TSR - LAB 1 COURSE 2015/16

INTRODUCTION TO THE LABS AND NODE.JS

REVERSE PROXY TCP/IP

This lab will be developed in four different sessions, with the following overall learning goals:

1. Become familiar with the procedures and tools needed to work in the TSR labs.
    - Identification and configuration of lab resources.
    - Learn to produce a minimal application, employing those operations needed to set it up in the lab.
2. Use Asynchronous programming to handle server requests by means of callbacks and Promises, getting familiar with the nodejs environment.
3. Develop a solution for remote clients, based on the Reverse Proxy concept.

This document consists of three sections, each one covering its corresponding goal, as put forward above. For each section, we provide a temptative temporal schedule (session or sessions that should be used to carry out the section's tasks):

1. Introduction to the TSR lab (available software, procedures) → session 1
2. First steps in Node.js → sessions 1 and 2
3. Development of a Reverse TCP/IP proxy → sessions 3 and 4

Fair use clause

The resources made available to the student are for the sole purpose of supporting the student's educational needs, as seen fit in the context of the studies he/she is carrying out. Usage of generic reosurces, like virtual machines connected to the internet (with suitable limitations), and continuously running is the responsibility of the student, who commits to maintain the confidentiality of his/her passwords and other private credential materials.

Any attempt to interfere with the activity of other students, accessing their virtual machines or otherwise hindering their work will be considered as a **bad usage** of the resources put at a student's disposal, and a violation of this clause.

Any usage deviating from the main purpose for which resources are made available to a student will be punishable. To detect deviations, we have set up suitable activity tracking mechanisms on those resources. Logs obtained thorugh such mechanisms shall and will be used to support proposed sanctions for violations of this clause.

CONTENTS

1 Introduction to TSR labs

Activities carried out in the lab facilities will use machines running the 64-bit Linux distribution CentOS 7[1].

Students can access a environment very close in functionality to that available at the labs via the DSIC virtualization service: **evir** (http://evir.dsic.upv.es/, LINUX option), however we advise strong caution on using it, as the evir environment poses restrictions on the **IP ports** that can be used, affecting some of the exercises to be carried out with node.js[2].

The **machines in the lab** have all the software needed to carry out the lab assignments**.**

1. Text editors: e.g. gedit, geany, etc.
1. **node.js** environment
2. Basic node packages (http, net, bluebird, ..)
3. The **npm** package manager: to install whatever other nodejs packages are deemed useful to complete the assignments.

Lab networking policies restrict usage of ports to the range 8000 to 8010. In addition, communication among lab machines is made difficult due to the fact that students are really using a virtualized environment, not the actual desktop machines to run the software of the assignments.

So, we can run the following software:

1. **`node`** gives us access to node.js repl shell: you get the prompt ">" and you can write nodejs code that gets evaluated as soon as you hit the "Enter" key. This is a very useful tool to experiment with node and its language (javascript).
   - **`TAB`** can be used to autocomplete a command
   - **`up/down`** keys allow us to navigate our command history
   - Special variable _ lets us recover the last result evaluated
   - NOTE.- **`var id = expr`** stores the result ox **`expr`** within **`id`**, but does not return any result, thus preserving the previous value of _ . On the other hand, **`id = expr`** is itself an expression, returning the value of **`expr`**.
2. **`node fich.js`** runs the code of module (file) fich.js under node's runtime

We will first carry out a generic test of the installation. Let us proceed by writing code for a web server whose only function is to provide a fixed text answer to any client contacting it. The steps you should follow are:

1. Write the web server in javascript

[1] This work can be developed in another environments (Windows, MacOS, other Linux variants). You can find necessary setup information at https://nodejs.org/

[2] Eventually can be interferences between **evir** sessions when its processes must use the same ports that application uses

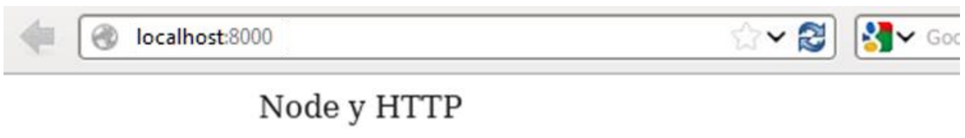
- Place the following code in that javascript module (ignore the comments in the right column, they only illustrate the purpose of the code)

Code (ejemploSencillo.js)	**comments**
`var http = require('http');`	**Imports module http**
`function dd(i) {return (i<10?"0":"")+i;}`	**dd(8) -> "08"** **dd(16) -> "16"**

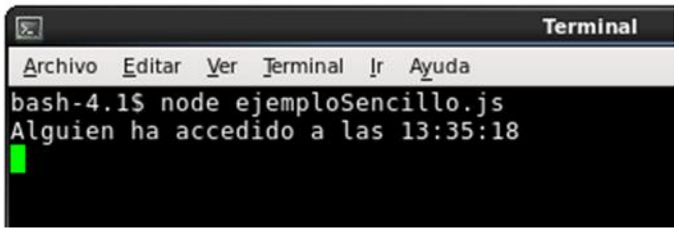
`var server = http.createServer(`
`   function (req,resp) {`
`      res.writeHead(200,{'Content-Type':'text/html'});`
`      res.end('<marquee>Node y Http</marquee>');`
`      var d = new Date();`
`      console.log('alguien ha accedido a las '+`
`         d.getHours() +":"+`
`         dd(d.getMinutes()) +":"+`
`        dd(d.getSeconds()));`
	creates the server, associating it with a function (the handler) retuning a fixed response. In addition it logs the current time in the console.

`}).listen(8000);`	**The server listens on port 8000**

2. Run it under node
3. Use a browser
   - Access URL **`http://localhost:8000`**
   - Verify in the browser the server's answer. Also, verify the server writes the expected console message



Node y HTTP



```
bash-4.1$ node ejemploSencillo.js
Alguien ha accedido a las 13:35:18
```

Two notes of advice:

1. Avoid ***copy&paste*** from the PDF doc: you may not actually be inserting the proper characters into the program file. The code is short enough: just type it in.
2. Be organized: It will save you some pain if you place the contents of this lab within its own directory (name it appropriately). Then do the same for each exercise: create a directory within the lab's directory to hold the material you produce following the instructions of this document for the different tasks

2 First steps in node.js

We will make use of some simple exercises to review some of the basic aspects of JavaScript, and the nodejs environment, as seen in the seminar sessions. This will give us some phands-on experience in the usage of the tools we will need later on to produce more complex code.

2.1.1 Posibles fuentes de error

During your lab work different error types can be detected. The common cases are described bellow

- Syntax errors.- They are detected when interpreting the code. Message indicates the kind of error and its location in the code

```
> 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)
```

- Logical errors (programming errors).- E.g. Trying to access to a property of 'undefined', trying to call a synchronous function without callback, passing a string when an object is expected, trying to access an invalid property, problems with user enterd data (e.g. Incorrect date format), etc.
  Code must be modified (e.g. you must validate the constraints over function args)

```
> 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)
```

NOTE.- Javascript is a dynamic and weakly-typed language. Strongly-typed languages (e.g. Java) can refuse to compile if an expression does not closely match the expected type; same code in javascript may produce unpredictable results or may perform implicit type conversion

- Operational Errors.- represent run-time problems experienced by correctly-written programs (these are not bugs in the program). These are usually problems with the system itself (e.g. out of memory, too many files opened), the system's configuration (e.g. no route to remote host), the network (e.g. socket hang-up), or a remote service (ej. failure to connect), ...
  Javascript has **try**, **catch**, and **throw** sentences, with familiar meaning (similar to Java)
  For any given error, code must include some solution strategy. The alternatives are:
  - Deal with failure directly.- when it's clear what you have to do to handle an error (e.g. error trying to open a log file; at first access you just need to create that log file)
  - Propagate the failure to your client.- when you don't know how to deal with the error, the simplest thing to do is to abort operation, clean up, and deliver a error back to the client
  - Retry the operation.- errors from the network and remote services may be transitory. I's useful to retry in a few seconds
  - Abort.- if we can't handle of propagate the error, it's not a transitory error, and we can't ignore it
  - Ignore it.- when you can't deal with it, nor propagate or retry, and threre's also no reason to crash the program. E.g. you are keepeng track of a group or remote services using DNS and one of those services fails otr of DNS; there's nothing you can do about it except log a message and proceed with the remaining services

To minimize errors is advisable to document every interface function

- parameters: meaning, legal values, and additional constraints of every one
- possible operational errors and its handling
- return value

and to use **assert** package (its main operations are **equal**(expr1,expr2, errorMsg) and **ok**(logicalExpr, ErrorMsg)

Example

```
/*
 * 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 File access

All methods provided by nodejs to act on files can be accessed thorugh the 'fs' module. Most methods are asynchronous

- Read a file’s content into memory

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

- Write memory contents into a file

```
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');
});
```

- Search for files and directories
  This example shows all files contained in the indicated directory or its subdirectories

```
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('.'))  // e.g. find from actual directory
```

- The “path” module allows us to manipulate file path names
  - normalize. From a string representing a path, deals with separators and special names and return a new string corresponding to the normalized path

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

  - join. From a list of arguments, joins it and normalize result, returning a normalized path string

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

  - basename, extname y dirname. To access to the components of a 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. Checks for existence of a path

```
> 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)

Text format to represent data. It has become a de-facto standard for data transmission in web application. It can represent (almost) any Javascript data types, in particular: objects (key-value pairs) and lists of values (including other objects or lists):

- Lists are simply represented like so: `[a,b,c]`
- An object is represented thus: `{"k1":v1, "k2":v2, ..}`

   - Notice the quotes around the key names: All properties of an object must appear surrounded by double quotes (")
- Values in objects and arrays can be strings, numbers, booleans, objects, arrays or null. However, NO functions, Date, etc are allowed
- Strings are Unicode-based
- Numbers are always represented in decimal notation

Basic operations on JSON are:

- `JSON.parse`(string) builds a JavaScript object from its JSON representation
- `JSON.stringify`(obj) produces the JSON representation of a JavaScript object

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

PoligonoRegular.prototype.perimetro = function() { // class method
   return this.numLados * this.longitudLado;
}

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

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

var c = new Cuadrado(6); // instance creation

Cuadrado.prototype.superficie = function() { // class method
   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)); // state of object c as a string
```

2.1.4 Asynchronous programming: events

```
//------------------------------------ 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
```

```
//------------------------------------ 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() {
```

```
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);
```

```
        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 Client/server interaction: the “net” module

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(client connected');
      c.on('end',
         function() {
            console.log(client
disconnected');
         });
      c.write('Hello\r\n');
      c.pipe(c); //echo what is received
from incoming stream
   });

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

2.1.6 Promise chaining

```
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 Command Line arguments

The shell captures the arguments in the command line and passes it to the javascript application packed as a sequence called **process.argv**

Write the file 'arg.js' with that code

And call it several times with different arguments. E.g. node one two three

The first elem of process.argv (process.argv[0]) contains 'node'. The second (process.argv[1]) contains the path of the executed program. We usually call process.argv.slice(2) to discard those elements (so after that call process.argv contains only the real arguments for our program)

2.1.8 Query the load of a machine

We can reuse the code for `netClient` and `netServer` to create a pair of applications `netClientLoad` and `netServerLoad` satisfying the following requirements:

- Use port 8000
- `netClientLoad` sends its own IP in the message body. The IP must be provided as a parameter of the program invocation
  - The process must terminate (e.g. con `process.exit()`)
  - On the lab machines (but not in EVIR) it is possible to find the IP of the machine using indirect means: ask an outside server that you contact just for this purpose. You can find a script that does just this in the TSR folder, under the name `averiguar_ip.sh`. This is what it does:

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

- `netServerLoad` answers with its own IP and a number representing its current load, computed from the contents found at `/proc/loadavg` as follows:

```
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;
};
```

  This assigns a weight of 10 to last minute's load, a weight of 2 to the load of the last 5 minutes, and a weight of 1 to the load of the last 15 minutes. Each item gets an additional hundredth to avoid confusion between the value 0 and an error. After all, we are interest in the relative value of this quantity, not its absolute value.

Thus, `netServerLoad` does not require ant input parameters whereas `netClientLoad` requires the following two parameters that need to be provided in the command line:

1. Remote IP (Server's IP).
2. Local IP (IP of the machine executing the client's code).

Write both programs, deploy them to two different machines, and let them communicate through port 8000, so that **`netServerLoad`** writes a message for each received request, with the IP of the client, and **`netClientLoad`** writes the load as received in the last request made to the server.

Remember that you can access the arguments of a nodejs program as follows:

- **`Process.argv`**: array containing the command line used to run the nodejs program, separated into each one of the arguments used.
- **`process.argv.length`**: consequently contains the number of arguments passed through the command line, including the program name and the name of the executable (node).
- **`process.argv[2]`**: thus accesses the first "real" argument for our program. That is, if we have run our program like this "**`node programa arg1`** …". **`process.argv[0]`** holds string '**`node`**', and **`process.argv[1]`** holds string '**`programa`**'.
    - Remember: use the 'net' module to work with network sockets

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

    - For testing purposes we have left a live instance of the server listening on port 9000 of **`tsr1.dsic.upv.es`**

2.1.9 Transparent proxy between two machines

We want to build a service acting as a proxy: invoking it results in an invocation of the proxied service, returning the result of that proxied service to the client. HTTP proxies are an example of this kind of behavior. Have a look at http://en.wikipedia.org/wiki/Proxy_server for a more comprehensive description.

There are 4 components in this system:

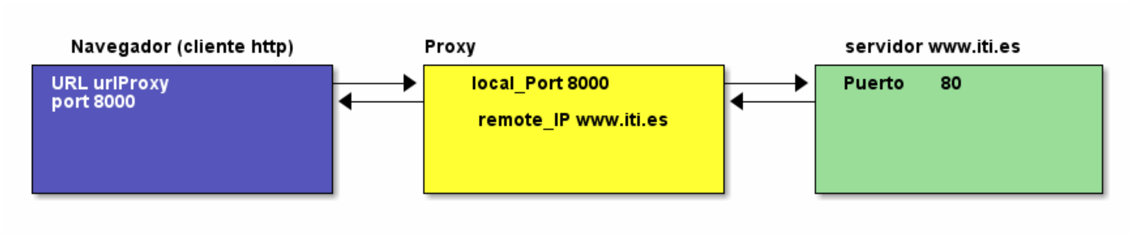
1. The service **invoker**, or the client that sends its requests. We can use any suitable external client for testing, like a browser (HTTP client).
1. The **proxy**, receiving requests directly from the client, and redirecting them to their final destination.
2. The final destination of the requests, the server actually implementing the **service**. It does not care where requests are coming from: wether the first actor (invoker) or the second (proxy). It behaves the same. For testing we can use a suitable external service like a conventional web server (given that our client needs to talk HTTP too)
2. The **protocol**, a specification of the messages and their flow between a client and a server.

The server acts at the TCP/IP level, not interfering at all with the contents of the messages being exchanged. It can change ports and addresses, but nothing else. Thus it acts as a neutral transport, ignoring the actual protocol employed between client and service.

- In sum, any TCP/IP protocol should be suitably handled by the proxy

Our first attempt will consists of a proxy that always contacts a pre-configured web server. The module **`myTcpProxy.js`** is a server with the following characteristics:

1. Waits for requests to port 8000
2. Upon a received request for connection, it builds another connection to the preconfigured target IP and port
   - For our exercise we will use port 80 of the web server of the Instituto Tecnológico de Informática (ITI) (within UPV) (IP 158.42.156.2)
3. Returns the response to the client



This "transparent" proxy is actually visible to the client (needs to contact it after all). It behaves like a server to the client and as a client to the actual service.

Code (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);
```

The proxy uses two sockets: *socket* to talk to the client, and *serviceSocket* to talk to the real service:

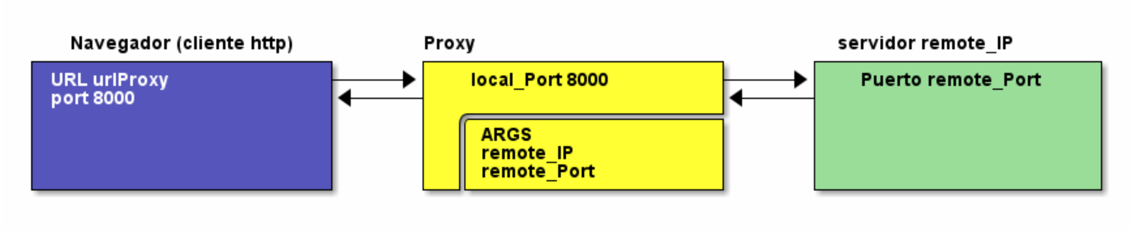
1. reads a message (`msg`) from the client

2. opens a connection to the server
3. writes a copy of the message
4. waits for the server's answer
5. returns a copy to the client

Verify the above program works properly by using a web prowser point to the URL http://proxy_IP_address:8000. Remember you can use `averiguar_ip.sh` to find the proxy's IP address.

Warning: When running your proxy you may get the error "EADDRINUSE". This may be caused by the existence of another program using the same port. Check that you did not leave any other experimental process running, other node problams that did not terminate their execution. This is a common gotcha. You can fix this problem by killing the offending processes. To inspect the processes running, use command '`ps -all`' in a shell terminal. You can kill all running node programs issuing command '`pkill -f node`' at the console. This command terminates all processes launched by the node runtime under our user name.

Modify the proxy's code so that the address and port of the final service are passed via command line arguments. This makes `myTcpProxy` usage much more flexible and general.



Thus now `myTcpProxy` has the following **input arguments** (in this order):

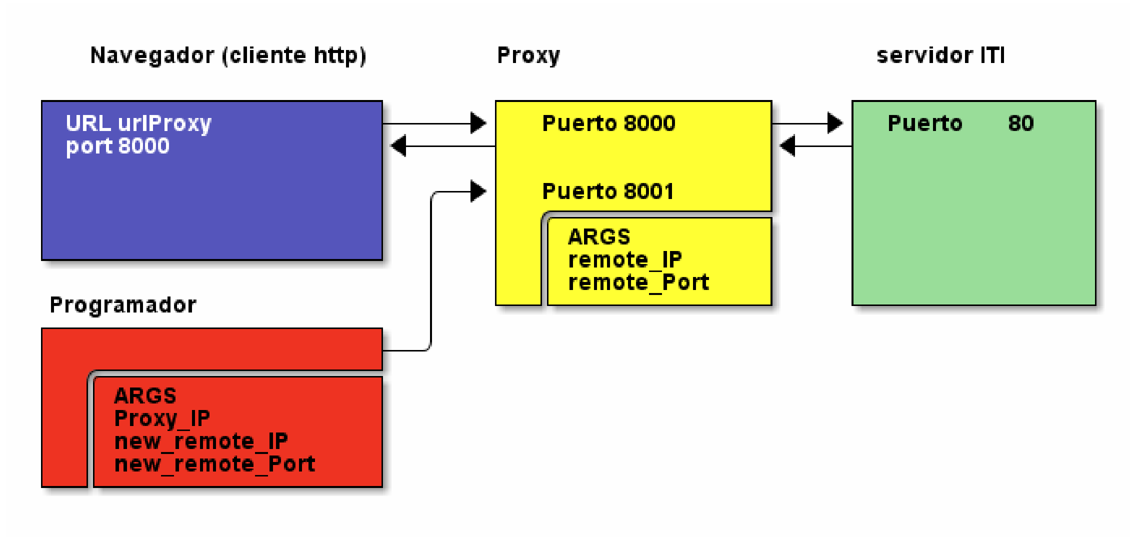
1. Remote IP (Server's IP address).
2. Remote port number (Port at which the remote server waits for requests).

Use this new version of `myTcpProxy` to access **ITI**'s web server again, as in the previous example. You must continue using port 8000 for the proxy.

Now `myTcpProxy` lets us access any remote server. Use it **with appropriate parameters,** to make `netClientLoad` and `netServerLoad` communicate through this proxy. `netClientLoad` should be started with the IP address of the proxy, instead of that of `netServerLoad`.

Note: If the proxy and `netServerLoad` run on the same machine change the port of `netServerLoad` to something different from 8000 (why?)

Create a modification of the proxy, and name it `newTcpProxy.js`. This new proxy will use a **programming port**.



Programming clients can send pair of values (IP, port) to reprogram the target server address/port while the proxy is running. You can use port 8001 for this.

- Requests to port 8001 must send the new values for the new server adderss and port.
- Requests to port 8000 are redirected to the target server, as configured through the command line or thorugh the last received programming message.

newTcpProxy has the same input arguments as **myTcpProxy**, to pass an initial server address/port.

1. Remote IP (Initial server's IP).
2. Remote port (Initial request port for the server)

NOTE that the proxy's request port is still port 8000, the programming port will be port 8001. These two values are preestablished and are not provided as arguments to the proxy.

You will need a programmer program to act as a client of the proxy's programming port. Let us name it **programador.js**. This program should expect the following input arguments:

1. Proxy's IP address
2. Remote server's IP address.
3. Remote server's port.

Note (again): the proxy's programming port is pre-established as 8001, and is not passed as an argument.

The **programador.js** program must send messages with the following format:

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

You can test **newTcpProxy** using the client/server pair **netServerLoad/netClientLoad**. You can also use a browser/web server pair.

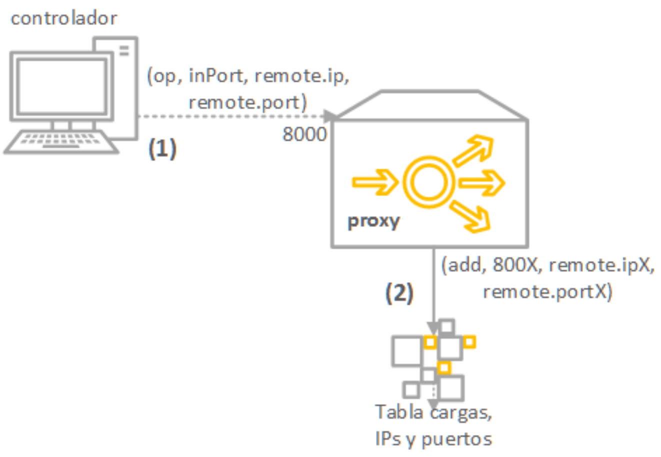
3 Part 3. Final application: reverse TCP/IP proxy

A reverse proxy is a server placed between clients and real servers providing the services clients want to access:

- It accepts client's request
  - Hiding server's topology
  - Can perform authentication and access control, encryption, etc.
- Redirects traffic towards suitable backend servers
  - Can be used to balance the load of those servers.
  - Can work around backend server failures.
- Can inspect, transform and reroute traffic when the underlying protocol is understood (e.g., HTTP).
  - Audits, logs, etc.

In our scenario, there are an arbitrary number of clients sending requests to ports 8001 to 8008[3] of our proxy.

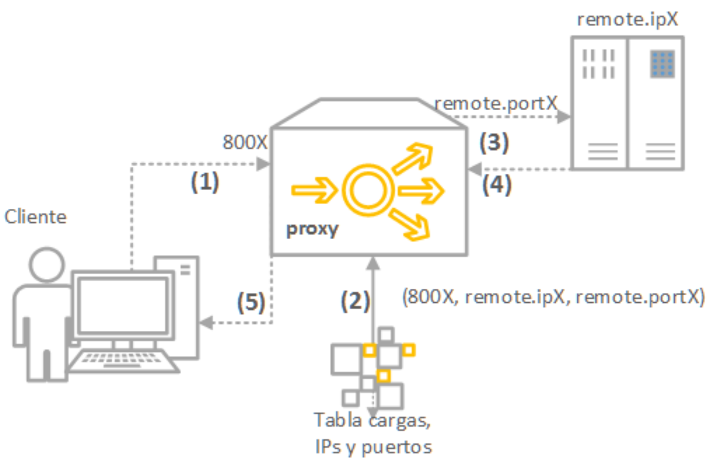
This time we use port 8000 (**`controlador`**), to configure the behavior of the proxy, indicating what service must be used as the final service for each port 8001 to 8008 of the proxy. E.g., if the proxy receives the following message through its port 8000:


controlador
(op, inPort, remote.ip, remote.port)
(1)
8000
proxy
(add, 800X, remote.ipX, remote.portX)
(2)
Tabla cargas, IPs y puertos

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

… From this moment on, requests arriving to the proxy through port 8001 will be connected to port 80 of 158.42.4.23. This is equivalen to the proxy we just saw earlier (**`myTcpProxy.js`**).

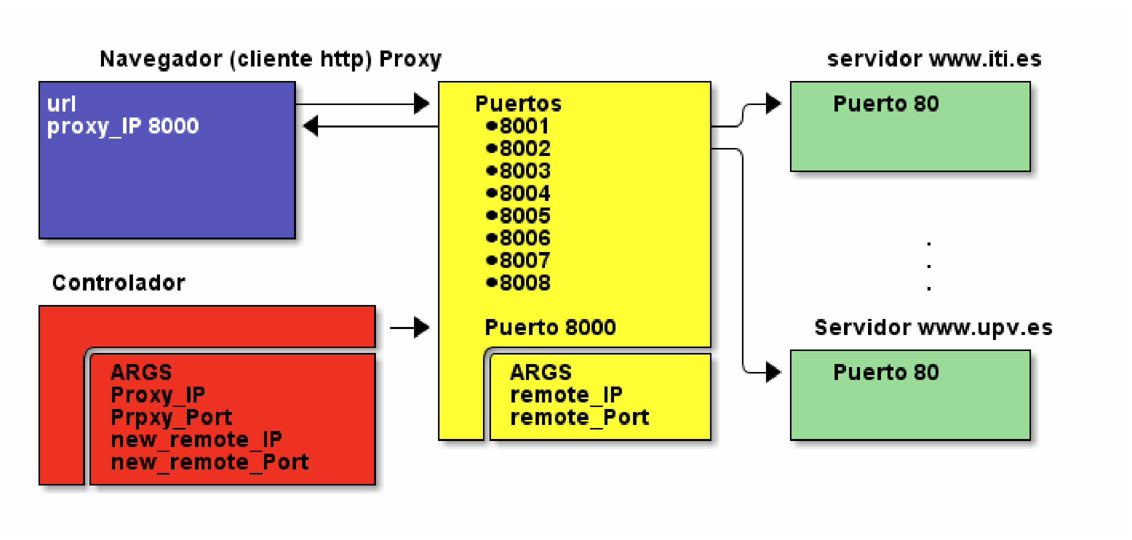
You must develop a **program capable to communicate with the proxy to configure it through the control port (8000)**. You must also **implement the proxy** working as described, of course!



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

- The name of the controller program program must be **`controlador.js`**. The proxy will simply be named **`proxy.js`**

`proxy` does not need any input argument, as the ports to be used will always be fixed as described earlier. We will bake in the addresses of the initial remote servers. Testing will use protocols HTTP, HTTPS y FTP via a web browser.



As an example we provide the following testing values for the initial table:

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 (ftp)
8005	www.disca.upv.es	80 (http)

But the program should work well with any valid table like the above.

On the other hand, **`controlador`** must accept the following **input arguments**:

1. Proxy's IP address.
2. Proxy's input port to be programmed.
3. New IP to be associated with the input port.
4. New remote port to be associated with the above proxy port.

Note: arguments 3 and 4 determine the new redirection the proxy must perform upon connections coming through the port provided in the second argument. Again, the programming port is fixed at 8000.