

TSR - LAB 1 COURSE 2017/18

INTRODUCTION TO THE LABS AND NODE.JS

TCP/IP REVERSE PROXY

This lab consists of three sessions and has the following learning goals:

1. Become familiar with the procedures and the tools needed for the work in the TSR labs
2. Use asynchronous programming to handle server requests, getting familiar with the NodeJS environment
3. Develop a solution for remote clients, based on the *proxy* and *reverse proxy* concepts

Links to external information:

- API Of NodeJS
- REPL
- Management of NodeJS events, and its http and net modules.

CONTENT

Content.....	1
1 Session 1.....	2
1.1 Tools.....	2
1.2 Procedures	2
1.3 First Steps in NodeJS.....	5
1.3.1 Access to files.....	5
Path module	5
1.3.2 Asynchronous programming: events.....	7
1.3.3 Client/server Interactions	7
Module http	7
Module net.....	8
1.3.4 JSON (JavaScript Object Notation).....	9
1.3.5 Access to command-line arguments	9
2 Session 2.....	10
2.1 How to query a computer's load	10
Transparent proxy	11
3 Session 3. Reverse proxy	13
4 Appendix. Using virtual machines	14
Transferring files	17
Start a remote session.....	18
Final details	18

1 SESSION 1

1.1 Tools

All labs are developed in JavaScript, using NodeJS.

Desktop computers at the labs provide access to sessions on shared virtual machines. This characteristic is **not compatible** with the use of **ports** or other exclusive resources, particularly in NodeJS. Because of this, these sessions aren't appropriate to run your applications developed in NodeJS. We will use **private virtual machines**, whose access procedure is described in the appendix of this document.

Some characteristics of these private virtual machines are:

- We have their complete control (root), and they will be accessed using a VPN. We may install on them any software package or module that we require.
- LINUX (CentOS7 of 64 bits), 2GB RAM, 50GB storage.
- NodeJS (version 6.11), with the eslint tool.
- Basic auxiliary libraries (fs, http, net, ...)
- The npm module manager (in order to install other NodeJS modules)
 - You'll need to run "npm init" if you install it on your own
- Text editors (geany and Visual Code Studio in the Programming menu). We can use any one of them or install any other.

You may install this same set of tools on your own computer, since it may work on Windows, Linux or MacOS. Thus, you'll avoid the port-related constraints that exist in the labs. Take a look at <http://nodejs.org> to this end.

1.2 Procedures

Fair use clause

The resources made available to the student are for the sole purpose of supporting the student's educational needs. The student commits to maintain the confidentiality of his/her passwords and other private credentials. 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 these resources and a violation of this clause.

To detect those deviations, we have set up suitable activity tracking mechanisms on those resources. Logs obtained through such mechanisms shall and will be used to support proposed sanctions for violations of this clause.

Generally, we may use any text editor to write our JavaScript programs, with a ".js" extension (e.g., x.js). Those programs will be run using the **node** command: e.g., **node x** (or **node x.js**).

The subsequent list presents small code fragments that correspond to concrete aspects that we should master for rapid application development in the following sessions. Recommendations:

1. Create a specific directory **TSRprac1**, and create inside it the subdirectories **pruebas**, **carga**, **proxy**, and **proxyInverso**
2. **Type** the code instead of doing a “copy and paste” from the PDF document
 - a. It requires little endeavour (the programs are short)
 - b. It facilitates to become familiar with the code and the programming language
 - c. It confronts us with typical syntactic errors
 - d. It prevents “copy and paste” errors from appearing. Many times, some characters from the PDF file are converted into others in the automated “copy” stage.

In the lab sessions, using JavaScript, we may find distinct **types of errors**.

Syntactic error		
Definition	Detection/solution	Example
Illegal syntax (invalid identifier, structures incorrectly nested, etc.)	The interpreter indicates the error type and its location in the code. Correct on code source	<pre>> "Potato"(); TypeError: string is not a function at repl:1:9 at REPLServer.defaultEval (repl.js:132:27) at bound (domain.js:254:14) at REPLServer.runBound [as eval] (domain.js:267:12) at REPLServer.<anonymous> (repl.js:279:12)</pre>
Logical error		
Definition	Detection/solution	Example
Programming error (try to access a non-existent property or 'undefined', pass arguments of an incorrect data type, etc.) JavaScript is not strongly type-oriented: some errors, which other languages catch at compilation time, are only shown when the program is run.	Wrong results when executing. Modify the code. It suits that in the code verify always the restrictions to apply on the arguments of the functions	<pre>> function add(array) {return array.reduce(function(x,y){return x+y})} undefined > add([1,2,3]) 6 > add(1) TypeError: undefined is not a function at add (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)</pre>
Operational error		
Definition	Detection/solution	
Exceptional situations that can arise during the execution of the program. They can be due to: <ul style="list-style-type: none"> • The environment (e.g., memory exhaustion, too many open files) • Current system configuration (e.g., there is no route to a remote host) • Use of the network (e.g., problems with the use of sockets) • Access problems to a remote service (e.g., I cannot connect to the server) • Wrong or inconsistent input data • Etc. 	Use of try , catch , and throw , similar to those of Java Strategy: <ul style="list-style-type: none"> - If it is clear how to resolve the error, manage it (e.g., error when opening a file for writing -> create it as a new file) - If the management of the error is not responsibility of this fragment of the program, propagate the error to the caller. - For transitory errors (e.g., network-related problems), retry that operation If we cannot manage neither propagate the error: <ul style="list-style-type: none"> - If it prevents the program from going on, abort it. - In another case, write the error down in a log file. 	

To minimise errors, we recommend:

- Use the strict mode: **node --use_strict file.js**
- Use **eslint** to diagnose the syntax and *style* of our program: **eslint file.js**
- Document correctly each function
 - Meaning and type of each argument, as well as any additional constraint
 - Which type of operational errors can appear, and how to manage them
 - Its intended return value
- Use the package **assert** in the code (with the basic operations **equal(expr1,expr2, errorMessage)** and **ok(logicalExpression, errorMessage)**)

This is an example that applies those recommendations

```

/*
 * 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 */
}

```

1.3 First Steps in NodeJS

1.3.1 Access to files

All methods related to files appear in the 'fs' module. The operations are asynchronous by default. However, for each asynchronous function **xx**, there is also a synchronous variant **xxSync**.

- (f1.js) Read the content of a file

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

- (f2.js) Write content to a file

```
var fs = require('fs');
fs.writeFile('/tmp/f', 'content of the new file', 'utf8',
  function (err,data) {
    if (err) {
      return console.log(err);
    }
    console.log('This writing has been completed.');
```

- (f3.js) Access to directories

It gets the list of files in the directory DIR or any of 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('.')); // search in current directory
```

Path module

It contains functions that simplify the manipulation of path names.

To show an alternative in the development and testing of code, we use in these examples an interactive modality based on *the Read-Eval-Print-Loop* (REPL).

- To this end, we run (without arguments) the **node** command. This opens a terminal where we can run interactively any JavaScript statement.

The examples are:

- **normalize**. From a string with a path name, it interprets the separators and the special directory names, and returns a new string that corresponds to this same pathname once normalised.

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

- **join**. From a variable list of arguments, joins them and normalises the resulting pathname.

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

- **basename**, **extname** and **dirname**. They allow to extract the distinct components of a pathname

```
> 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**. It checks whether a given pathname exists or not.

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

1.3.2 Asynchronous programming: events

single.js	eventSimple.js
<pre> function fib(n) { return (n<2)? 1: fib(n-2)+fib(n-1); } console.log("Initiating the execution..."); setTimeout(// Delay of 10 ms. function() { console.log('M1: First message??'); }, 10); var j = fib(40); // It needs more than 1 sec. function otherMsg(m,rs) { console.log(m + ": The result is "+rs); } otherMsg("M2",j); //M2 is written before M1 setTimeout(// M3 is written after M1. Why? function() {otherMsg('M3',j);}, 1); </pre>	<pre> var ev = require('events'); var emitter = new ev.EventEmitter; var e1 = "print", e2= "read"; // name of events var n1 = 0, n2 = 0; // auxiliary vars // register listener functions on the event emitter emitter.on(e1, function() { console.log('event '+e1+' :'+(++n1)+' times')}); emitter.on(e2, function() { console.log('event '+e2+' :'+(++n2)+' 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 </pre>

1.3.3 Client/server Interactions

Module http

For development of web servers (HTTP servers)

Example: Web server that greets to the client that contacts it

Code	Commentary
<pre> var http = require('http'); function dd(i) {return (i<10?"0:"")+i;} var server = http.createServer(function (req,res) { res.writeHead(200,{'Content-Type':'text/html'}); res.end('<marquee>Node and Http</marquee>'); var d = new Date(); console.log('Somebody has accessed this at '+ d.getHours() + ":" + dd(d.getMinutes()) + ":" + dd(d.getSeconds())); }).listen(8000); </pre>	<p>It imports module http dd(8) -> "08" dd(16) -> "16"</p> <p>It creates the server and associates it this function that returns a fixed response and writes the current time to the console</p> <p>The server listens to port 8000</p>

Please, run the server and use a web browser as its client:

- Access to the URL `http://localhost:8000`

- Check (in the browser) which is the response from the server, and in the console which is the message written by the server.



Node y HTTP

```

Terminal
Archivo  Editar  Ver  Terminal  Ir  Ayuda
bash-4.1$ node ejemploSencillo.js
Alguien ha accedido a las 13:35:18

```

Module net

Client (netClient.js)	Server (netServer.js)
<pre> 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'); }); </pre>	<pre> var net = require('net'); var 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'); }); </pre>

1.3.4 JSON (JavaScript Object Notation)

It is a format to represent data that has turned into de facto standard for the web.

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

Client (JSONc.js)	Server (JSONs.js)
<pre> ... var msg = JSON.stringify({ "name": "mkyong", "age": 30, "address": { "street": "8th Street", "city": "New York" }, "phone": [{ "type": "home", "number": "111-1111" }, { "type": "fax", "number": "222-2222" }] }); var socket = net.connect({port:8000}, function() {socket.write(msg);}) </pre>	<pre> ... var server = net.createServer(function(c) { c.on('data', function(data) { var person = JSON.parse(data); console.log(person.name); //mkyong console.log(person.address.street); //8th Street console.log(person.address.city); //New York console.log(person.phone[0].number); //111-1111 console.log(person.phone[1].type); //fax })}); server.listen(8000, function() { //listening listener console.log('server bound'); }); </pre>

1.3.5 Access to command-line arguments

The shell collects all the arguments from the command line and passes them to the JavaScript program packed in an array called `process.argv` (Abbreviation of 'argument values').

`Process.argv` is an array, thus we can get its length and access to each argument by its position

- `process.argv.length`: Number of arguments received from the command line
- `process.argv[i]`: This gets the i-th argument; e.g., for the command line "`node program arg1 ...`", then `process.argv[0]` holds the string 'node', `process.argv[1]` contains the string 'program', `process.argv[2]` holds 'arg1', etc.

Example: Write a file called **args.js** with the following code

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

...and invoke it with multiple arguments. For instance:

```
node args.js one two three four
```

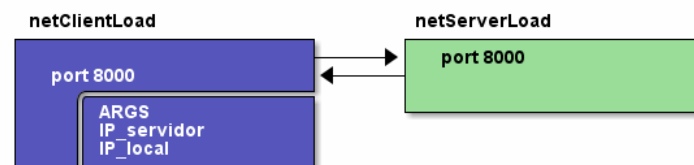
The two first elements of the array correspond to 'node' and the path of the program, respectively. We may use `process.argv.slice(2)` for removing 'node' and the path of the program, so that in `process.argv` only the actual program arguments remain.

2 SESSION 2

2.1 How to query a computer's load

Some scalable client/server systems, replicate their servers (using horizontal scalability), and deliver the requests among them according to their current workload level.

We create the embryo for a system of this type, with a single client and a single server (possibly in distinct machines) that communicate through port 8000



- The server is called **netServerLoad.js**, and it does not receive command-line arguments .
 - The following function **getLoad** computes its current load. That function reads the data from the file **/proc/loadavg**, filters its interesting values (adds one hundredth of a second to them in order to avoid the confusion between value 0 and an error), and processes them calculating a weighted average (with weight 10 to the load of the last minute, weight 2 to the last 5 minutes, and weight 1 to the last 15)

```

function getLoad(){
  data=fs.readFileSync("/proc/loadavg"); //It requires 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;
};
  
```

- The client file is **netClientLoad.js**: it receives as its command-line arguments the IP address of the server and its local IP address.
- Protocol: When the client sends a request to the server, it includes its own IP, the server computes its load and returns a response to the client that includes its own IP (i.e., that of the server) and that workload level (i.e., the result of that call to **getLoad**).

To help you test and debug your code, we are keeping an active server at **tsr1.dsic.upv.es**

Important:

- Your starting point to develop these programs consists of the code for netClient.js and netServer.js seen earlier
- You must make sure that all processes terminate (e.g. using **process.exit()**)
- You can use either the domain name or the IP address for a server machine. This also works for the virtual machines you will be using.

Complete both programs, place them in different VMs (with the help of a classmate), and test their communication through port 8000: **netServerLoad** must compute the load as an

answer to each request from a client. Correspondingly, `netClientLoad` must use its console to show the answer received from the server.

Transparent proxy

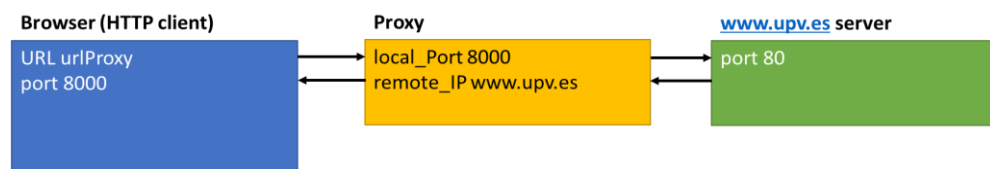
A proxy is a server that, when invoked by a client, forwards the request to a third party (the real server), who, after processing the request, returns it to the proxy, which, in turn routes it to the original client that sent the request.

- From a client's point of view, it works as the real server (hides the real server)
- It can run on an arbitrary machine (unrelated to the client's or server's)
- The proxy can use different ports/addresses, but does not modify the messages in transit.

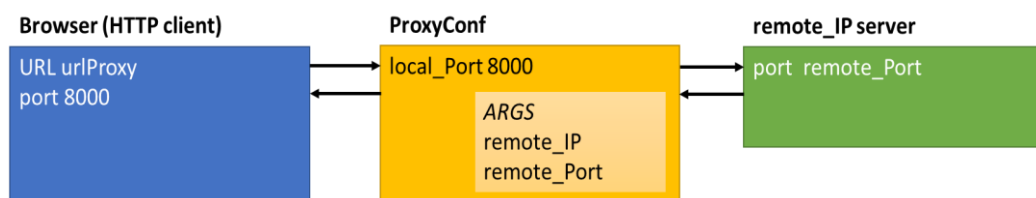
More info in http://en.wikipedia.org/wiki/Proxy_server

You should progress through three incremental stages:

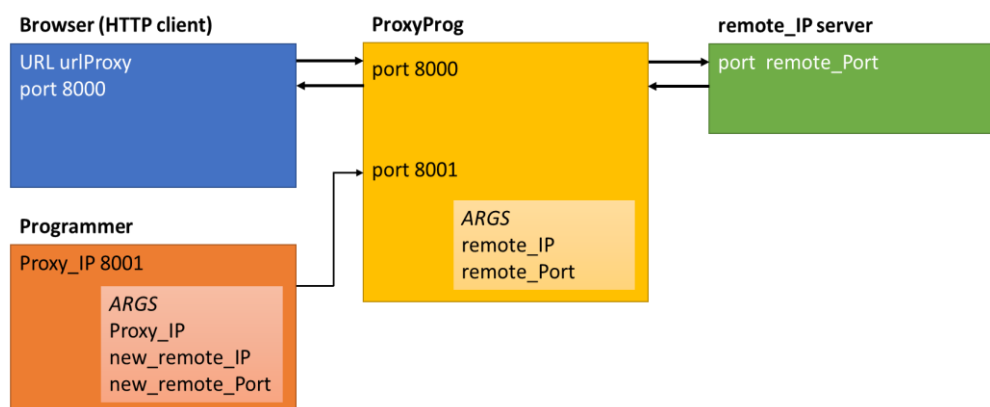
1. **Basic proxy (Proxy)**. When an HTTP client (e.g., a browser) contacts proxy's port 8000, the proxy forwards that request to the web server of the UPV (158.42.4.23 port 80), returning each server answer to the client.



2. **Configurable Proxy (ProxyConf)**: instead of hard coding the IP and port of the real target server in its code, it gets them through command line arguments



3. **Programmable Proxy (ProxyProg)**. Initially, the IP address and port number are passed through command line arguments, but they can be altered at run time, sending an appropriate message to port 8001 of the proxy.



Important:

- We provide the code for Proxy.js: you should study it until you understand how it works.
 - Verify it works correctly by pointing a browser to URL `http://proxy_address:8000/`

Code (Proxy.js)	Comments
<pre> var net = require('net'); var LOCAL_PORT = 8000; var LOCAL_IP = '127.0.0.1'; var REMOTE_PORT = 80; var REMOTE_IP = '158.42.4.23'; // www.upv.es var server = net.createServer(function (socket) { var 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); </pre>	<p>Uses a socket to talk with the client (socket) and another to talk with the server (serviceSocket)</p> <ol style="list-style-type: none"> 1.- It opens a connection to the server 2.- It reads a message (<code>msg</code>) from client 3.- It writes a copy of the message 4.- It waits for an answer, and copies it to the client.

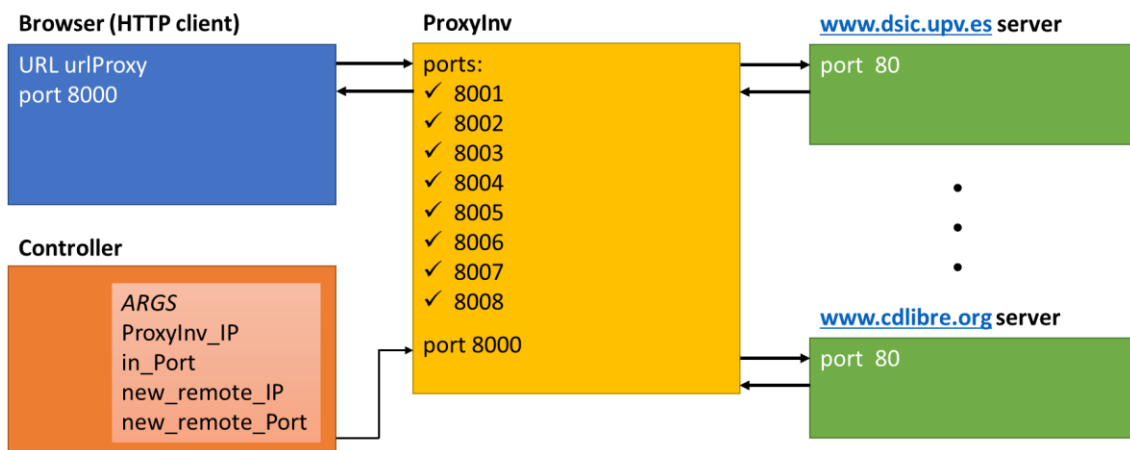
- In scenario 2, you must test the following cases:
 - Access to **UPV's** server should still work
 - ProxyConf proxying between netClientLoad and netServerLoad
 - If ProxyConf runs in the same machine as netServerLoad, their ports will collide -> modify netServerLoad's code to use a different port.
 - If you get an "EADDRINUSE" error when running the proxy, another program is using the proxy's port.
- In scenario 3 we need to code the program used to reconfigure the proxy. We call it **programador.js**
 - **programador.js** gets the IP of the proxy, as well as the IP and port of the new target server through command line arguments.
 - **programador.js** encodes the new server's IP and port, and sends them as a message to port 8001 of the proxy's machine, after which it terminates.
 - The format of **programador.js** messages should be like this:

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

3 SESSION 3. REVERSE PROXY

A reverse proxy behaves as an intermediary between several clients and a pool of servers.

- It hides the topology of servers.
 - Clients always contact the proxy (they do not know the IP or identities of the real servers)
- Can perform authentication, access control, encryption, etc.
 - Can inspect, transform and route HTTP traffic (audits, logs, etc.)
- Redirects traffic to the appropriate server
 - E.g., to balance the load or handle a server's failure



Our reverse proxy (**ProxyInv**) does not implement any characteristic related with efficiency (e.g., caches) or reliability (e.g., failure detection and request retries). Thus, it is a very simple variant of that kind of proxies. It receives no command line arguments. It expects client requests on ports 8001 through 8008, initially mapping requests on each one of those ports to the target servers shown in the following table

Proxy port	FQDN of remote service	IP	Remote service port
8001	www.dsic.upv.es	158.42.184.5	80 (http)
8002	www.upv.es	158.42.4.23	80 (http)
8003	www.libreoffice.org	89.238.68.168	80 (http)
8004	memex.dsic.upv.es	158.42.179.56	8080 (http)
8005	www.cdlibre.org	147.156.222.65	80 (http)

ProxyInv can be programmed through port 8000. The programming client has the name **controlador.js**, and uses these messages to talk to the proxy

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

Controlador.js must accept the following command line arguments:

1. Proxy IP address.
2. Proxy port to be reprogrammed (within range 8001..8008)
3. New target IP.
4. New target port.

4 APPENDIX. USING VIRTUAL MACHINES

In order to have a more realistic scenario for deploying our programs, we will use a virtualization service, with one virtual machine assigned to each student. The goal of this appendix is to help the student understand how to use this setup.

The system images on each VM are similar to those installed in the physical laboratory machines. The main differences are:

- The student can administer his/her VM, thus he/she can perform more critical operations than a standard user can (e.g., install new software, add users, ...).
- VMs are stateful: changes are persisted, and survive cycles of activation/deactivation.
- Each VM has its own IP. This IP can be contacted from within a special VPN (Virtual Private Network). You need to work within this VPN in order to access one of the course's VM.
- The **virtual desktops** in which students log in when in the DSIC laboratories are already connected to that VPN, thus they can access the VMs.
- If you need to access the VMs when not in the laboratories (although you should still be in the UPV's VPN), the recommended method is to establish a connection to a virtual desktop via either:
 - Windows: `windesktop.dsic.upv.es`
 - LINUX: `linuxdesktop.dsic.upv.es`
 getting into an environment very similar to that available within the DSIC laboratories, and having access to the VMs.

Assuming you have access to this VPN, let us focus on how to work from such a **virtual desktop** environment. From this VPN we will have access to the domain **.dsic.cloud**.

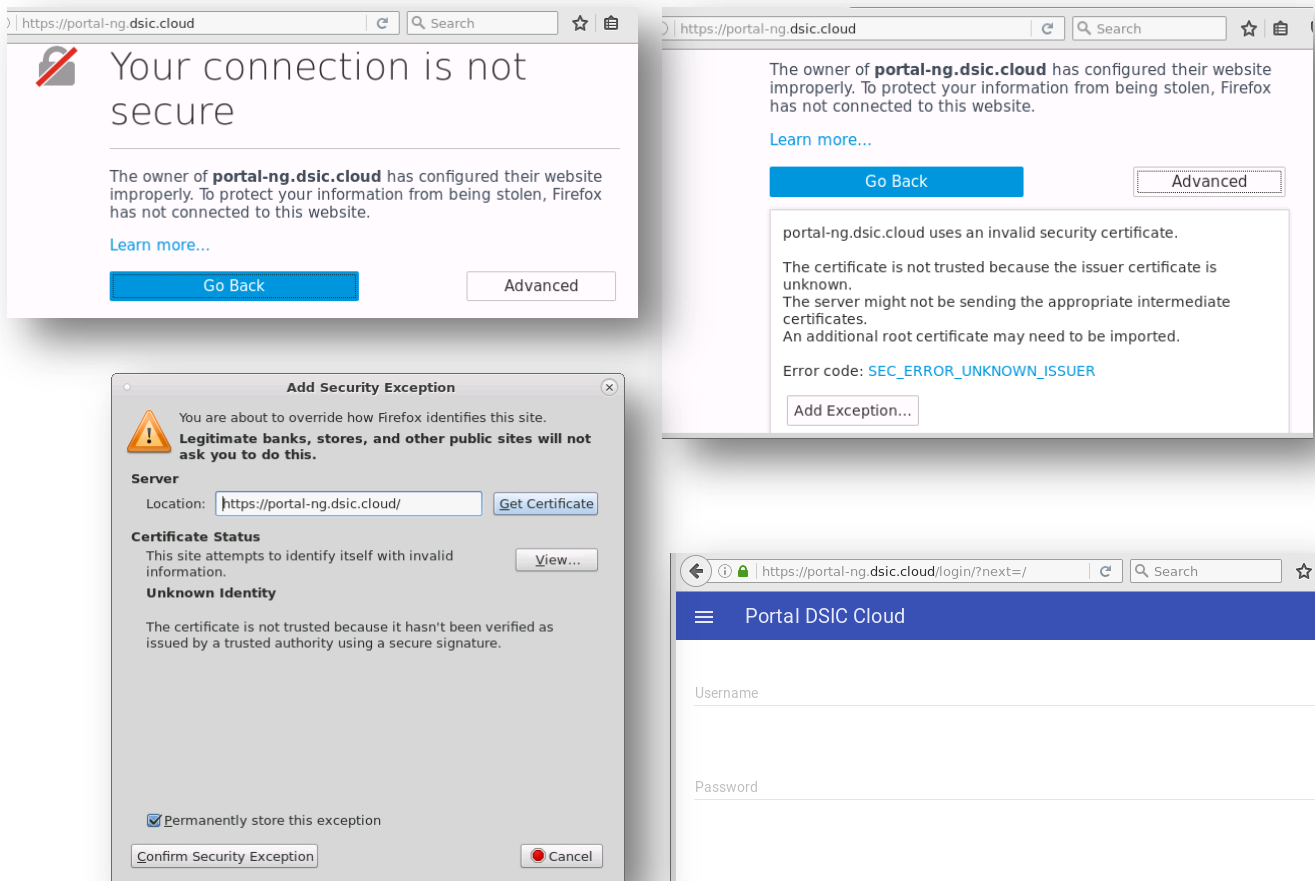
It is important to distinguish VM management activity from activity within a VM. You need a specific service to manage your virtual machine. In particular you need such a service to boot the VM up. You can access the VM management service by connecting to the following URL:





`https://portal-ng.dsic.cloud.`

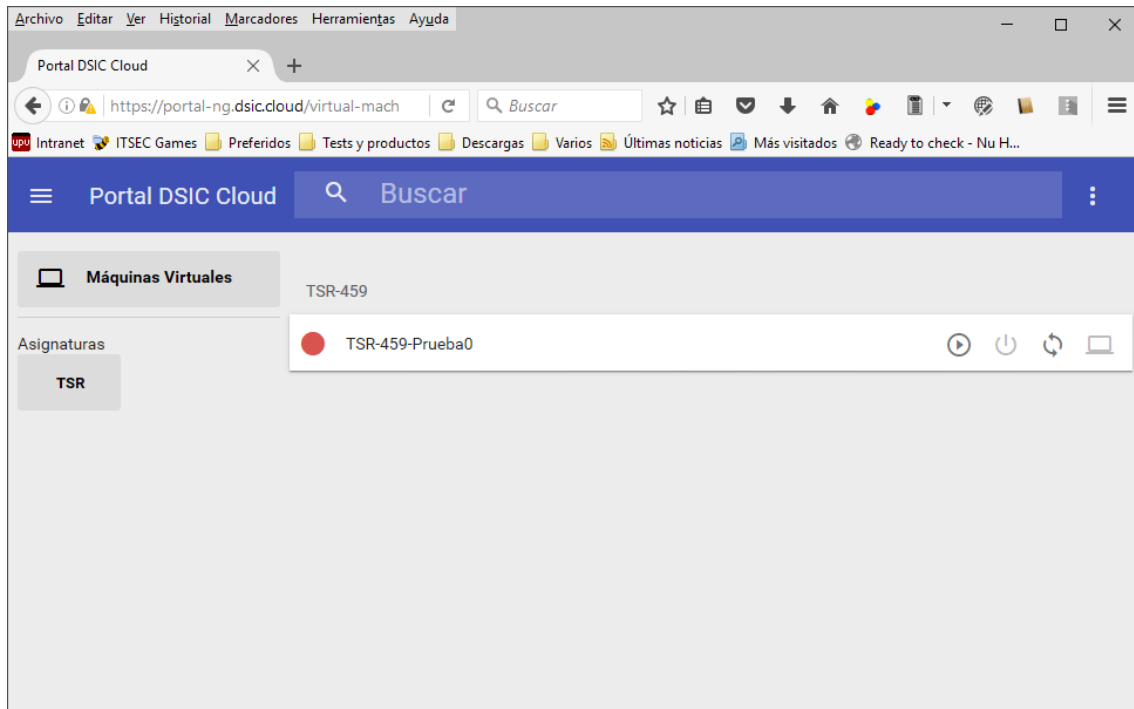
- Once you authenticate yourself (use your student credentials) it shows the list of VMs you have available, with a set of operations you can carry out on each VM.

The steps to access this service are:

1. The first time around, your browser will probably complain, as the server is not using an officially signed certificate. You must tell your browser to accept such certificate by adding a security exception, as shown in the following figures.



2. After this configuration step, the server shows an identification dialog where it is waiting for those credentials you use to access the DSIC laboratories.
3. Once you get past the previous challenge, the service shows the list of VMs you have access to. A red dot means the machine is down. In the same row, to the right, you can find 4 icons: to boot () the VM, stop (), refresh () the information or start () an interactive session using VNC. Shadowed icons indicate the operation is not currently available.



4. Pushing the VM name label expands some extra details, including its IP.

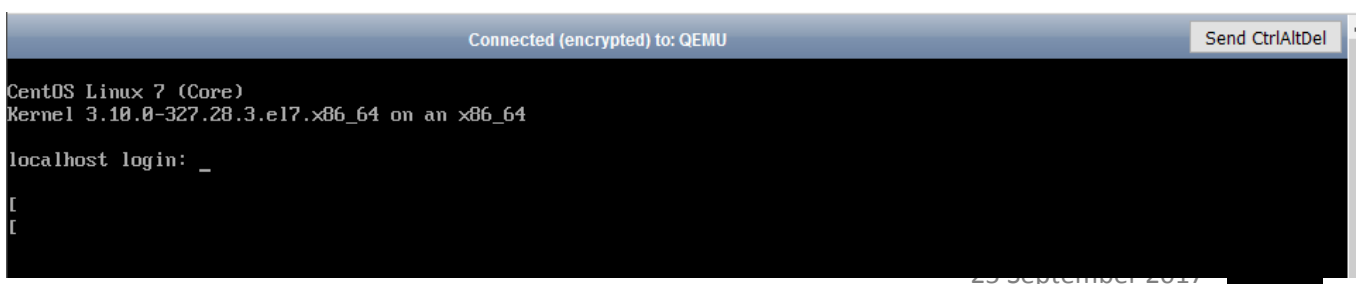


5. After booting up the machine, this is shown:

- a) Our VM name should be completed using the “**.dsic.cloud**” domain name.

In the current example, from the VM name *TSR-459-Prueba0*, we can obtain its FQDN as *tsr-459-Prueba0.dsic.cloud*.

- b) In order to boot up a stopped VM we MUST use this service. You should avoid shutting down a VM through this service, however, as the shutdown will be uncontrolled (similar to pulling the power plug from a physical computer). It is preferable to shut down from within the VM executing a command like this: “`shutdown -h now`”
- c) VNC console access is implemented by the browser, causing a warning dialog



similar to the one we presented earlier on.

Next, we show how to carry on the most common activities: transfer files, and initiate a remote connection. In both cases we assume you operate from within the VPN.

Transferring files

a) LINUX command line:

- If working from Windows, open a Git-bash command window. From this command window you can access many common UNIX commands.

```
scp files root@mivirtual.dsic.cloud:
```

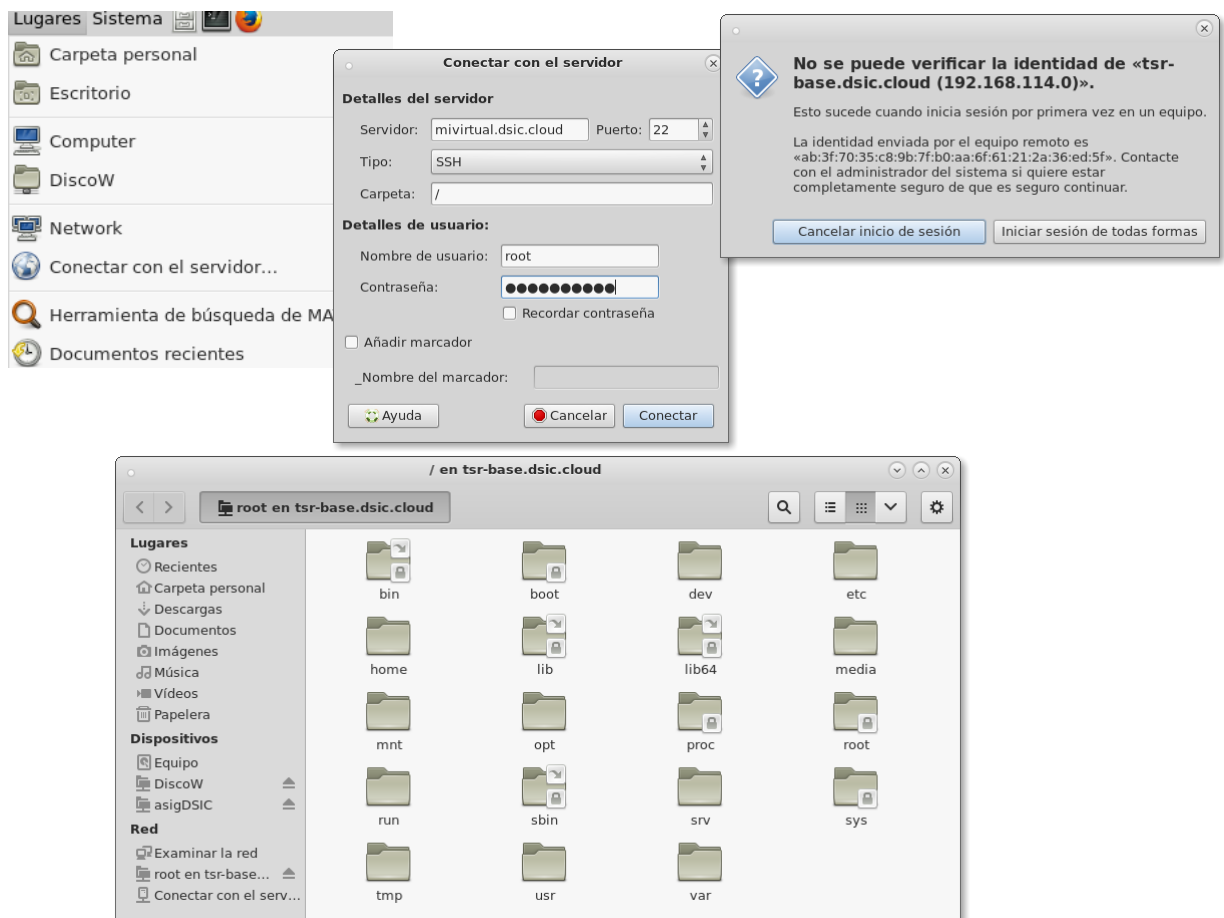
Check out scp's manual page for more detailed instructions.

b) Graphic environment

From your LINUX desktop environment, locate the Sistema menu. There you will find an item called “Conectar con el servidor...”

- Configure it for *mivirtual.dsic.cloud*, user *root*, port 22, protocol *ssh*.

Disregard warnings about the identity of the target machine. After this, you can **work with the remote file system** via a file explorer window.



Start a remote session

- a) Command line
 - If on Windows, use Git-Bash, as before.

`ssh root@mivirtua1.dsic.cloud`
Check ssh man page for more detailed usage information.
- b) Graphic desktop: use a remote desktop protocol (RDP) client
 - We get a graphical LINUX desktop environment on the VM.

Final details

The password for **root** on the VMs will be given to you in the lab. YOU MUST CHANGE IT WHEN YOU FIRST LOG IN.

It is never ok to keep an administrative password known by all. Note that failing to change the password you are inviting undesirable interference from others. Failing to perform this change impacts the security of your environment and it is your responsibility.

Initial configuration of the VM's firewall blocks traffic to many of the VM ports from outside. To enable access to the ports you need (e.g. from 8000 through 8100), you must run the following instructions once:

```
firewall-cmd --permanent --add-port=8000-8100/tcp  
firewall-cmd --reload
```

Since lab resources are scarce, it is natural for two students to share the same terminal during a lab session, which usually leads to both using the same VM. It is important to organize access to that VM to avoid interference between both students.

It will be helpful ...

1. To verify if any other user is accessing the VM when you access it.
2. Use some instant messaging service to communicate accesses.
3. Agree beforehand on a schedule or pattern to access the VM.