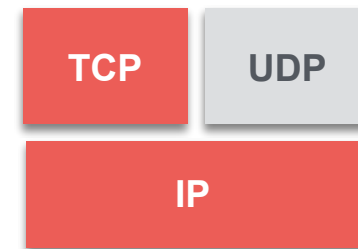
- **http://upm.es** indica un servidor Web, accesible con el protocolo HTTP, en el puerto por defecto (80 de TCP) en la máquina **upm.es**
  - ♦ El acceso a servicios en puertos no estándar debe incluirlo en el URL: **http://upm.es:8080**

## ◆ IANA e IETF ordenan la asignación de puertos a servicios con RFCs

- <https://www.iana.org>, <http://www.ietf.org>



# Interfaz de Sockets

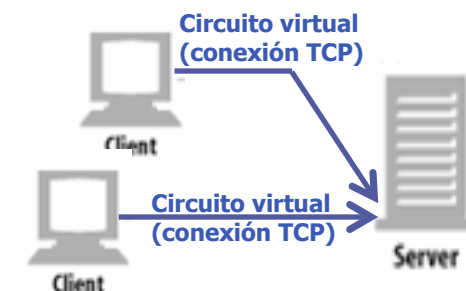


## ◆ El Interfaz de sockets

- API que permite comunicar clientes con servidores utilizando la pila TCP/IP
  - ◆ Cliente y servidor son roles que se programan en una aplicación
    - Cuando un lado tiene ambos roles la aplicación es de tipo P2P - Peer To Peer

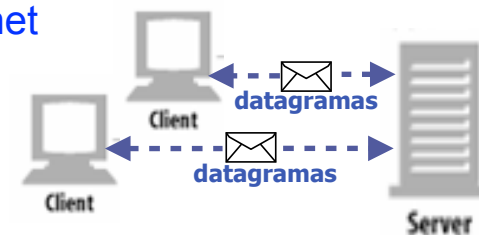
## ◆ Socket TCP

- Conecta clientes con servidores utilizando **circuitos virtuales**
  - ◆ Los **circuitos virtuales** TCP se denominan también **conexiones TCP**
- Un **circuito virtual** permite **intercambio fiable** de información entre ambos extremos
  - ◆ **Garantiza la entrega** de datos, pero a costa de **aumentar** el retardo



## ◆ Socket UDP

- Permite envío de datagramas entre aplicaciones a través de Internet
  - ◆ Los datagramas tienen menor retardo de transmisión, pero pueden perderse
- Tiene varios tipos de servicio
  - ◆ **Unicast**: Envío de datagramas con un solo destinatario
  - ◆ **Broadcast**: Difusión de datagramas a muchos destinatarios
  - ◆ **Multicast**: Envío de datagramas dentro de un grupo de destinatarios





# Sockets TCP de cliente

Juan Quemada, DIT - UPM

# Socket TCP de cliente

## ◆ Servicio TCP

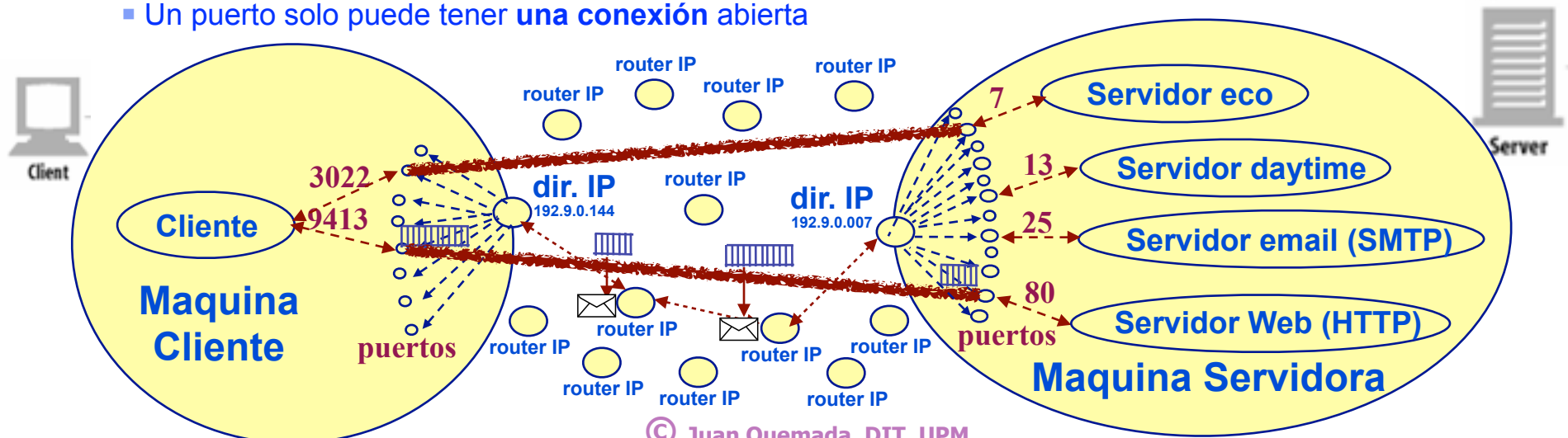
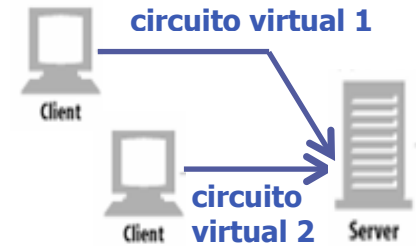
- Servicio orientado a conexión basado en circuitos virtuales

## ◆ Circuito virtual (o conexión TCP)

- Permite intercambiar información de forma **fiable** entre sus extremos
  - ♦ **Garantiza la entrega** de datos, pero a costa de **aumentar** el retardo porque tiene que retransmitir paquetes perdidos
- Se **identifica** por los puertos y direcciones IP que conecta (identificador único en Internet)
  - ♦ Los identificadores de las 2 conexiones de la figura son **<3022, 192.9.0.144, 7, 192.9.0.007>** y **<9413, 192.9.0.144, 80, 192.9.0.007>**
    - El identificador es **único**, solo puede existir en Internet una conexión con un identificador dado

## ◆ Socket de cliente

- Permite solicitar un circuito virtual a un servidor enviándole una **solicitud de conexión**
  - ♦ La solicitud de conexión se envía desde un puerto del cliente elegido al azar entre los no ocupados
    - Si el servidor acepta la conexión, se **establece el circuito**
      - Una vez establecido se pueden enviar o recibir datos escribiendo o leyendo en el socket
- Un puerto solo puede tener **una conexión** abierta



# Socket TCP de cliente



## ◆ Clase **net.Socket**

- Objetos de gestión de una **conexión TCP**, tanto en el **cliente**, como en el **servidor**
  - ◆ Permite crear una conexión, enviar o recibir datos, liberar la conexión, etc.
  - Es parte del paquete **net** de node.js: <https://nodejs.org/api/net.html>

## ◆ Elementos de Socket

- **socket.connect(<port>, [<host>], [<connectListener>])**
  - ◆ Solicita establecer una conexión desde puerto aleatorio a **<port>** en **<host>**, e instala **connectListener(socket)**
    - Cuando esta establecida, devuelve el **socket** como parámetro de retorno
- **socket.destroy()**
  - ◆ **Libera** la conexión
- **socket.setTimeout(<timeout>[, <callback>])**
  - ◆ Instala **<callback>** como manejador del evento **'timeout'** que ocurre transcurridos **<timeout>** milisegundos
- **socket.write(<data>, [<encoding>], [<callback>])**
  - ◆ Envía **<data>** interpretados según **<encoding>** e instala **<callback>** que se invocará cuando los datos salgan
- Eventos: **connect** (conexión establecida), **data** (llegan datos), **end** (conexión cerrada), **close**, ...

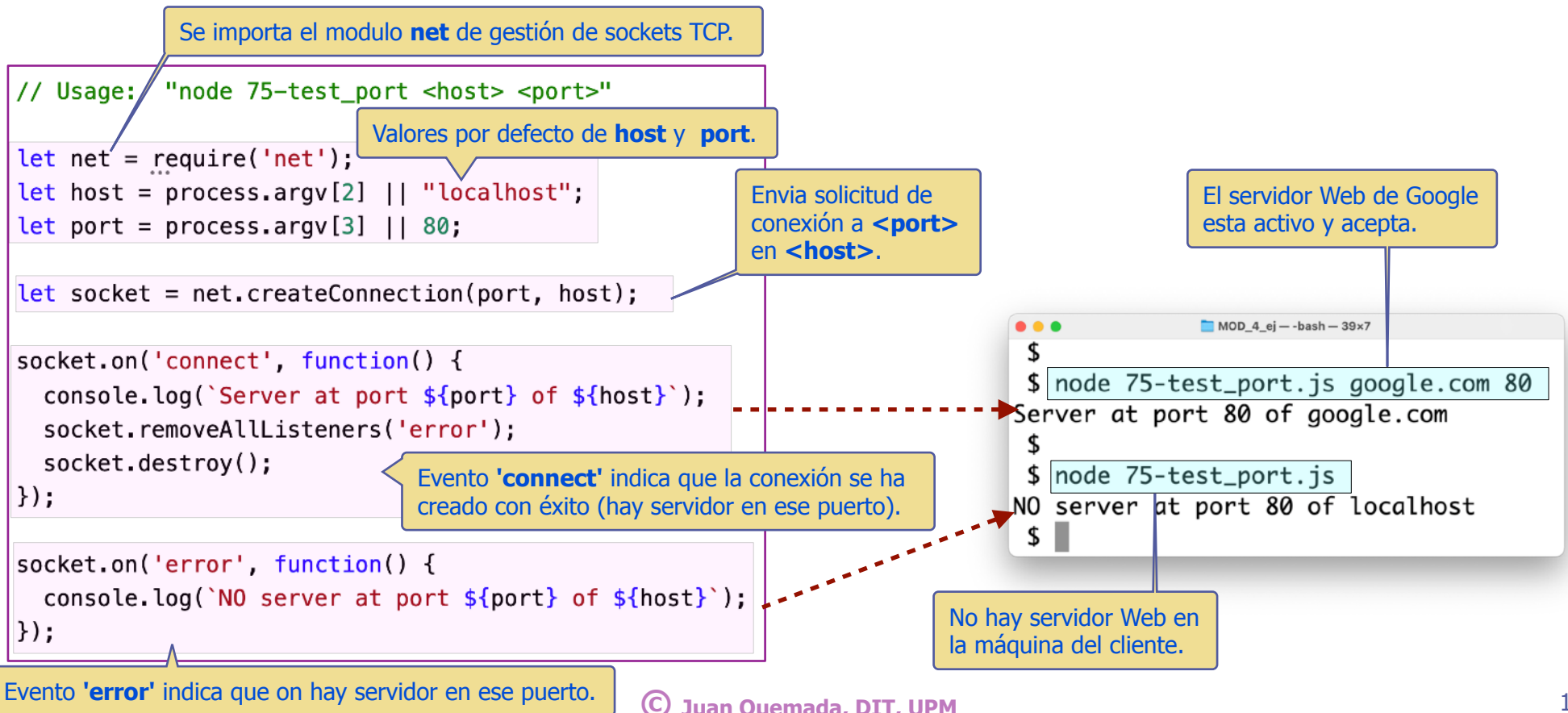
## ◆ Las conexiones se pueden establecer también con la factoría

- **net.createConnection(options[, connectListener])**

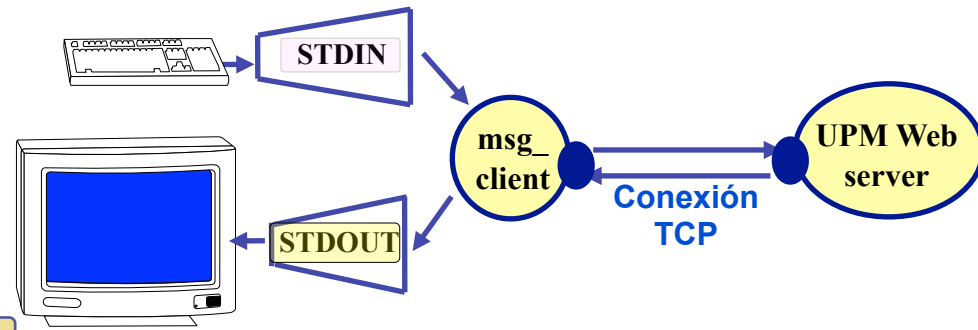
# test\_port TCP

## ◆ Programa **test\_port**

- Intenta establecer una conexión TCP en un puerto de una máquina
  - ◆ Si hay un servidor activo en dicho puerto la conexión se **acepta** y ocurrirá en evento '**connect**'
    - Los servidores aceptan las conexiones siempre que no estén desbordados
  - ◆ Si **NO** hay un servidor activo en dicho puerto la conexión se **rechaza** y ocurrirá en evento '**error**'



# msg\_client TCP



## Programa msg\_client

- Envía mensajes a un servidor
- ◆ Muestra la respuesta del servidor

```
// Usage: "node 77-msg_client <host> <port>"
```

```
let net = require('net');
let host = (process.argv[2] || "localhost");
let port = (process.argv[3] || 9000);
```

Envía solicitud de conexión a **<port>** en **<host>**.

```
let socket = net.createConnection(port, host);
```

```
process.stdin.on('data', function(data) {
  socket.write(data); // keyboard to server
});
```

Al teclear una línea ocurre el evento 'data', que la reenvía al servidor.

// server to screen

```
socket.on('data', function(data) {
  process.stdout.write(`\nResponse:\n ${data}`);
});
```

Al llegar un mensaje del servidor ocurre el evento 'data', que se reenvía a screen.

```
socket.on('error', function(e) {
  console.log(`No server at ${host} ${port}`);
  socket.destroy(); // No server exists
  process.exit();
});
```

Si ocurre cualquier error se destruye el socket, probablemente que no hay servidor en el puerto.

```
process.stdin.resume(); // Activate stdin
```

La entrada de teclado (stdin) debe activarse.

```
MOD_4_ej - node 77-msg_client.js upm.es
$
$ node 77-msg_client.js upm.es 80
GET /

Response:
HTTP/1.1 400 Bad Request
Date: Sun, 17 Jan 2021 14:50:53 GMT
Server: Apache
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1

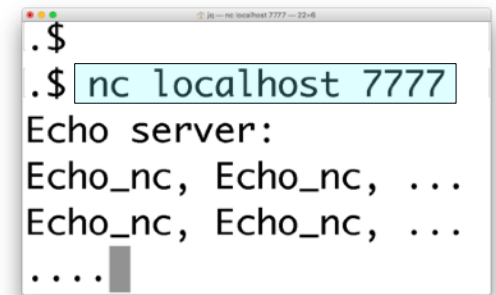
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```

En esta captura de pantalla se conecta con el servidor Web de UPM y se teclea una cabecera HTTP mal formada. El servidor acepta la conexión, como hacen siempre, pero al enviar la cabecera mal formada responde con un ERROR "400 Bad Request", como indica el protocolo HTTP que debe hacerse.

# Cientes nc y telnet

## ◆ nc o netcat

- Cliente muy versátil para realizar conexiones y escuchas TCP y UDP
  - ◆ Permite establecer conexiones TCP y dialogar intelectivamente con el servidor
- Por ejemplo: **"\$ nc localhost 7000"**
  - ◆ Establece una conexión TCP desde un puerto aleatorio al puerto 7777 de la misma máquina
    - Si hubiese un servidor en el puerto, este acepta la conexión y se puede dialogar con él.
- Ver: <https://en.wikipedia.org/wiki/Netcat>



```
.$  
.$ nc localhost 7777  
Echo server:  
Echo_nc, Echo_nc, ...  
Echo_nc, Echo_nc, ...  
....
```

## ◆ Vamos a utilizar **nc** como **cliente** para **probar servidores**

- También veremos después como se programa un cliente que establece conexiones TCP

## ◆ Telnet - cliente de terminal remoto

- Permite acceso remoto a un ordenador a través de la shell
  - ◆ Su uso no se recomienda por seguridad, aunque puede dialogar con servidores,
    - Es mucho menos versátil que **nc**
- Hoy se recomienda usar **ssh** (**secure shell**) para conexiones remotas seguras



# Sockets TCP de servidor

Juan Quemada, DIT - UPM



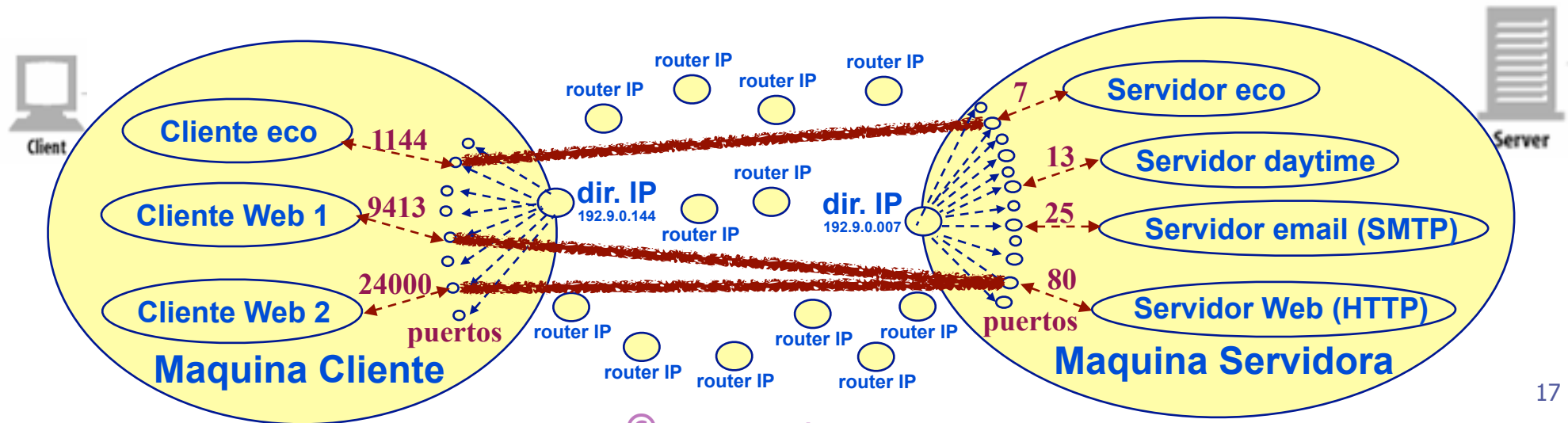
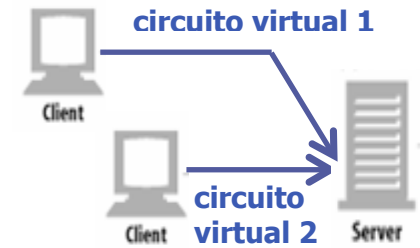
# Socket TCP de servidor

## ◆ Servidor

- Sirve **información** y **servicios** a los clientes que lo solicitan
- Es **público**
  - ◆ Tiene una **dirección** y **puerto conocidos** donde los clientes se conectan
- Es pasivo, solo espera a que los clientes se conecten

## ◆ Socket de servidor

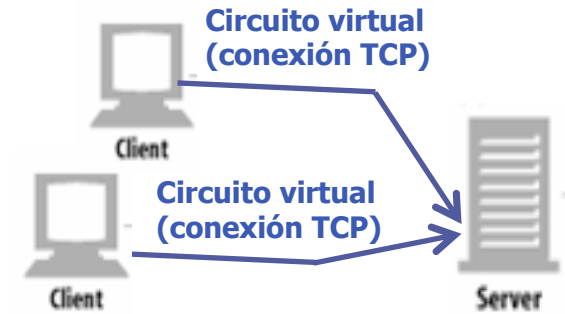
- Se conecta a un **puerto** al comenzar a operar
- Espera a recibir **solicitudes de conexión TCP** de clientes en ese puerto
  - ◆ Siempre acepta la conexión, salvo que este saturado
- Atiende a **muchos clientes** a través de conexiones TCP diferentes
- Si un puerto **no tiene un servidor conectado**, rechaza las solicitudes de conexión



# Sockets de servidor

## ◆ Clase **net.Server**

- Permite crear crear servidores TCP
  - ◆ Es parte del módulo **net** de node.js: <https://nodejs.org/api/net.html>



## ◆ **let servSocket = net.createServer(connectionListener)**

- Factoría para con la que se suelen crear servidores TCP
  - ◆ Cuando llega una solicitud de conexión se invoca **connectionListener(socket)**
    - **socket** da acceso a la conexión TCP con el cliente que acaba de conectarse
- Ver: [https://nodejs.org/api/net.html#net\\_net\\_createserver\\_options\\_connectionlistener](https://nodejs.org/api/net.html#net_net_createserver_options_connectionlistener)

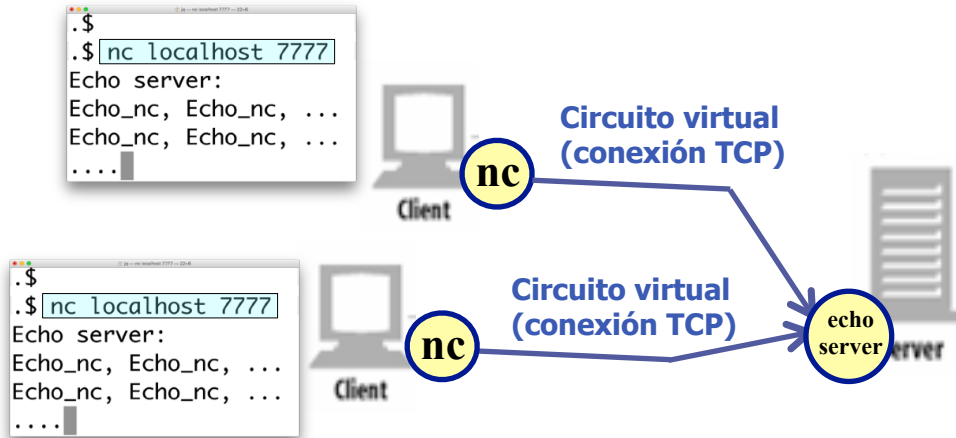
## ◆ **server.listen(port)**

- Método que conecta un servidor al puerto **port** donde recibirá solicitudes de conexión
  - ◆ Ver: [https://nodejs.org/api/net.html#net\\_server\\_listen](https://nodejs.org/api/net.html#net_server_listen)

## ◆ Posee otros métodos, propiedades y eventos

- **close(..), address(), ref(), ....., maxConnections, ....., 'connection', 'close', 'error', ...**

# Servidor Echo



## ◆ Protocolo de Echo

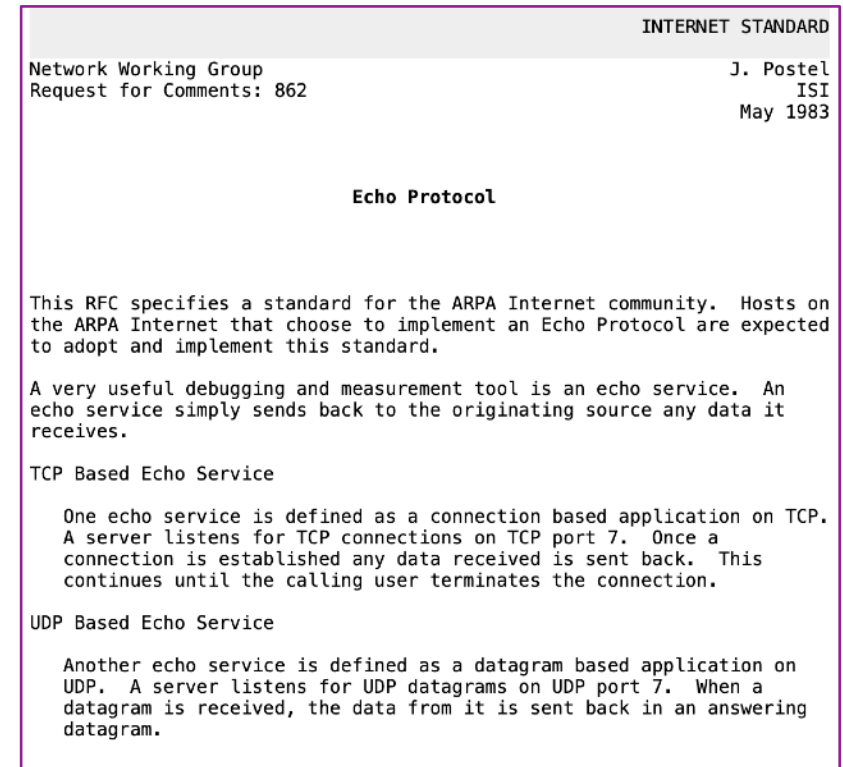
- Se describe en la RFC 862: <https://tools.ietf.org/html/rfc862>

## ◆ Servidor de eco

- Espera conexiones de clientes, que acepta cuando llegan
  - ◆ Una vez establecida la conexión con un cliente, devuelve el eco de cada línea que envía el cliente

## ◆ Es un protocolo muy sencillo de la primera Internet

- Su función era pedagógica, además de facilitar pruebas de conexión a la red



# Servidor Echo y clientes nc

## ◆ Funcionamiento del servidor

- Para cada solicitud de conexión de cliente se invoca el manejador: **(socket) => {...}**
  - ◆ Cada del manejador crea una closure que atiende al cliente que se ha conectado
    - La closure guarda el objeto socket y el manejador del evento 'data' siguen activos hasta que el cliente libera la conexión y el socket se destruye

Se importa el modulo **net** de gestión de sockets TCP.

```
let net = require('net');
```

Valor por defecto **port**.

```
let port = (process.argv[2] || 7);
```

Se crea de socket de servidor.

```
let server = net.createServer((socket) => {  
  socket.write('Welcome (Echo server)\n');  
  
  socket.on('data', function(data) {  
    socket.write(data); // Send back echo  
  });  
});
```

Si arranca el socket en el puerto indicado o por defecto.

```
server.listen(port);
```

```
console.log("Echo server at port: " + port);
```

Se arranca el servidor **echo** en el puerto 7000.

```
node 80-echo_serv 7000
```

Echo server at port: 7000

El manejador de conexiones TCP crea una closure que almacena el socket de acceso al cliente, con el manejador de 'data' hasta que el cliente cierra la conexión.

Se conecta el cliente 1 (**nc**) al puerto 7000 de la misma máquina (localhost).

```
nc localhost 7000
```

Welcome (Echo server)  
Soy el cliente 1  
Soy el cliente 1  
...

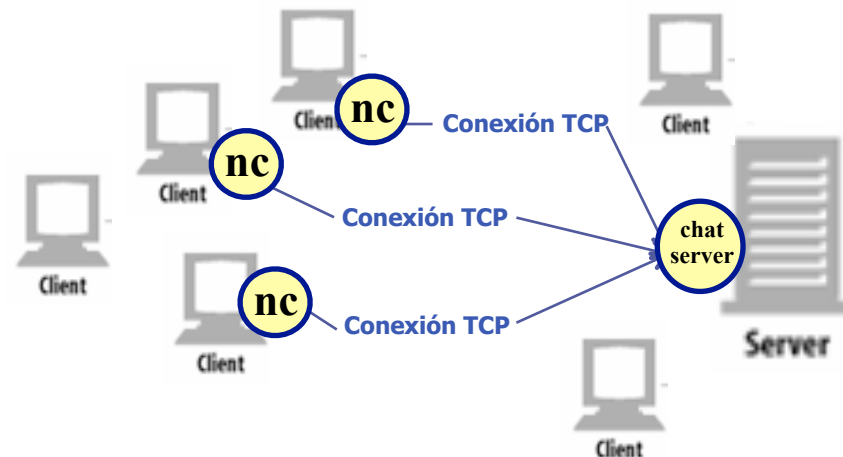
Se conecta el cliente 2 (**nc**) al puerto 7000 de la misma máquina (localhost).

```
nc localhost 7000
```

Welcome (Echo server)  
Soy el cliente 2  
Soy el cliente 2  
.....

The diagram illustrates the Echo server and its interaction with two clients. A central server icon is labeled 'Server'. Two client icons are labeled 'Client'. Arrows labeled 'Conexión TCP' point from each client to a central circle labeled 'echo server'.

# Servidor de chat



- ◆ El próximo ejemplo muestra un servidor de chat sencillo
  - Los clientes deben conectarse con **nc** al servidor
- ◆ Un cliente establece una conexión TCP para participar en el chat
  - A partir de ese momento
    - ◆ El cliente recibe todos los mensajes de todos los demás
    - ◆ Todos los demás reciben los mensajes que envíe ese cliente
- ◆ IRC - Internet Relay Chat (RFC 1459)
  - Existe una norma de servicio de chat para Internet
    - ◆ Pero es muy complicada para verla aquí (RFC 1495: <https://tools.ietf.org/html/rfc1459>)

# Servidor de chat

## ◆ Funcionamiento del servidor

- Cada solicitud de conexión de cliente invoca el manejador de solicitudes
  - ◆ (socket) => { ..... }

## ◆ La atención concurrente a varios clientes se realiza almacenando sus sockets en el array **clients**

- Los manejadores de 'end' y de 'data' siguen activos hasta que se destruye el socket en que se han instalado

Al solicitar conectar un cliente, el **manejador de solicitudes** del socket de servidor, ejecuta 3 instrucciones:

- 1) Añade el socket del nuevo cliente a **clients**
- 2) Instala manejador de 'end': elimina cliente de **clients**.
- 3) Instala manejador de 'data': reenvía msg.

Array de sockets de los clientes conectados. Es visible en todas las instancias de closures de cliente.

El socket, del cliente que se desconecta, se elimina de **clients**.

Reenvía cada msg que llega a los demás clientes.

Se arranca el servidor de **chat** en el puerto 9000.

```
MOD_4_ej - node 81-chat -- 26x5
$
$
$ node 81-chat
Chat server at port: 9000
```

```
let net = require('net');
let port = (process.argv[2] || 9000);
let clients = []; // Array of connected clients
```

```
let server = net.createServer( (socket) => {
```

```
  clients.push(socket); // add new client
```

```
  socket.on('end', function() { // remove client
    let i = clients.indexOf(socket);
    clients.splice(i, 1);
```

Se añade el nuevo socket al final del array **clients**.

```
  }); // send msg to other clients
```

```
  socket.on('data', function(msg) {
    for (let i=0; i < clients.length; i++) {
      if (clients[i] !== socket) {
        clients[i].write('-> ' + msg);
      }
    }
  });
});
```

```
server.listen(port);
console.log("Chat server at port: " + port);
```

22

```
jq - nc localhost 9000 -- 23x6
$
$ nc localhost 9000
Client 1 says hi!
-> Client 2 says hi!
-> Client 3 says hi!
```

nc

Client

Conexión TCP

nc

Conexión TCP

Conexión TCP

nc

Client

```
jq - nc localhost 9000 -- 24x5
$
$ nc localhost 9000
-> Client 1 says hi!
Client 2 says hi!
-> Client 3 says hi!
```

Client

```
jq - nc localhost 9000 -- 23...
$
$ nc localhost 9000
-> Client 1 says hi!
-> Client 2 says hi!
Client 3 says hi!
```



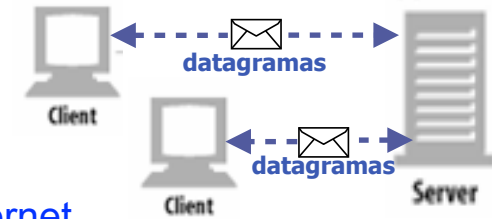
# Sockets UDP

Juan Quemada, DIT - UPM

# Servicio UDP

## ◆ Datagrama

- Mensaje, con **dirección incluida**, enviado a otra aplicación a través de Internet
  - ♦ No hay que establecer conexión previa con un destinatario para enviarle información

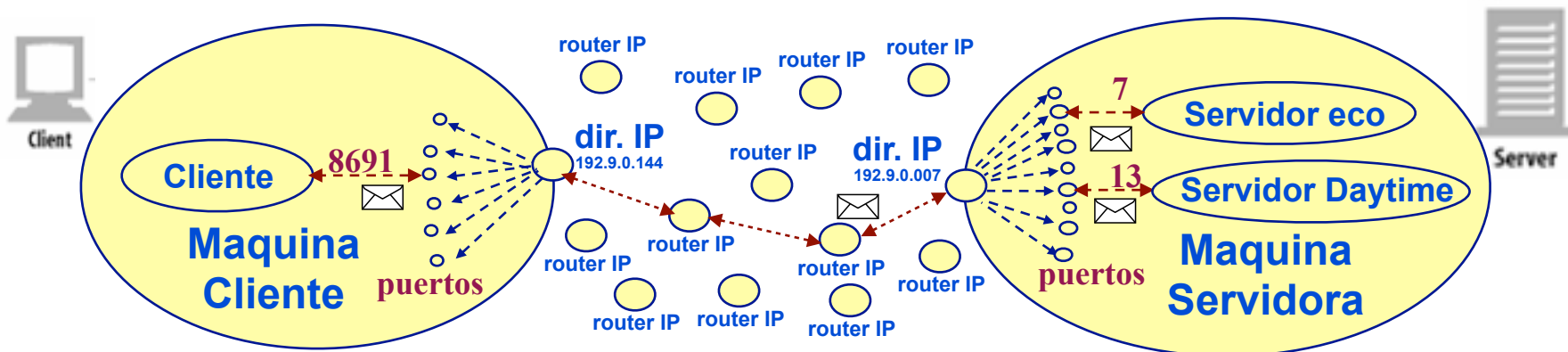


## ◆ Servicio UDP

- Servicio **sin-conexión** (connectionless) de tipo **best-effort** basado en **datagramas**
  - ♦ Los mensajes se envían en paquetes IP que no son fiables y se pueden **perder**, **desordenar** y **duplicar**
    - Si la red esta descargada el envío suele ser fiable y rápido, pero si está cargada suele tener perdidas
  - ♦ La comunicación interactiva de **voz** y **video** debe tener **bajo retardo**, aunque haya **perdidas**, y utilizan UDP
  - ♦ Además del servicio **unicast** (servicio uno a uno de este tema), UDP soporta **broadcast** y **multicast**

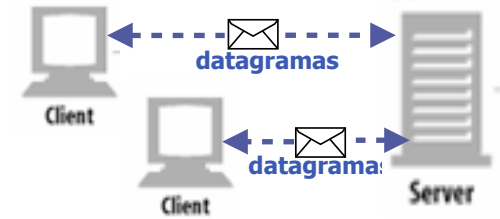
## ◆ Socket UDP

- UDP solo tiene un tipo de sockets que utilizan tanto clientes y servidores
  - ♦ La distinción entre cliente y servidor es aquí de tipo funcional y operativo
    - Los **clientes** (portátiles, tabletas, móviles, ...) conectan usuarios con los **servidores** que suelen estar en la nube





# Sockets UDP



## ◆ Clase **dgram.Socket**

- Objetos para enviar datagramas
  - ◆ Modulo **dgram** de node.js: <https://nodejs.org/api/dgram.html>

## ◆ Lo sockets UDP se crean con esta factoría

- **let socketUDP = dgram.createSocket('udp4');**
  - ◆ **'udp4'** indica que el socket utilizará IPv4 y **'udp6'** que utilizará IPv6

## ◆ Elementos de Socket

- **socketUDP.send(<data>, <port>, <host>)**
  - ◆ Envía un datagrama a la aplicación en **<port>** de **<host>** que transporta **<data>**
- **socketUDP.bind(<port>)**
  - ◆ **Asocia** un socketUDP a un **puerto**, para que escuche en él
- **socketUDP.connect(<port>, <host>)**
  - ◆ Configura el socket para que solo reciba datagramas del servidor en **<port>** de **<host>**
- **socketUDP.close()**
  - ◆ **destruye** el socket, que ya no podrá recibir mas datagramas
- Eventos: **'message'** (datagrama disponible), **'error'** (cualquier tipo de error), **close**, ...

# Servidor y Cliente UDP (Eco)

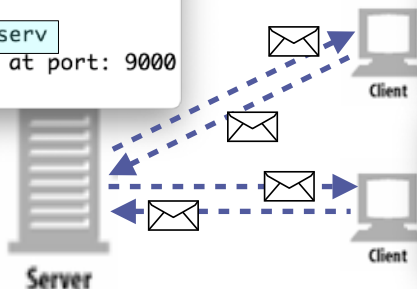
Se arranca el servidor de **chat** en el puerto 9000.

```
$
$
$ node 85-UDP_serv
UDP Echo server at port: 9000
```

Se conecta el cliente 1 al puerto 9000 de la misma máquina (localhost).

```
$
$ node 86-UDP_client
Client 1 says hi!
Client 1 says hi!
...
```

```
$
$ node 86-UDP_client
Client 2 says hi!
Client 2 says hi!
.....
```



// Usage: "node 85-UDP\_serv <port>"

```
let datagram = require('dgram');
let port = process.argv[2] || 9000;
```

Se crea de socket UDP para IPv4.

// Create socket

```
let socketUDP = datagram.createSocket('udp4');
```

```
socketUDP.on('message', (data, client) => { // Echo
  socketUDP.send(data, client.port, client.address);
});
```

Hace eco del msg que llega del cliente.

```
socketUDP.on('error', (err) => {
  socketUDP.close();
});
```

Si ocurre cualquier error se destruye el socket.

Si arranca el socket en el puerto indicado o por defecto.

```
socketUDP.bind(port); // Start server at port
```

```
console.log(`UDP Echo server at port: ${port}`);
```

Se importa el modulo dgram de gestión de sockets.

// Usage: "node 86-UDP\_client <host> <port>"

```
let datagram = require('dgram');
let host = process.argv[2] || "localhost";
let port = process.argv[3] || 9000;
```

Se crea de socket UDP para IPv4.

Valores por defecto de **host** y **port**.

// Create socket

```
let socketUDP = datagram.createSocket('udp4');
```

// Send keyboard msg to s

```
process.stdin.on('data', (data) => {
  socketUDP.send(data, port, host);
});
```

Al teclear una línea ocurre el evento 'data', que la reenvía al servidor.

// Send server msg to s

```
socketUDP.on('message', (data) => {
  process.stdout.write(data+"");
});
```

Al llegar un datagrama ocurre el evento 'message', que se reenvía a screen.

// a possible host or DNS e

```
socketUDP.on('error', function(err) {
  console.log(`Error: ${host}:${port}`);
  socketUDP.destroy();
  process.exit();
});
```

Si ocurre cualquier error se destruye el socket. Probablemente haya sido un host inexistente.

```
process.stdin.resume(); // Activa
```

La entrada de teclado (stdin) debe activarse.



JavaScript

# Final del tema