TSR - PRÁCTICA 1 CURSO 2015/16

INTRODUCCIÓN AL LABORATORIO Y NODE.JS

PROXY INVERSO TCP/IP

Esta práctica consta de cuatro sesiones y tiene los siguientes objetivos:

1. Introducir los procedimientos y las herramientas necesarias para el trabajo en el laboratorio de TSR
    - Identificación y configuración de los recursos con los que cuentan los laboratorios
    - Aprender a realizar una aplicación mínima, con todas las operaciones necesarias que permitan ponerla en funcionamiento en el laboratorio
2. Utilizar la programación asíncrona para el manejo de peticiones en un servidor (entorno node.js, programación por callbacks, uso de promesas)
3. Desarrollar una solución para clientes remotos basada en el concepto de Proxy Inverso

El boletín se organiza en tres apartados, cada uno de los cuales cubre el correspondiente objetivo. Junto a cada apartado aparece una ordenación temporal orientativa (sesión o sesiones en las que completar dicho apartado):

1. Introducción al laboratorio de TSR (software disponible, procedimiento de trabajo). (sesión 1)
2. Primeros pasos en Node.js (sesiones 1 y 2)
3. Desarrollo de un proxy TCP/IP inverso (sesiones 3 y 4)

Cláusula de buen uso

Los recursos que se ponen a disposición del alumno se justifican por su carácter educativo, adecuado para la titulación en que se están formando. La utilización de recursos tan genéricos como máquinas virtuales, conectados a Internet (con limitaciones) y funcionando ininterrumpidamente es responsabilidad del alumno, quien adquiere el compromiso, entre otros, de velar por la confidencialidad de sus claves de acceso.

Se entiende como mal uso **el intento de interferir** en la actividad de los demás compañeros, accediendo a sus máquinas virtuales o afectando a su trabajo.

Cualquier uso que se desvíe del propósito para el que fueron concebidos estos recursos será sancionable. Para poder hacerlo se establecen mecanismos de seguimiento en las instalaciones que monitorizan las informaciones necesarias, y que serían aportadas como prueba ante el instructor de la sanción.

CONTENIDO

1 INTRODUCCIÓN AL LABORATORIO DE TSR

Las actividades de laboratorio se desarrollan sobre una distribución basada en LINUX (CentOS 7 de 64 bits)[1].

Puede accederse remotamente a un entorno similar al del laboratorio mediante el sistema virtualizado del DSIC: **evir** (http://evir.dsic.upv.es/, opción LINUX), pero se desaconseja su uso porque **no es compatible** con el uso de **los puertos** en node.js[2].

En los **equipos de escritorio** está instalado el software de desarrollo necesario

1. Editor de textos: podemos utilizar cualquiera de los instalados (ej. gedit, geany, etc.). Una buena alternativa (gratuita y disponible en todas las plataformas) es Visual Code
1. Entorno node.js
2. Bibliotecas auxiliares básicas (http, net, bluebird, ...)
3. Gestor de paquetes npm: permite instalar cualquier otra biblioteca que requiera nuestra aplicación

Existen restricciones en el uso de puertos (sólo queda libre para el usuario el rango 8000 a 8010), y se manejan direcciones virtuales → la comunicación entre equipos de laboratorio puede ser compleja.

Node.js está preinstalado en los equipos de laboratorio. Podemos ejecutar:

1. **`node`** para acceder al shell de node.js: abre indicador ">" y permite escribir código que se evalúa al pulsar return, tras lo cual muestra el resultado. Resulta muy útil para experimentar, depurar código, etc.
   - Podemos utilizar TAB para autocompletar orden
   - Las teclas up/down permiten desplazarse en la historia de órdenes
   - La variable especial _ representa el resultado anterior
   - NOTA.- **`var id = expr`** guarda el resultado de la expresión en **id** pero no devuelve nada, mientras que **`id = expr`** guarda resultado en id y devuelve dicho resultado
2. **`node fich.js`** para ejecutar el código del fichero fich.js en el entorno node

El primer paso es un test inicial de la instalación. Para ello vamos a escribir el código de un servidor web que se limita a saludar al cliente que contacta con él. Los pasos a seguir son:

1. Escribimos el servidor web en javascript

[1] El boletín se puede desarrollar en otros entornos (Windows o MacOS) y otras variantes de Linux. El alumno puede encontrar la información necesaria en https://nodejs.org/

[2] Ocasionalmente las sesiones **evir** interfieren entre sí cuando sus procesos coinciden en los puertos que desea emplear la aplicación

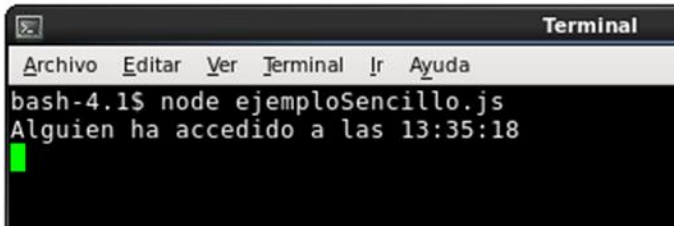
- Escribimos un fichero con el siguiente código (los comentarios en la parte derecha no forman parte del código, y sólo se incluyen para entenderlo mejor)

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

`var 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 puerto 8000

2. Lo ejecutamos en el entorno node.js
3. Utilizamos un navegador web como cliente
   - Accedemos a la URL **`http://localhost:8000`**
   - Comprobamos en el navegador la respuesta del servidor, y en la consola el mensaje escrito por el servidor

Node y HTTP



Recomendaciones:

1. Evita "copiar y pegar" los programas desde el documento PDF, ya que podemos insertar símbolos inadecuados que introduzcan errores sintácticos difíciles de depurar. Los programas son cortos, por lo que se aconseja teclearlos de nuevo
2. Es aconsejable utilizar un directorio específico para esta práctica, y crear dentro del mismo los subdirectorios que se consideren necesarios para las distintas actividades que plantea el boletín

2 PRIMEROS PASOS EN NODE.JS

Repasamos mediante algunos ejemplos sencillos los aspectos básicos de JavaScript y Node.js introducidos en clase. De ese modo se pueden probar empíricamente dichas herramientas, y posteriormente las utilizaremos para crear aplicaciones propias.

2.1.1 Posibles fuentes de error

Durante el desarrollo de las prácticas pueden aparecer distintos tipos de errores. A continuación describimos los casos comunes y cómo abordar la solución

- Errores sintácticos. Se detectan al interpretar el código. Se indica tipo de error y su ubicación en el código

```
> while 2<3 {console.log("...")}
SyntaxError: Unexpected number
    at Object.exports.createScript (vm.js:44:10)
    at REPLServer.defaultEval (repl.js:117:23)
    at bound (domain.js:254:14)
    at REPLServer.runBound [as eval] (domain.js:267:12)
    at REPLServer.<anonymous> (repl.js:279:12)
    at REPLServer.emit (events.js:107:17)
    at REPLServer.Interface._onLine (readline.js:214:10)
    at REPLServer.Interface._line (readline.js:553:8)
    at REPLServer.Interface._ttyWrite (readline.js:830:14)
    at ReadStream.onkeypress (readline.js:109:10)
```

- Errores lógicos (errores de programación).- Ej. intentar acceder a una propiedad de 'undefined', invocar una función asíncrona sin indicar callback, pasar un string cuando se esperaba un objeto, intentar acceder a una propiedad inexistente, problemas con datos introducidos por el usuario (ej. formato de fecha incorrecto), etc. Deben resolverse modificando el código (ej. verificar siempre las restricciones a aplicar sobre los argumentos a 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)
   at REPLServer.<anonymous> (repl.js:279:12)
   at REPLServer.emit (events.js:107:17)
   at REPLServer.Interface._onLine (readline.js:214:10)
   at REPLServer.Interface._line (readline.js:553:8)
   at REPLServer.Interface._ttyWrite (readline.js:830:14)
```

NOTA.- al tratarse de un lenguaje dinámico sin una orientación a tipos fuerte, parte de los errores que en otros lenguajes (ej. Java) captura el compilador, sólo aparecen aquí durante la ejecución

- Errores operacionales.- corresponden a problemas detectados en ejecución a pesar de que el código del programa es correcto (son parte del funcionamiento normal del programa). Pueden deberse al entorno (ej. nos quedamos sin memoria, demasiados ficheros abiertos), 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), ...
  Javascript dispone de construcciones try, catch, throw, con un significado similar al que tienen en Java
  El programador puede diseñar varias estrategias de solución en el código:
  - Cuando está claro cómo resolver el error, gestionar directamente dicho error (ej. ante un error al abrir un fichero para escritura, quizá sea adecuado crearlo como un fichero nuevo)
  - Cuando la gestión del error no es responsabilidad de ese fragmento de programa, propagar el error a su cliente (a quien lo invoca)
  - Para errores que pueden ser transitorios (ej. problemas con la red), reintentar la operación
  - Si no podemos gestionar ni propagar el error, e impide continuar con el programa, abortar
  - En otro caso, simplemente anotar el error (ej. en un log)

Para evitar en lo posible los distintos tipos de errores, se recomienda documentar correctamente cada función de interfaz

- los argumentos: significado y tipo de cada uno, así como cualquier restricción adicional
- qué tipo de errores operaciones pueden aparecer, y cómo se van a gestionar
- el valor de retorno

y utilizar el paquete **assert** en el código (con las operaciones básicas **equal**(expr1,expr2, mensajeError) y **ok**(exprLógica, mensajeError)

Ejemplo

```
/*
 * Make a TCP connection to the given IPv4 address.  Arguments:
 *    ip4addr        a string representing a valid IPv4 address
 *    tcpPort        a positive integer representing a valid TCP port
 *    timeout        a positive integer denoting the number of milliseconds
 *                   to wait for a response from the remote server before
 *                   considering the connection to have failed.
 *    callback       invoked when the connection succeeds or fails.  Upon
 *                   success, callback is invoked as callback(null, socket),
 *                   where `socket` is a Node net.Socket object.  Upon failure,
 *                   callback is invoked as callback(err) instead.
 *
 * This function may fail for several reasons:
```

```
 *    SystemError    For "connection refused" and "host unreachable" and other
 *                   errors returned by the connect(2) system call.  For these
 *                   errors, err.errno will be set to the actual errno symbolic
 *                   name.
 *    TimeoutError   Emitted if "timeout" milliseconds elapse without
 *                   successfully completing the connection.
 *
 * All errors will have the conventional "remoteIp" and "remotePort" properties.
 * After any error, any socket that was created will be closed.
 */
function connect(ip4addr, tcpPort, timeout, callback) {
  assert.equal(typeof (ip4addr), 'string', "argument 'ip4addr' must be a string");
  assert.ok(net.isIPv4(ip4addr), "argument 'ip4addr' must be a valid IPv4 address");
  assert.equal(typeof (tcpPort), 'number', "argument 'tcpPort' must be a number");
  assert.ok(!isNaN(tcpPort) && tcpPort > 0 && tcpPort < 65536,
      "argument 'tcpPort' must be a positive integer between 1 and 65535");
  assert.equal(typeof (timeout), 'number', "argument 'timeout' must be a number");
  assert.ok(!isNaN(timeout) && timeout>0, "argument 'timeout' must be a positive int");
  assert.equal(typeof (callback), 'function');
  /* do work */
}
```

2.1.2 Acceso a ficheros

Todos los métodos correspondientes a operaciones sobre ficheros aparecen en el módulo 'fs'.

Las operaciones son asíncronas, pero para cada función asíncrona **xx** suele existir la variante síncrona **xxSync**

- Leer el contenido de un fichero

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

- Escribir contenido en un fichero

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

- Acceso a directorios

  El siguiente ejemplo obtiene todos los ficheros contenidos en el directorio indicado o en alguno de sus subdirectorios

```
var fs = require('fs')
```

```
function getFiles (dir, files_){
    files_ = files_ || [];
    var files = fs.readdirSync(dir);
    for (var i in files){
        var name = dir + '/' + files[i];
        if (fs.statSync(name).isDirectory()){
            getFiles(name, files_);
        } else {
            files_.push(name);
        }
    }
    return files_;
}

console.log(getFiles('.')) // directorio donde buscar (ej el actual)
```

- Módulo path

  Contiene varias funciones que simplifican la manipulación de rutas (nombres de ficheros o directorios, que pueden contener el separador **/** y los nombres especiales **.** y **..)**

  - normalize. A partir de una tira representando el path, interpreta los separadores y los nombres especiales, y devuelve una nueva tira que corresponde a esa misma ruta normalizada

    ```
    > var path = require('path');
    > path.normalize('/a/.///b/d/../c/')
    '/a/b/c'
    ```

  - join. A partir de una lista variable de argumentos, los une y normaliza el path resultante, devolviendo la tira que corresponde al path normalizado

    ```
    > var path = require('path');
    > var url = '/index.html';
    > path.join(process.cwd(), 'static', url);
    '/home/nico/static/index.html'
    ```

  - basename, extname y dirname. Permiten extraer los distintos componentes de un path

    ```
    > var path = require('path')
    > var a = '/a/b/c.html'
    > path.basename(a)
    'c.html'
    > path.extname(a)
    '.html'
    > path.dirname(a)
    '/a/b'
    ```

  - exists. Permite comprobar la existencia o no de un path concreto

    ```
    > var path = require('path')
    > path.exists('/etc', function(exists){
       console.log("Does the file exist?", exists)})
    > Does the file exist? true
    ```

2.1.3 JSON (JavaScript Object Notation)

Es un formato para representar datos que se ha convertido en estandard de facto para la web. Permite representar listas de valores y objetos (diccionarios con pares clave/valor):

- Una lista se representa como [a,b,c]
- Un objeto se representa como {"k1":v1, "k2":v2, ..}
  - Todas las propiedades de un objeto deben aparecer entre ""
  - Los valores de un objeto pueden ser tiras, nums, logico, objeto, array o null, pero NO funciones, Date, etc
- Las tiras pueden contener cualquier caracter unicode
- Los numeros siempre en base 10

Las operaciones fundamentales son:

- JSON.parse(string) construye un objeto javascript a partir de su representación JSON
- JSON.stringify(obj) construye la representación JSON de un objeto javascript

```
function PoligonoRegular (lados,longitud) { // constructor de la clase
   this.numLados     = lados;
   this.longitudLado = longitud;
}

PoligonoRegular.prototype.perimetro = function() { // metodo de la clase
   return this.numLados * this.longitudLado;
}

function Cuadrado(longitud) { // constructor de la clase
   this.numLados     = 4;
   this.longitudLado = longitud;
}

Cuadrado.prototype = Object.create(PoligonoRegular.prototype); // herencia
Cuadrado.prototype.constructor = Cuadrado;

var c = new Cuadrado(6); // creacion instancia

Cuadrado.prototype.superficie = function() { // metodo de la clase
   return this.longitudLado * this.longitudLado;
}

console.log('el perimetro de un cuadrado de lado 6 es '+ c.perimetro());
console.log('y su superficie es '+c.superficie());

console.log(JSON.stringify(c)); // estado del objeto c como string
```

2.1.4 Programación asíncrona: eventos

single.js

```
function fib(n) {return (n<2)? 1: fib(n-2)+fib(n-1);}

console.log("iniciando ejecucion...");

setTimeout( //espera 10 seg y ejecuta la funcion
   function() {console.log('M1: Quiero escribir esto...');},
   10);
```

	`var j = fib(40); // esta invocacion cuesta mas de 1seg`
	
	`function otherMsg(m,u) {console.log(m + ": El resultado es "+u);}`
	
	`otherMsg("M2",j); //M2 se escribe antes que M1 porque el hilo principal rara vez se suspende`
	
	`setTimeout( // M3 se escribe tras M1`
	`  function() {otherMsg('M3',j);},`
	`  1);`

eventSimple.js	`var ev = require('events');`
	`var emitter = new ev.EventEmitter;`
	`var e1 = "print", e2= "read";  // name of events`
	`var num1 = 0, num2 = 0;  // auxiliary vars`
	
	`// register listener functions on the event emitter`
	`emitter.on(e1,`
	`  function() {console.log('event '+e1+' has happened '+num1+' times')})`
	`emitter.on(e2,`
	`  function() {console.log('event '+e2+' has happened '+num2+' times')})`
	`emitter.on(e1, // more than one listener for the same event is possible`
	`  function() {console.log('something has been printed!')})`
	
	`// generate the events periodically`
	`setInterval(`
	`  function() {emitter.emit(e1);}, // generates e1`
	`  2000);  // every 2 seconds`
	`setInterval(`
	`  function() {emitter.emit(e2);}, // generates e2`
	`  8000);  // every 8 seconds`

2.1.5 Interacción cliente/servidor: módulo net

Cliente (netClient.js)	`var net = require('net');`
	
	`var 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)	`var net = require('net');`
	
	`var server = net.createServer(`

```
    function(c) { //connection listener
        console.log('server connected');
        c.on('end',
            function() {
                console.log('server 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');
    });
```

2.1.6 Encadenamiento de promesas

```
var promise = require('bluebird'),
    fs      = require('fs');

var readFileSync = promise.promisify(fs.readFile); //synchonous variant
var f = readFileSync('netServer.js','utf-8');

function printFile (data) {console.log(data);        return data;} // aux function
function showLength(data) {console.log(data.length); return data;} // aux function
function showErrors(err)  {console.log('Error reading file ..'+err);}

f.then(printFile).then(showLength,showErrors);
```

2.1.7 Acceso a argumentos en línea de órdenes

El shell recoge todos los argumentos en línea de órdenes y se los pasa a la aplicación javascript empaquetados en un vector denominado **process.argv** (abreviatura de 'argument values').

Define un fichero args.js con el siguiente código

```
console.log(process.argv);
```

E invócalo con distintos argumentos. Por ejemplo:

```
node args.js uno dos tres cuatro
```

Puedes observar que los dos primeros elementos del vector corresponden a 'node' y el path del programa ejecutado, respectivamente. Es frecuente aplicar process.argv.slice(2) para descartar dichos elementos del vector (de forma que sólo quedan los argumentos reales para nuestra aplicación).

2.1.8 Consultar la carga de un equipo

A partir de **`netClient`** y **`netServer`** creamos un par de aplicaciones **`netClientLoad`** y **`netServerLoad`** con las siguientes características:

- Emplean el puerto 8000
- **`netClientLoad`** aporta su IP como cuerpo del mensaje. Dicha IP deberá ser introducida como parámetro al invocar el programa
  - Hay que asegurarse de que el proceso finaliza (p.ej. con **`process.exit()`**)
  - En los equipos del laboratorio, pero no en EVIR, se puede averiguar de una forma *indirecta* (preguntarle quiénes somos a alguien "de fuera"). Lo tenéis en el directorio de la asignatura como **`averiguar_ip.sh`**

```
wget -qO - http://memex.dsic.upv.es/cgi-bin/ip
```

- **`netServerLoad`** contesta con su IP y un número que representa su carga de trabajo actual, calculada a partir de **`/proc/loadavg`** de la siguiente forma:

```
var fs = require('fs');
function getLoad(){
  data=fs.readFileSync("/proc/loadavg");
  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;
};
```

Da un peso 10 a la carga del último minuto, un peso 2 a la de los últimos 5 minutos, y un peso 1 a la de los últimos 15. A cada ítem le suma una centésima para evitar la confusión entre el valor 0 y un error; al fin y al cabo, cuando lo apliquemos, lo interesante será el valor relativo, no el absoluto.

Por tanto, **`netServerLoad`** no requiere parámetros de entrada, y **`netClientLoad`** tendrá dos argumentos de entrada que se pasarán a través de la línea de órdenes:

1. Dirección IP remota (Dirección IP del servidor).
2. Dirección IP local (Dirección IP desde donde se ejecuta el cliente).

Completa ambos programas, colócalos en equipos diferentes y haz que se comuniquen mediante el puerto 8000, de manera que **`netServerLoad`** emita un mensaje por cada petición recibida con IP del cliente, y **`netClientLoad`** muestre la carga devuelta como resultado de la última petición.

Recuerda que para obtener los argumentos de la línea de órdenes en un programa JavaScript puedes utilizar las siguientes instrucciones:

- **`process.argv.length`** : permite obtener el número de argumentos pasados por la línea de órdenes, incluyendo el nombre del programa.

- **`process.argv[2]`** : permite obtener el primer argumento de entrada si hemos invocado mediante "**`node programa arg1 …`**". **`process.argv[0]`** contiene la cadena '**`node`**' y **`process.argv[1]`** contiene la cadena '**`programa`**'.
    - Muy importante el módulo net para trabajar con sockets

```
var net = require('net');
  net.createServer(function (socket) {
  console.log('socket connected!');
  // etcétera ...
}).listen(8000)
```

    - A modo de referencia, se ha dejado una instancia en marcha en el puerto 9000 de **`tsr1.dsic.upv.es`**

2.1.9 Intermediario entre 2 equipos (proxy transparente)

Queremos construir un servicio que actúa como intermediario: al ser invocado devuelve como resultado el que se produciría al invocar a otro, que puede residir incluso en otro equipo. 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 este sistema intervienen 4 componentes:

1. El **invocante** del servicio, que actúa como cliente y que emite las peticiones. Podemos utilizar un **invocante externo** específico (ej. un navegador web que actúa como cliente HTTP).
1. El **intermediario**, que recibe peticiones del cliente y las reencamina al destinatario final.
2. El destinatario final, que implementa realmente el **servicio**. Le es indiferente que las peticiones procedan directamente del primer actor (invocante) o del segundo (intermediario). Podemos utilizar un **servicio final externo**, como un servidor de web convencional (nos limitamos a protocolo HTTP)
2. El **protocolo**, que funciona como especificación de los mensajes entre invocante y servicio, tanto en su forma (sintaxis) como en su significado (semántica).

El intermediario no altera el contenido del mensaje más allá de las propiedades aplicables a TCP/IP (es decir, puede modificar puertos y direcciones). Por lo demás actúa como un transporte *neutro* que desconoce el protocolo establecido entre invocante y servicio.

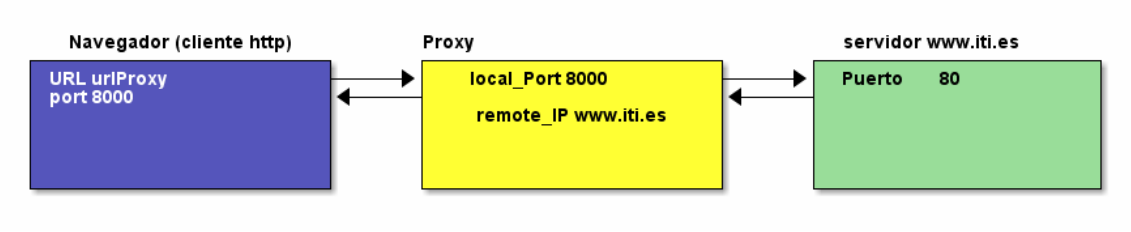
- En resumen, mientras se trate de un protocolo basado en TCP/IP, el intermediario ha de funcionar

Como primer experimento creamos un intermediario que, empleado por un cliente HTTP, devuelva los documentos de un servidor de web preestablecido. El programa **`myTcpProxy.js`** actúa como un servidor basado en el módulo net que:

1. Recibe peticiones en el puerto 8000
2. Toma la petición y construye otra cambiando destino y puerto

- Como servicio a invocar, tomaremos el puerto 80 del servidor web del Instituto Tecnológico de Informática de la UPV (ITI) (IP 158.42.156.2)

3. Devuelve la respuesta del servidor al invocante



Este proxy "transparente" es *visible* para el cliente (al fin y al cabo necesita contactar con él). Se comporta como servidor para recibir peticiones de su cliente, y como cliente para enviar las peticiones al servidor web real.

Código (myTcpProxy.js)

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

var LOCAL_PORT  = 8000;
var LOCAL_IP  = '127.0.0.1';
var REMOTE_PORT = 80;
var REMOTE_IP = '158.42.156.2'; // www.iti.es

var server = net.createServer(function (socket) {
        socket.on('data', function (msg) {
            var serviceSocket = new net.Socket();
            serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
                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);
```

Emplea para ello dos sockets (socket para comunicar con el cliente y serviceSocket para dialogar con el servidor):

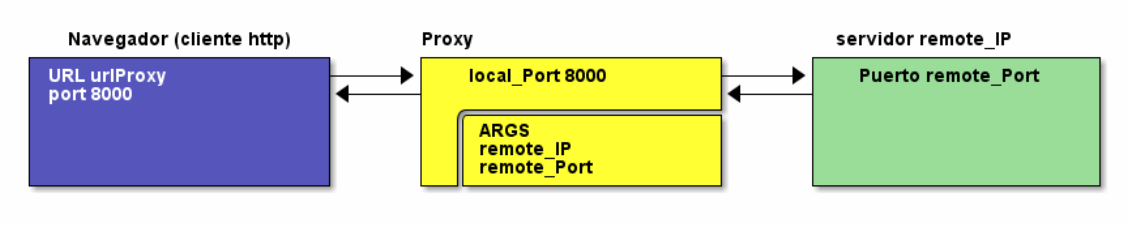
1. lee un mensaje (`msg`) del cliente
2. abre una conexión con el servidor
3. escribe una copia del mensaje
4. espera la respuesta del servidor
5. y devuelve una copia al cliente

Comprueba qué funcionamiento es correcto usando un navegador web que apunte a http://direccion_del_proxy:8000/, y recuerda que se puede obtener la IP del proxy con **`averiguar_ip.sh`**

Advertencia: Es posible que al ejecutar un programa en node aparezca el error "EADDRINUSE". Normalmente se debe a que estamos referenciando un puerto que está siendo utilizado por

otro programa node (ej. porque algún programa node anterior ha quedado en ejecución fuera de nuestro control). Esto se puede determinar ejecutando el comando '**`ps -all`**' en la línea de órdenes de Linux. En caso que aparezcan programas node en ejecución podemos finalizarlos utilizando el comando '**`pkill -f node`**'. Este comando finaliza todos los programas node que se están ejecutando con nuestro identificador de usuario.

Modifica el código del proxy para que reciba como parámetros de entrada la dirección del servidor remoto (dirección IP y puerto). De esta forma **`myTcpProxy`** podrá ser usado de una forma más flexible.

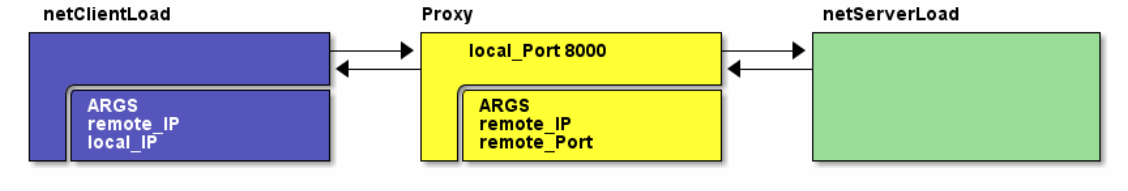


Ahora **`myTcpProxy`** tendrá los siguientes **argumentos de entrada** (en ese orden):

1. Dirección IP remota (Dirección IP del servidor remoto).
2. Puerto remoto (Puerto de recepción de peticiones del servidor remoto).

Utiliza esta nueva versión de `myTcpProxy` para acceder de nuevo al servidor de**l ITI**, como en el ejemplo anterior. Debes seguir utilizando como puerto local de atención de peticiones el puerto 8000.

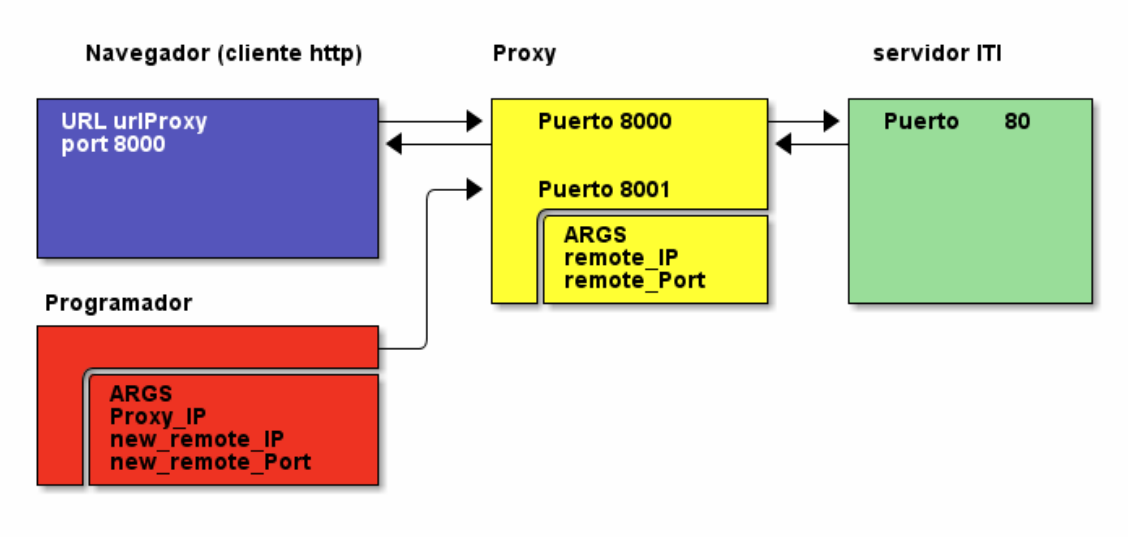
Ahora **`myTcpProxy`** nos permite acceder a cualquier servidor remoto. **Utilízalo, invocándolo con los parámetros adecuados,** de forma que **`netClientLoad`** y **`netServerLoad`** puedan comunicarse mediante este proxy.



En este caso **`netClientLoad`** recibirá como segundo parámetro la dirección IP del proxy en vez de la dirección IP de **`netServerLoad`**.

Nota: Si el proxy se ejecuta en la misma máquina que **`netServerLoad`** es necesario cambiar el puerto de atención del servidor de carga a uno diferente de 8000, ya que este es el puerto en el que atiende el proxy).

Crea una variante del proxy llamada `newTcpProxy.js` que utilice un **puerto de programación** al que se puedan enviar un par de valores (IP, puerto) para reprogramar el servidor remoto.



En este caso:

- Las peticiones al puerto 8001 (puerto de programación del proxy) enviarían el valor del nuevo servidor y puerto remoto.
- Las peticiones al puerto 8000 (puerto de atención de peticiones del proxy) se atienden con la redirección vigente del servidor remoto.

`newTcpProxy` tendrá los mismos **argumentos de entrada** que **`myTcpProxy`**, con el fin de proporcionar la dirección del servidor remoto inicial.

1. Dirección IP remota (Dirección IP del servidor remoto inicial).
2. Puerto remoto (Puerto de recepción de peticiones del servidor remoto inicial).

Nota: el puerto de atención de peticiones del proxy sigue siendo 8000, y el puerto de programación será el 8001. Estos dos valores no se pasan como parámetros de entrada del proxy, ya que su valor es fijo!!.

Necesitarás un programa cliente (**`programador.js`**) que envíe dichos valores (*programe* el proxy). Este programa tendrá los siguientes **argumentos de entrada**:

1. Dirección IP del proxy (dirección IP del proxy).
2. Dirección IP remota (dirección IP del nuevo servidor remoto).
3. Puerto remoto (puerto de recepción de peticiones del nuevo servidor remoto).

Nota: No se pasa como argumento el puerto de programación del proxy, ya que este debe ser siempre 8001. La dirección IP del proxy sí que se pasa ya que nos permite utilizar el programador desde una máquina no local al proxy.

El **`programador`** debería enviar mensajes con un contenido como el siguiente:

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

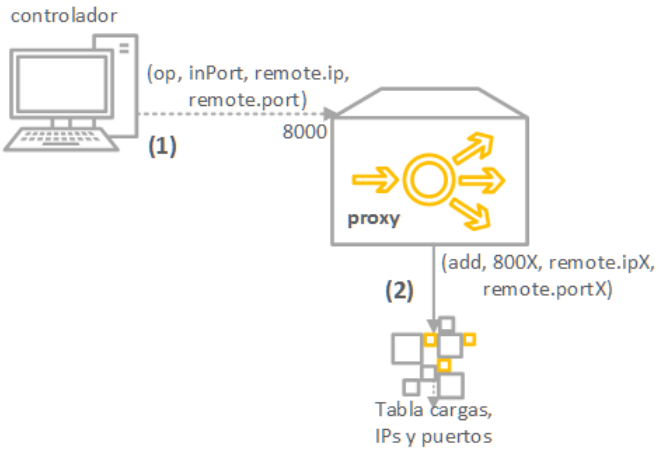
Para verificar el funcionamiento de **`newTcpProxy`** lo puedes probar con el par **`netServerLoad`**/**`netClientLoad`**, o bien con un cliente HTTP si el destino es un servidor de web.

3 APARTADO 3. APLICACIÓN FINAL: PROXY TCP/IP INVERSO

Un proxy inverso es un servidor que colocamos entre los clientes remotos y nuestros servidores web:

- Acepta las peticiones de los clientes
  - Oculta la topología de los servidores
  - Puede realizar autenticación, control de acceso, cifrado, etc.
- Redirige el tráfico hacia el servidor que corresponda
  - Permite equilibrar la carga de los servidores
  - Puede tolerar el fallo de un servidor
- Permite inspeccionar, transformar y encaminar el tráfico HTTP
  - Auditoría, logs, etc.

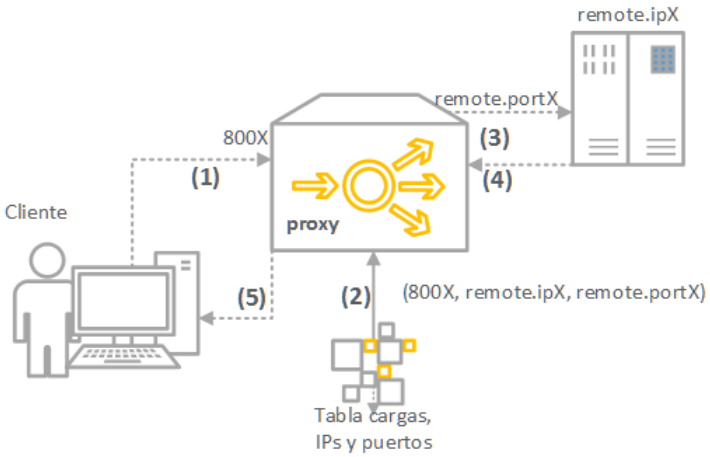
Disponemos de un número arbitrario de clientes que envían solicitudes de servicio a los puertos 8001 a 8008[3] de nuestro proxy.



Mediante el puerto 8000 (programa **`controlador`**), podemos programar al proxy para indicarle qué servicio viene representado por un puerto en concreto. P.ej., si el proxy recibe en su puerto 8000 este mensaje:

```
var msg =JSON.stringify ({'op':"set", 'inPort':8001, 'remote':{'ip':"158.42.4.23", 'port':80}})
```

… las peticiones que lleguen desde ese momento al puerto 8001 de nuestro proxy se servirán conectando con el puerto 80 de 158.42.4.23. Esto sería equivalente al proxy visto anteriormente (**`myTcpProxy.js`**).

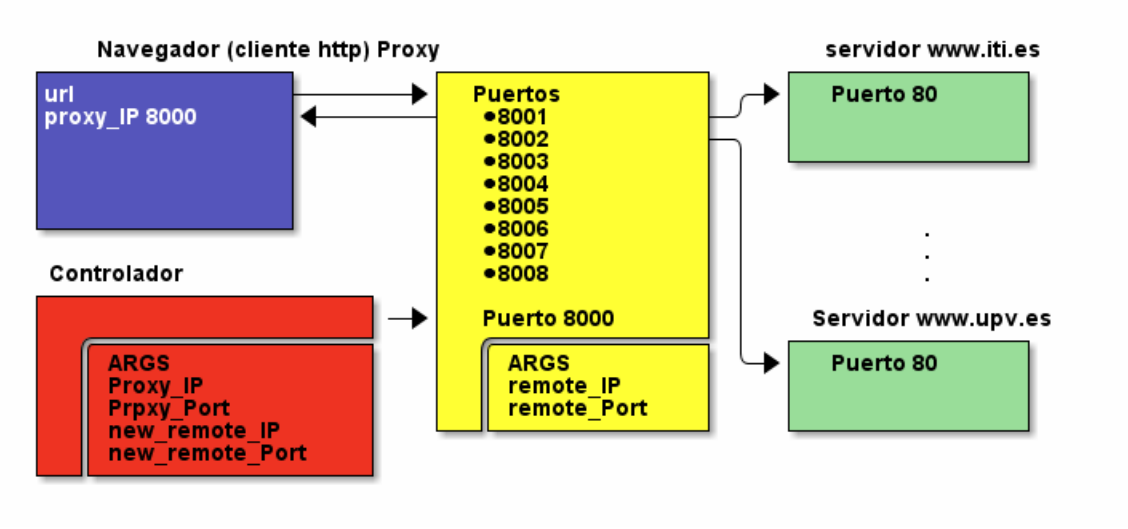


Debe desarrollarse un **programa capaz de comunicarse con el proxy para programarlo** a través del puerto de control, y un proxy que implemente la funcionalidad descrita.

- El programa se denominará **`controlador.js`**, este proxy se llamará **`proxy.js`**

[3] Recordamos que en el laboratorio podemos disponer de los puertos 8000 a 8010 para nuestros programas

`proxy` no tendrá argumentos de entrada, ya que los puertos de atención de peticiones siempre serán los puertos de 8001 a 8008, el puerto de programación siempre será el 8000, y las direcciones de los servidores remotos iniciales son valores constantes que se escriben dentro del programa. Las pruebas a verificar se basarán en los protocolos HTTP, HTTPS y FTP mediante un navegador de web.



Seguidamente se muestran algunos valores de prueba de estos servidores remotos, a título de ejemplo:

Puerto del proxy	IP del servicio remoto	Puerto del servicio remoto
8001	www.dsic.upv.es	80 (http)
8002	www.upv.es	80 (http)
8003	www.google.es	80 (https)
8004	www.iti.es	80 (http)
8005	www.disca.upv.es	80 (http)

Pero debe estar preparado para atender cualquier asociación válida.

Por su parte, **`controlador`** debe tener los siguientes **argumentos de entrada**:

1. Dirección IP del proxy
2. Puerto de entrada del proxy (Puerto de entrada que será reprogramado).
3. Nueva dirección IP remota.
4. Nuevo puerto remoto.

Nota: los argumentos 3 y 4 determinan la nueva redirección de la entrada al proxy especificada en el segundo argumento. Los valores válidos para este segundo argumento están en el rango 8001 a 8008. El puerto de programación no se pasa como argumento, suponemos que siempre es 8000.