TSR - PRÁCTICA 1 CURSO 2021/22

INTRODUCCIÓN AL LABORATORIO: JAVASCRIPT Y NODEJS

Esta práctica consta de tres sesiones, a realizar en sesiones de escritorio LINUX, y tiene los siguientes objetivos:

1. Introducir los procedimientos y las herramientas necesarias para el trabajo en el laboratorio de TSR.
2. Introducir técnicas básicas para la programación y desarrollo en JavaScript y NodeJS.

Es necesario haber tenido un contacto previo con este tipo de actividades, propósito cubierto por la **`Práctica 0`**.

- Entre el material disponible en su carpeta[1] del área de Recursos de PoliformaT conviene destacar los documentos de introducción a los laboratorios del DSIC, y de introducción al laboratorio de TSR. Entre ambos se describe la operativa para las prácticas de la asignatura.
- Es de destacar que en la redacción de este boletín se asume el uso de las sesiones de escritorio en el laboratorio o mediante **`Polilabs`**, pero es sencillo reinterpretarlo para su aplicación en otros escenarios…
    - La imagen equivalente para VirtualBox
    - Vuestro servidor virtual de portal
    - Directamente sobre vuestro equipo personal (si disponéis de todos los requisitos)

[1] Consultar el apartado final **`Referencias`**

INTRODUCCIÓN

Conflictos y herramientas

Las prácticas se desarrollan en JavaScript sobre el entorno NodeJS

Cuando trabajamos con los equipos de escritorio de los laboratorios del DSIC, realmente lo hacemos en sesiones que se inician en máquinas virtuales compartidas. Esta característica **no es compatible** con el uso **de puertos** u otros recursos exclusivos, particularmente en NodeJS, razón por la que se necesitará alguna organización que impida la interferencia entre el trabajo de varios alumnos.

- En resumen: l@s usuari@s compiten por el uso de los mismos puertos cuando se encuentran resolviendo el mismo problema.

Los números de puerto que aparecen en los enunciados serán reinterpretados y modificados para que no coincidan entre alumnos diferentes. Dado que en el laboratorio hemos reservado el rango 50000 a 59999 (el 5 es fijo), y que ningún ejercicio requiere usar más de 10 puertos simultáneamente (los representaríamos con la cifra de la unidad, de 0 a 9), nos quedan 3 cifras para diferenciar cada alumno. Un criterio consiste en tomar los últimos 3 dígitos del DNI.

- Así, una alumna con DNI 29332481 dispondrá de los puertos 54810 a 54819, con la *casi* seguridad de no entrar en conflicto con otr@s compañer@s.
- En un ejercicio que cite los puertos 8000, 8001 y 8002, la alumna anterior puede reinterpretar esos puertos como 54810 (en vez de 8000), 54811 (en vez de 8001) y 54812 (en lugar de 8002).

Las características de estas máquinas de escritorio compartidas son:

- LINUX (Ubuntu de 64 bits), acceso a vuestro almacenamiento en red.
- Entorno NodeJS (versión 12).
- Bibliotecas auxiliares básicas (fs, http, net, ...)
- Gestor de paquetes npm (permite instalar cualquier otra biblioteca de NodeJS)
    - Necesitarás ejecutar “npm init” si lo instalas por tu cuenta
- Editores de textos (en el menú Programación). Podemos utilizar cualquiera que incluya resaltado sintáctico, pero en la Práctica 0 ya se recomendó Visual Studio Code.

Cualquier alumno puede instalar un entorno similar en sus propios equipos bajo Windows, LINUX o MacOS, sin las limitaciones descritas en cuanto a puertos. Consultar http://nodejs.org

Procedimientos

Cláusula de buen uso del laboratorio

Los recursos que se ponen a disposición del alumno se justifican por su carácter educativo. La correcta utilización de los mismos es responsabilidad del alumno, que se compromete a:

- Velar por la confidencialidad de sus claves de acceso.
- No interferir en la actividad de los demás compañeros.
- No utilizar dichos recursos para fines distintos a los descritos en los boletines de prácticas

En las instalaciones existen mecanismos que monitorizan el uso de los recursos y aportan informaciones que pueden ser consultadas con posterioridad.

Por lo general escribimos con cualquier editor el código JavaScript en un fichero con la extensión js (x.js), y ejecutamos dicho código con la orden **node x[.js] [argsOpcionales]**

Esta práctica consta de tres sesiones que se organizan en dos partes: la primera, dedicada a JavaScript; y la segunda, a NodeJS. Su desarrollo será guiado por el profesor.

Debe tomarse en cuenta que durante el desarrollo de las prácticas pueden aparecer distintos **tipos de errores**. Algunos errores frecuentes son los siguientes.

Error Sintáctico		
Definición	Detección/solución	Ejemplo
Sintaxis ilegal (identificador no válido, estructuras mal anidadas, etc)	Al interpretar el código se indica tipo de error y su ubicación en el código. Corregir sobre código fuente	> "patata"(); TypeError: string is not a function at repl:1:9 at REPLServer.defaultEval (repl.js:132:27) at bound (domain.js:254:14) at REPLServer.runBound [as eval] (domain.js:267:12) at REPLServer.<anonymous> (repl.js:279:12) ....
Error Lógico		
Definición	Detección/solución	Ejemplo
Error de programación (intentar acceder a una propiedad inexistente o 'undefined', pasar como argumento un tipo de datos incorrecto, etc.) JavaScript no es fuertemente orientado a tipos: algunos errores que otros lenguajes capturan al compilar, aquí sólo se muestran al ejecutar	Resultados incorrectos al ejecutar. Modificar el código. Conviene que en el código se verifiquen siempre las restricciones a aplicar sobre los argumentos de las funciones	> function suma(array) {return array.reduce(function(x,y){return x+y})} undefined > suma([1,2,3]) 6 > suma(1) TypeError: undefined is not a function at suma (repl:1:36) at repl:1:1 at REPLServer.defaultEval (repl.js:132:27) at bound (domain.js:254:14) at REPLServer.runBound [as eval] (domain.js:267:12) ....

Error Operacional	
Definición	Detección/solución
Situaciones excepcionales que pueden surgir durante el funcionamiento normal del programa. Pueden deberse a: • el entorno (ej. nos quedamos sin memoria, demasiados ficheros abiertos) • la configuración del sistema (ej. no hay una ruta hacia un host remoto) • uso de la red (ej. problemas con el uso de sockets) • problemas de acceso a un servicio remoto (ej. no puedo conectar con el servidor) • inconsistencia en datos introducidos por el usuario • etc.	construcciones try, catch, throw, similares a las de Java Estrategia: - Si está claro cómo resolver el error, gestiónalo (ej error al abrir un fichero para escritura -> crearlo como fichero nuevo) - Si la gestión del error no es responsabilidad de ese fragmento de programa, propagar el error al invocante - Para errores transitorios (ej problemas con la red), reintentar la operación Si no podemos gestionar ni propagar el error - si impide continuar con el programa, aborta - en otro caso, anota el error en un log

Para minimizar los posibles errores, se recomienda:

- utilizar el modo estricto: **`node --use_strict fichero.js`**
- documentar correctamente cada función de interfaz
    - significado y tipo de cada argumento, así como cualquier restricción adicional
    - qué tipo de errores operacionales pueden aparecer, y cómo se van a gestionar
    - el valor de retorno

PRIMERA PARTE

Ejecución de programas JavaScript

Adjunta, en la documentación asociada a esta práctica de laboratorio, se encuentra la carpeta denominada **`javascript`** que contiene varios programas de introducción a los requerimientos de la asignatura.

- En estos programas se abordan algunas características básicas de JavaScript tales como: cláusula **this** (junto con la función **bind**), clausuras, programación asíncrona, etc.
- Puedes ver la relación de los mismos en el apartado final de este texto.

La actividad asociada a esta parte consiste en el análisis, estudio y ejecución de los códigos antes mencionados, sacando las pertinentes conclusiones.

Se dejan al profesor los detalles explicativos adicionales, junto con la planificación y desarrollo de esta primera parte.

SEGUNDA PARTE

Introducción a NodeJS

Acceso a ficheros

Todos los métodos correspondientes a operaciones sobre ficheros aparecen en el módulo fs.js. Las operaciones son asíncronas, pero para cada función asíncrona **xx** suele existir la variante síncrona **xxSync**. Los siguientes códigos pueden ser copiados y ejecutados.

- Leer asíncronamente el contenido de un fichero (**`read1.js`**)

```
const fs = require('fs');
fs.readFile('/etc/hosts', 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
  console.log(data);
});
```

- Escribir asíncronamente el contenido en un fichero (**`write1.js`**)

```
const fs = require('fs');
fs.writeFile('/tmp/f', 'contenido del nuevo fichero', 'utf8',
  function (err) {
  if (err) {
    return console.log(err);
  }
  console.log('se ha completado la escritura');
});
```

- Escrituras y lecturas adaptadas. Utilización de módulos.

Definimos el módulo **`fiSys.js`**, basado en **`fs.js`**, para acceder a ficheros junto con algunos ejemplos de su uso.

(**`fiSys.js`**)

```
//Módulo fiSys
//Ejemplo de módulo de funciones adaptadas para el uso de ficheros.
//(Podrían haberse definido más funciones.)

const fs=require("fs");

function readFile(fichero,callbackError,callbackLectura){
	fs.readFile(fichero,"utf8",function(error,datos){
		if(error) callbackError(fichero);
		    else callbackLectura(datos);

	});
}


function readFileSync(fichero){
	var resultado; //retornará undefined si ocurre algún error en la lectura
	try{
		resultado=fs.readFileSync(fichero,"utf8");
	}catch(e){};
	return resultado;
}


function writeFile(fichero,datos,callbackError,callbackEscritura){
	fs.writeFile(fichero,datos,function(error){
			if(error) callbackError(fichero);
			    else callbackEscritura(fichero);
	});
}

exports.readFile=readFile;
exports.readFileSync=readFileSync;
exports.writeFile=writeFile;
```

(**read2.js**)

```
//Lecturas de ficheros

const fiSys=require("./fiSys");


//Para la lectura asíncrona:
console.log("Invocación lectura asíncrona");
fiSys.readFile("/proc/loadavg",cbError,formato);
console.log("Lectura asíncrona invocada\n\n");

//Lectura síncrona
console.log("Invocación lectura síncrona");
const datos=fiSys.readFileSync("/proc/loadavg");
if(datos!=undefined)formato(datos);
        else console.log(datos);
console.log("Lectura síncrona finalizada\n\n");

//-----------
function formato(datos){
        const separador=" "; //espacio
        const tokens = datos.toString().split(separador);
        const min1 = parseFloat(tokens[0])+0.01;
        const min5 = parseFloat(tokens[1])+0.01;
        const min15 = parseFloat(tokens[2])+0.01;
        const resultado=min1*10+min5*2+min15;
        console.log(resultado);
}

function cbError(fichero){
        console.log("ERROR DE LECTURA en "+fichero);
}
```

(**write2.js**)

```
//Escritura asíncrona de ficheros

const fiSys = require('./fiSys');

fiSys.writeFile('texto.txt','contenido del nuevo fichero',cbError,cbEscritura);

function cbEscritura(fichero){
        console.log("escritura realizada en: "+fichero);
}

function cbError(fichero){
        console.log("ERROR DE ESCRITURA en "+fichero);
}
```

Programación asíncrona: eventos particularizados programados

Uso del módulo events. Analiza el código siguiente y comprueba su funcionamiento.
(**`emisor1.js`**)

```
const ev = require('events')              // library import (Using events module)

const emitter = new ev.EventEmitter()     // Create new event emitter
const e1='print', e2='read'               // identity of two different events

function handler (event,n) {    // function declaration, dynamic type args, higher-order
function
    return () => { // anonymous func, parameterless listener, closure
      console.log(event + ':' + ++n + ' times')
    }
}

emitter.on(e1, handler(e1,0)) // listener, higher-order func (callback)
emitter.on(e2, handler(e2,0)) // listener, higher-order func (callback)
emitter.on(e1, ()=>{console.log('something has been printed')}) //several listeners on e1

emitter.emit(e1) // emit event
emitter.emit(e2) // emit event

console.log('--------------------------')
setInterval(()=>{emitter.emit(e1)}, 2000) // asynchronous (event loop), setInterval
setInterval(()=>{emitter.emit(e2)}, 8000) // asynchronous (event loop), setInterval
console.log('\n\t========> end of code')
```

Analiza este otro ejemplo (emisor2.js). Al generar eventos se pueden generar valores asociados (argumentos para el oyente del evento).
(**`emisor2.js`**)

```
const ev = require('events')

const emitter = new ev.EventEmitter()
const e1='e1', e2='e2'

function handler (event,n) {
   return (incr)=>{ // listener with param
      n+=incr
      console.log(event + ':' + n)
   }
}

emitter.on(e1, handler(e1,0))
emitter.on(e2, handler(e2,'')) // implicit type casting

console.log('\n\n-------------------------- init\n\n')
for (let i=1; i<4; i++) emitter.emit(e1,i) // sequence, iteration, generation with param
console.log('\n\n-------------------------- intermediate\n\n')
for (let i=1; i<4; i++) emitter.emit(e2,i) // sequence, iteration, generation with param
console.log('\n\n-------------------------- end')
```

Actividad: Completa el fichero emisor3.js para cumplir las siguientes especificaciones:

(**`emisor3.js`**)

```
...
const e1='e1', e2='e2'
let inc=0, t

function rand() { // debe devolver valores aleat en rango [2000,5000) (ms)
   ... // Math.floor(x) devuelve la parte entera del valor x
   ... // Math.random() devuelve un valor en el rango [0,1)
}

function handler (e,n) { // e es el evento, n el valor asociado
   return (inc) => {..} // el oyente recibe un valor (inc)
}

emitter.on(e1, handler(e1,0))
emitter.on(e2, handler(e2,''))

function etapa() {
   ...
}

setTimeout(etapa,t=rand())
```

Durante la ejecución se completan distintas etapas, cada una con una duración aleatoria entre 2 y 5 segundos. Al completarse cada etapa:

- Se emitirán los eventos e1 y e2, pasando como valor asociado el de la variable inc
- La variable inc aumenta una unidad
- Al finalizar cada etapa debe mostrarse por consola la duración de la misma

Ejemplo de ejecución (únicamente el fragmento inicial)

```
e1 --> 0
e2 --> 0
duracion 3043
e1 --> 1
e2 --> 01
duracion 3869
e1 --> 3
e2 --> 012
duracion 2072
e1 --> 6
e2 --> 0123
duracion 2025
e1 --> 10
e2 --> 01234
```

Interacción cliente/servidor

MÓDULO HTTP

Para desarrollo de servidores Web (servidores HTTP)

Ej.- servidor web (**`ejemploSencillo.js`**) que saluda al cliente que contacta con él

Código	comentario
`const http = require('http');`	Importa módulo http
`function dd(i) {return (i<10?"0":"")+i;}`	dd(8) -> "08" dd(16) -> "16"

`const server = http.createServer(`
`  function (req,res) {`
`  res.writeHead(200,{'Content-Type':'text/html'});`
`  res.end('<marquee>Node y Http</marquee>');`
`  var d = new Date();`
`  console.log('alguien ha accedido a las '+`
`  d.getHours() +":"+`
`  dd(d.getMinutes()) +":"+`
`  dd(d.getSeconds()));`
`}).listen(8000);`
	crea el servidor y
	le asocia esta función
	que devuelve una
	respuesta fija
	y
	además escribe la
	hora en la consola
	
	El servidor escucha en
	el port 8000

Como se trata del primer caso de un ejercicio que usa puertos, debes recordar la posibilidad de conflicto. Si tu DNI fuera, p.ej., 29332481 dispones de los puertos 54810 a 54819. Puedes sustituir el número indicado en el enunciado (8000) por el primero libre a tu disposición (54810 en este ejemplo).

Ejecuta el servidor y utiliza un navegador web como cliente

- Accede a la URL **`http://localhost:8000`**

  (**`http://localhost:54810`** en el ejemplo de usuario comentado)

- Comprueba en el navegador la respuesta del servidor, y en la consola el mensaje escrito por el servidor

MÓDULO NET

Cliente (`netClient.js`)

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

const client = net.connect({port:8000},
   function() { //connect listener
      console.log('client connected');
      client.write('world!\r\n');
   });

client.on('data',
   function(data) {
      console.log(data.toString());
      client.end(); //no more data written to the
stream
   });

client.on('end',
   function() {
      console.log('client disconnected');
   });
```

Servidor (`netServer.js`)

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

const server = net.createServer(
   function(c) { //connection listener
      console.log('server: client connected');
      c.on('end',
         function() {
            console.log('server: client disconnected');
         });
      c.on('data',
         function(data) {
            c.write('Hello\r\n'+ data.toString()); // send
resp
            c.end(); // close socket
         });
   });

server.listen(8000,
   function() { //listening listener
      console.log('server bound');
   });
```

Acceso a argumentos en línea de órdenes

El shell recoge todos los argumentos en línea de órdenes y se los pasa a la aplicación JS empaquetados en un array denominado process.argv (abreviatura de 'argument values'), por lo que podemos calcular su longitud y acceder a cada argumento por su posición

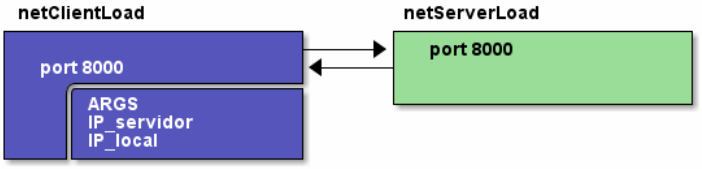
- **`process.argv.length:`** número de argumentos pasados por la línea de órdenes
- **`process.argv[i]`** : permite obtener el argumento i-ésimo. Si hemos usado la orden "**`node programa arg1`** …" entonces **`process.argv[0]`** contiene la cadena '**`node`**', **`process.argv[1]`** contiene la cadena '**`programa`**', **`process.argv[2]`** la cadena **`'arg1'`**, etc.

Téngase en cuenta que puede utilizarse **args=process.argv.slice(2)** para descartar 'node' y el path del programa, de forma que en **args** sólo quedarán los argumentos reales para la aplicación encapsulados y accesibles como elementos de un array, a partir de la posición 0.

Consulta la carga del equipo

En un entorno cliente/servidor a gran escala suelen replicarse los servidores (replicación horizontal), y repartir el trabajo entre ellos según su nivel actual de carga.

Creamos el embrión para un sistema de ese tipo, con un único cliente y un único servidor (posiblemente en máquinas distintas) que se comunican a través del puerto 8000



- El servidor se denomina **`netServerLoad.js`**, y no recibe argumentos en línea de órdenes.
  - La siguiente función **getLoad** calcula su carga actual (puedes copiar el código directamente). Dicha función lee los datos del fichero /proc/loadavg, filtra los valores que le interesa (les suma una centésima para evitar la confusión entre el valor 0 y un error), y los procesa calculando una media ponderada (peso 10 a la carga del último minuto, peso 2 a los últimos 5 minutos, peso 1 a los últimos 15)

```
function getLoad(){
  data=fs.readFileSync("/proc/loadavg"); //requiere fs
  var tokens = data.toString().split(' ');
  var min1  = parseFloat(tokens[0])+0.01;
  var min5  = parseFloat(tokens[1])+0.01;
  var min15 = parseFloat(tokens[2])+0.01;
  return min1*10+min5*2+min15;
};
```

- El cliente se denomina **`netClientLoad.js`**: recibe como argumentos en línea de órdenes la dirección IP del servidor y su IP local
- Protocolo: Cuando el cliente envía una petición al servidor, incluye su propia IP, el servidor calcula su carga y devuelve una respuesta al cliente en la que incluye la propia IP del servidor y el nivel de carga calculado con la función **getLoad**.

Para facilitar las pruebas y depuración, se ha dejado un servidor activo en la IP **tsr1.dsic.upv.es**

Importante:

- Debes partir del código de netClient.js y netServer.js descritos en el punto anterior
- Hay que asegurarse de que todo proceso finaliza (ej. con **`process.exit()`**)
- Se puede averiguar la IP desde la interfaz web del portal, o usando la orden **ip addr**
- Completa ambos programas, colócalos en equipos diferentes colaborando con algún compañero, y haz que se comuniquen mediante el puerto 8000: **`netServerLoad`** debe

calcular la carga como respuesta a cada petición recibida desde el cliente, y **`netClientLoad`** debe mostrar la respuesta en pantalla.

Intermediario entre 2 equipos (proxy transparente)

Un intermediario o proxy es un servidor que al ser invocado por el cliente redirige la petición a un tercero, y posteriormente encamina la respuesta final de nuevo al cliente.

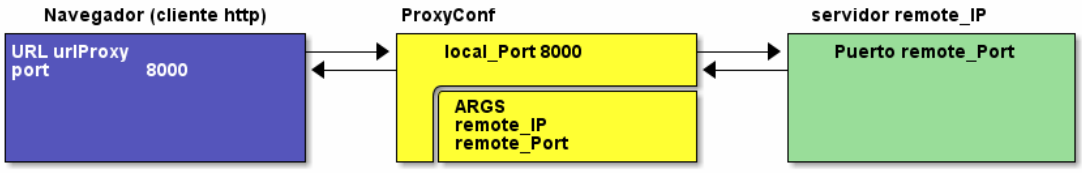
- Desde el punto de vista del cliente, se trata de un servidor normal (oculta al servidor que realmente completa el servicio)
- Puede residir en una máquina distinta a la del cliente y la del servidor
- El intermediario puede modificar puertos y direcciones, pero no altera el cuerpo de la petición ni de la respuesta

Es la funcionalidad que ofrece un redirector, como un proxy HTTP, una pasarela ssh o un servicio similar. Más información en http://en.wikipedia.org/wiki/Proxy_server

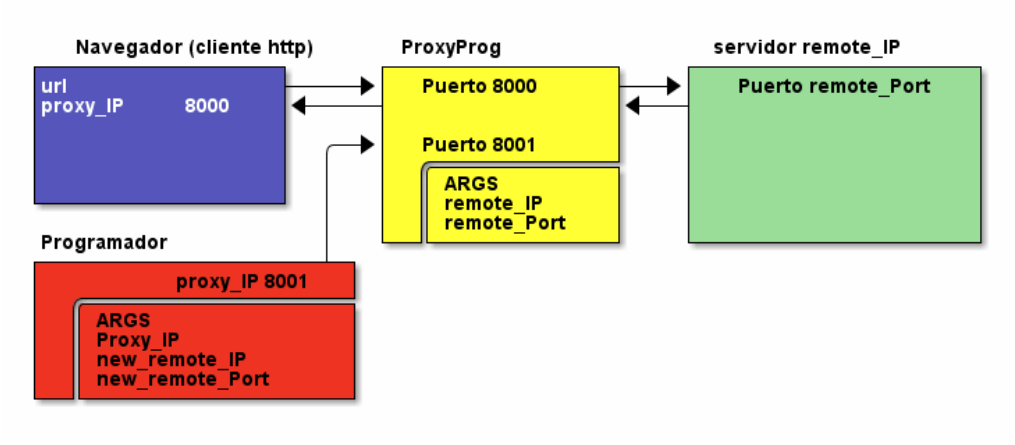
Planteamos tres pasos incrementales:

1. Proxy básico (**Proxy**). Cuando el cliente http (ej navegador) contacta con el proxy en el puerto 8000, el proxy redirige la petición al servidor web memex (158.42.185.55 puerto 8080), y posteriormente devuelve la respuesta del servidor al invocante

2. Proxy configurable (**ProxyConf**): en lugar de IP y port remotos fijos en el código, los recibe como argumentos en línea de órdenes



3. Proxy programable (**ProxyProg**). Los valores de IP y port del servidor se toman inicialmente desde línea de órdenes, pero posteriormente se modifican en respuesta a mensajes del programador en el port 8001



Importante:

- Te proporcionamos el código de Proxy: analízalo hasta comprender su funcionamiento
  - Comprueba que su funcionamiento es correcto usando un navegador web que apunte a http://direccion_del_proxy:8000/

Código (Proxy.js)	**comentarios**

`const net = require('net');`
 	
`const LOCAL_PORT = 8000;`
`const LOCAL_IP = '127.0.0.1';`
`const REMOTE_PORT = 80;`
`const REMOTE_IP = '158.42.4.23'; // www.upv.es`
 	
`const server = net.createServer(function (socket) {`
`  const serviceSocket = new net.Socket();`
`  serviceSocket.connect(parseInt(REMOTE_PORT),`
`  REMOTE_IP, function () {`
`  socket.on('data', function (msg) {`
`  serviceSocket.write(msg);`
`  });`
`  serviceSocket.on('data', function (data) {`
`  socket.write(data);`
`  });`
`  });`
`}).listen(LOCAL_PORT, LOCAL_IP);`
`console.log("TCP server accepting connection on port: " + LOCAL_PORT);`
	Usa un socket para dialogar con el cliente (socket) y otro para dialogar con el servidor (serviceSocket)
	
	1.- lee un mensaje (`msg`) del cliente
	2.- abre una conexión con el servidor
	3.- escribe una copia del mensaje
	4.- espera la respuesta del servidor y devuelve una copia al cliente

- En el escenario 2, debemos realizar las siguientes pruebas:
  - Acceso al servidor de la UPV (seguimos utilizando como puerto local de atención de peticiones el puerto 8000)
  - ProxyConf como intermediario entre netClientLoad y netServerLoad
    - Si ProxyConf se ejecuta en la misma máquina que netServerLoad tenemos una colisión en el uso de puertos -> modifica el código de netServerLoad para que use otro puerto
    - Si al ejecutar el programa aparece el error "EADDRINUSE", indica que estamos referenciando un puerto que ya está en uso por parte de otro programa
- En el escenario 3 es necesario codificar el cliente que actúa como programador
  - Dicho programa (**`programador.js`**). recibe en línea de órdenes la dirección IP del proxy, y los nuevos valores de IP y puerto correspondientes al servidor
  - El programa codifica esos valores y los remite como mensaje al port 8001 del proxy, tras lo cual termina
  - El **`programador.js`** debería enviar mensajes con un contenido como el siguiente:

```
var msg = JSON.stringify ({'remote_ip':"158.42.4.23", 'remote_port':80})
```

Puedes experimentar usando como servidores de destino los dos ya referenciados (memex.dsic.upv.es, IP 158.42.186.55, puerto 8080; y www.upv.es, IP 158.42.4.23, puerto 80) o cualquiera que utilice HTTP (version “no segura”), como www.aemet.es (IP 212.128.97.138, puerto 80).

REFERENCIAS

1. `Laboratorio/Pract 0(autoaprendizaje)`, en la zona de recursos de la asignatura (https://poliformat.upv.es/x/RB2ItL) en PoliformaT, citado al comienzo de este boletín.
2. `Laboratorio/Prac1`, en la zona de recursos de la asignatura , destacando los ejemplos de la carpeta `javascript` empleados en esta práctica:

`j00.js`	Uso de funciones, objetos, cláusula this y función bind().
`j01.js`	Declaración de variables. Uso de funciones y clausuras.
`j02.js`, `j03.js`	Clausuras de variables y funciones.
`j04.js`, `j05.js`, `j06.js`, `j07.js`, `j08.js`, `j09.js`, `j10.js`, `j11.js`	Uso de operaciones asíncronas, modeladas con la función setTimeout.
`j12.js`	Ejercicio de recapitulación (operaciones asíncronas y clausuras)
`j13.js`	Ejecución de varias operaciones asíncronas paralelizadas al estilo fork-join
`j14.js`	Ejecución de varias operaciones asíncronas serializadas.