TSR - PRÁCTICA 1
(Segunda parte)

NODEJS

La práctica 1 consiste en 2 partes, una parte dedicada a JavaScript de 1 sesión y una parte dedicada a NodeJS de 2 sesiones. **Este documento describe la segunda parte de la práctica, dedicada a NodeJS**.

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.

INTRODUCCIÓN

Herramientas

Las prácticas se desarrollan en las **máquinas virtuales de portal**, en JavaScript sobre el entorno NodeJS. Esto elimina los posibles conflictos derivados de la compartición de recursos en poliLabs.

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

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]**

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

PRÁCTICA 1 (SEGUNDA PARTE): NODEJS

Para esta segunda parte disponemos de 2 sesiones. Se propone un trabajo a realizar para cada una de las sesiones. Sin embargo el alumno puede gestionarse su tiempo y dedicar más o menos a cada una de las sesiones que proponemos para terminar por completar todos los aspectos que se mencionan.

- **Sesión 1.2: Módulos de NodeJS.**
- **Sesión 1.3: Proxy inverso.**

SESIÓN 1.2: MÓDULOS DE NODEJS.

NodeJS incluye gran variedad de módulos que proporcionan funcionalidad útil para trabajar mediante ficheros, con la red, con aspectos de seguridad, etc. En esta sesión vamos a trabajar con algunos de los módulos más relevantes.

Se van a tratar los módulos: **fs, events, http, net.**

Durante esta sesión deberás practicar con el código que se proporciona como ejemplo para cada uno de estos módulos, y contestar las cuestiones o realizar las actividades que se proponen.

1.2.1 MÓDULO FS: 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.

Lee los siguientes apartados A, B y C de esta sección. Para cada uno de ellos copia el código, analízalo y ejecútalo hasta que lo comprendas. Al final hay una serie de preguntas que debes contestar.

A. 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);
});
```

B. 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');
});
```

C. 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**)

→ Bucle de eventos ⇒ O

```
//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;
```

(1) Importación de librerías

(2) Definición de métodos

(3) Métodos que han querido ser exportados para poder usarse como herramienta. (Todos)

Programa de lectura mediante el módulo "fiSys" (**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);
}
```

→ Bucle de eventos → 2

Programa de escritura mediante el módulo fiSys (**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);
}
```

→ Bucle de eventos ⇒ 2

Cuestiones

1. Razona cuántas iteraciones del bucle de eventos (turnos) suceden en cada uno de los programas analizados.

2. En el apartado C se ha desarrollado un módulo de usuario. Detalla los pasos que se han realizado en el código para crear un módulo, a diferencia del código necesario para programar una aplicación.

3. Realiza 2 versiones nuevas del apartado A. Una mediante promesas y otra mediante async/await. Utiliza el material de teoría como base.

1.2.2 MÓDULO EVENTS

El módulo "events" proporciona cierta funcionalidad para el uso de eventos en NodeJS. Es destacable el hecho de que varios módulos de NodeJS utilizan este módulo para gestionar eventos. Se te proporcionan dos programas que debes comprender y debes completar el tercero con tu propio código.

A. Programa que usa el módulo "events". (**`emisor1.js`**)

Analiza y ejecuta el programa, hasta comprender su funcionamiento.

```
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')
```

B. Programa que usa el módulo "events". (**emisor2.js**)

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

```
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

Se te proporciona el código del programa emisor3, y lo debes completar para que cumpla ciertas especificaciones que detallaremos a continuación.

Código a completar (**emisor3.js**)

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

function rand() { // debe devolver valores aleatorios 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())
```

Completa el código (añadiendo código en lugar de los puntos suspensivos) sin eliminar código para que cumpla con lo siguiente:

El código debe llamar una primera vez a la función "etapa". Dentro de la función "etapa" se debe reprogramar la llamada a "etapa" para que el programa ejecute una serie de "etapas".

Cada etapa debe iniciarse con un retraso de entre 2 y 5 segundos. Cada etapa debe hacer lo siguiente:

- Emitir los eventos e1 y e2, pasando como valor asociado el de la variable "inc".
- Aumentar la variable "inc" una unidad
- Mostrar por consola el retraso en la ejecución de la etapa.

Ejemplo de ejecución (únicamente el fragmento inicial)

```
e1 --> 0
e2 --> 0
etapa 1 iniciada después de 3043 ms
e1 --> 1
e2 --> 01
etapa 2 iniciada después de 3869 ms
e1 --> 3
e2 --> 012
etapa 3 iniciada después de 2072 ms
e1 --> 6
e2 --> 0123
etapa 4 iniciada después de  2025 ms
e1 --> 10
e2 --> 01234
etapa 5 iniciada después de 2325 ms
```

Cuestiones

El código que se ha propuesto para "emisor3", debe reprogramar cada nueva etapa dentro de la propia etapa. Razona qué implicaciones tendría programar todas las etapas de golpe con un código similar al siguiente:

```
t = 0
for (let i=0; i<=10; i++) {
      t = t + rand();
      setTimeout(etapa, t);
}
```

Solo habría 10 eventos en el bucle de eventos mientras que de la otra manera haces que la misma función "etapa()" pesdura ya que ahí hace la llamada

1.2.3 MÓDULO HTTP

Módulo con funciones para desarrollo de servidores Web (servidores HTTP)

Se te pide que analices y compruebes el funcionamiento del siguiente programa que emplea el módulo "http".

Servidor web (**ejemploSencillo.js**) que saluda al cliente

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

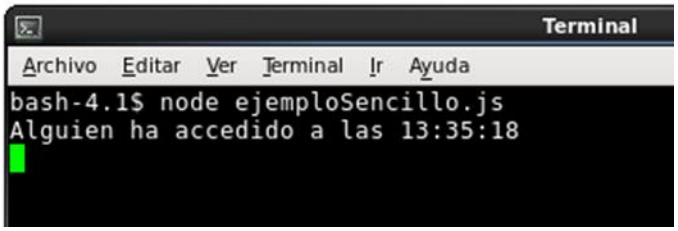
`const server = http.createServer(`
`  function (req,res) {`
`  res.writeHead(200,{'Content-Type':'text/html'});`
`  res.end('<marquee>Node y Http</marquee>');`
`  var d = new Date();`
`  console.log('alguien ha accedido a las '+`
`  d.getHours() +":"+`
`  dd(d.getMinutes()) +":"+`
`  dd(d.getSeconds()));`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

`}).listen(8000);`	El servidor escucha en el port 8000

Ejecuta el servidor y utiliza un navegador web como cliente

- Accede a la URL **http://localhost:8000**
- Comprueba en el navegador la respuesta del servidor, y en la consola el mensaje escrito por el servidor

Node y HTTP



1.2.4 MÓDULO NET

Este módulo contiene la API para emplear sockets básicos TCP/IP.

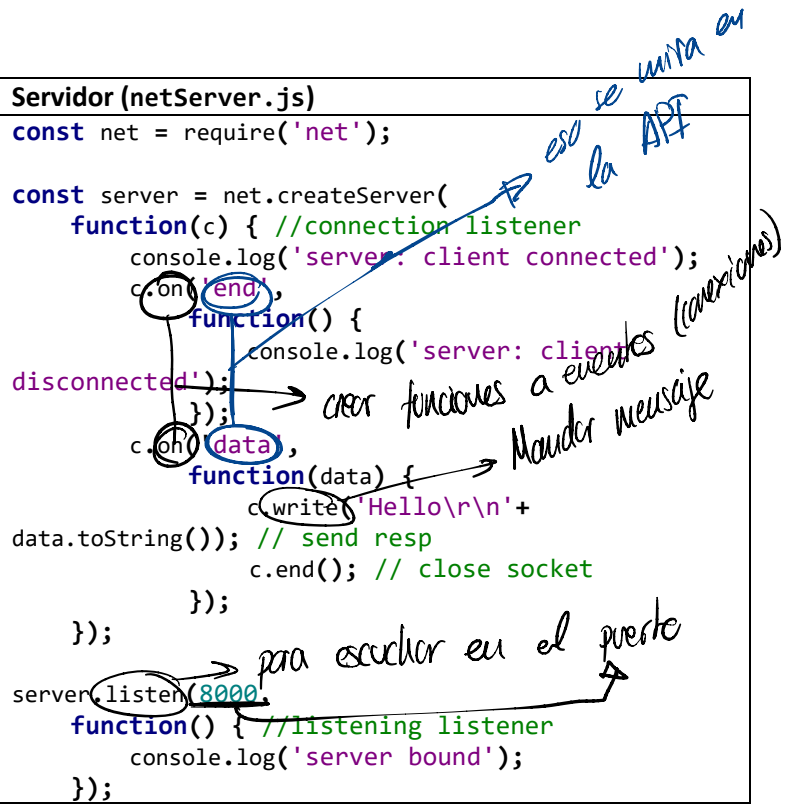
Se proporcionan dos programas, cliente y servidor. Analiza el código de ambos y ejecútalos varias veces para observar su funcionamiento.

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');
    });
```



Actividad

Se pide que modifiques el código de estos programas cliente y servidor para lograr un servicio que proporcione la carga del ordenador servidor. Utilizando "netClient" como base, crea un programa "netClientLoad". Utilizando el programa "netServer", crea un programa al que llamaremos "netServerLoad".

Para realizar esta actividad debes realizar 2 partes:

Parte 1: modificar el programa "netClientLoad" que acabas de crear, para que utilice correctamente la línea de argumentos.

Parte 2: Para actualizar el programa "netServerLoad", se te proporciona la función "getLoad()" que puedes integrar en su código

Parte 1 de la actividad – 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.

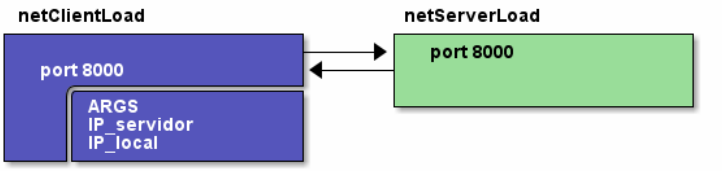
Parte 2 de la actividad. Consulta la carga del equipo

Puedes integrar la función "`getLoad()`" en el código de `netServerLoad`, para que obtenga su carga actual cada vez que un cliente pida el servicio. Nótese que sólo funciona en Linux:

```
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;
 };
```

A título de curiosidad, para que entendamos un poco el código de esta función, podemos observar lo que hace: Esta función lee datos del fichero `/proc/loadavg`. Este pseudo-fichero contiene información relativa a la carga del sistema Linux. Para proporcionar la carga actual, la función 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)

Descripción general de ambos programas.



- El servidor **`netServerLoad`** no recibe argumentos en línea de órdenes y escuchará en cierto puerto (el puerto 8000 por ejemplo).
- El cliente **`netClientLoad`** debe recibir como argumentos en línea de órdenes la dirección IP del servidor y su IP local. Conectará con el servidor en la dirección IP del servidor y el puerto del servidor.
- 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`**.
- Puede ser una buena idea que ambos programas intercambien los datos mediante JSON. Revisa la API `JSON.stringify()` y `JSON.parse()`
- Hay que asegurarse de que el cliente finaliza al obtener la carga (ej. con **`process.exit()`**) o cerrando su conexión.
- Se puede averiguar la IP desde la interfaz web del portal, o usando la orden **`ip addr`**, o **`ifconfig`**
- Completa ambos programas, colócalos en equipos diferentes colaborando con algún compañero (o conectando mediante `ssh`), 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.

SESIÓN 1.3: PROXY INVERSO.

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

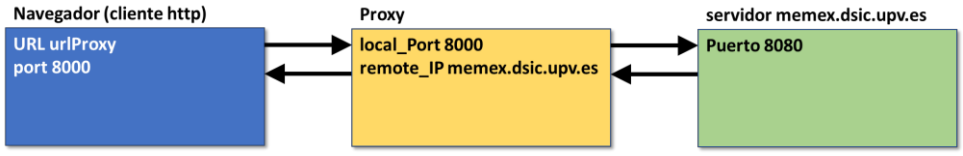
En esta sesión de prácticas vamos a trabajar con 3 versiones de proxy. Te damos la primera versión ya implementada y has de completar las otras 2. Al finalizar has de responder a las cuestiones que plateamos.

Los destinos con los que puedes probar son siempre servidores web que trabajan con la versión *sencilla* del protocolo, sin reenviar a otra dirección. En el curso pasado incluimos:

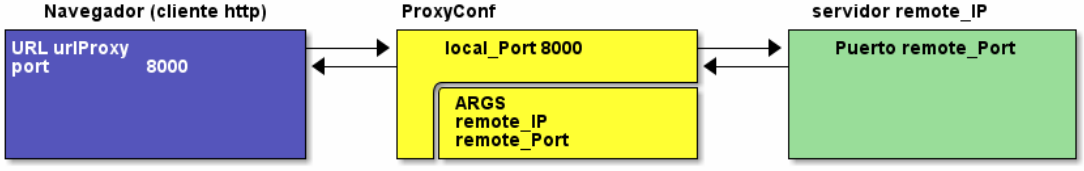
FQDN	IP (v4)	puerto
`memex.dsic.upv.es`	`158.42.186.57`	`8080`
`www.ite.es`	`46.24.7.149`	`80`
`www.yclasicos.com`	`81.25.126.181`	`80`

Las tres versiones a considerar son las siguientes:

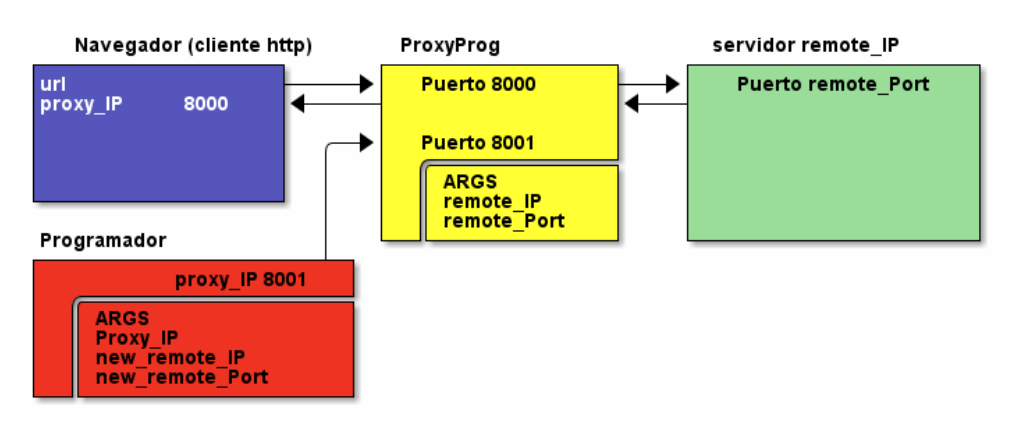
1. Proxy básico (**`Proxy`**). Cuando el cliente HTTP (ej navegador) contacta con el proxy en el puerto 8000, el proxy redirige la petición al servidor web memex (158.42.186.57 puerto 8080), y posteriormente devuelve la respuesta del servidor al invocante



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



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



Te proporcionamos el código de Proxy básico: analízalo hasta comprender su funcionamiento

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

1.- Proxy básico: En esta primera parte no has de programar nada. Has de comprobar su funcionamiento. Comprueba el funcionamiento del proxy usando un navegador web que apunte a http://direccion_del_proxy:8000/

Realiza diferentes pruebas y analiza el código que se te proporciona.

También puedes probar el proxy para que intermedie a un cliente `netClientLoad` y un servidor `netServerLoad`.

2.- Proxy configurable: Tomando como base el proxy básico, crea el proxy configurable (`ProxyConf.js`). Lo único a realizar es analizar la línea de órdenes para obtener de ella la dirección del servidor al que conectará el proxy.

Una vez lo tengamos hecho podemos hacer varias pruebas, iguales a las pruebas que hicimos en la versión anterior. Realiza al menos las 2 que proponemos:

1. Intermediar un servidor WEB. Por ejemplo intermediar en el acceso al servidor de la UPV (seguimos utilizando como puerto local de atención de peticiones el puerto 8000)

2. Intermediar en el acceso entre `netClientLoad` y `netServerLoad`.

    - Si `ProxyConf` se ejecuta en la misma máquina que `netServerLoad` y tenemos una colisión en el uso de puertos -> modifica el código de `netServerLoad` para que use otro puerto. También puedes modificar el código de `netServerLoad` para que reciba como argumento el puerto del que debe escuchar.
    - 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

3. Proxy programable:

Utilizando como base el proxy configurable, creamos el proxy programable (`ProxyProg.js`)

Hemos de implementar el código del programador (`programador.js`) y hacer algunos cambios al código del nuevo proxy. Para implementar el programador podemos tomar como base el programa cliente `netClientLoad` y hacer las modificaciones necesarias. Por su parte para hacer que el nuevo Proxy programable reciba peticiones del programador, podemos tomar parte del código del servidor que tenemos realizado (`netServerLoad`)

Hemos de completar ambos programas para que se contemplen los siguientes aspectos:

- El programador debe recibir en línea de órdenes la dirección IP del proxy, y los nuevos valores de IP y puerto correspondientes al servidor. Con la IP del proxy y el

puerto por defecto del proxy (puerto 8001), contactará con el proxy para enviarle los datos del servidor remoto.

- El programador codificará los valores y los remitirá como mensaje al proxy, tras lo cual termina. Por su parte, el proxy recibirá este mensaje para actualizar la dirección del servidor al que contactará a partir de ese momento.

- 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, dos servidores WEB diferentes e ir comprobando como el programador modifica el servidor destino. También podrías emplear 2 servidores `netServerLoad` y 1 cliente `netClientLoad`, más el programador.

Cuestiones.

1. ¿Qué ventajas observas en la interacción entre un cliente y un servidor al emplear JSON respecto a no emplearlo?

2. Imagina un nuevo proxy que dispone de un pool de servidores a los que enviar peticiones. ¿Cómo podríamos lograr que este nuevo proxy se encargue de enviar peticiones al servidor que esté menos cargado en cada momento? Razona cómo desarrollarías este nuevo proxy.

REFERENCIAS

1. `Laboratorio/Práctica 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/Práctica 1, primera sesión`, en la zona de recursos de la asignatura , destacando los ejemplos empleados en esta sesión de prácticas: