TSR - PRACTICE 1
(Second part)

NODEJS

Lab 1 consists of 2 parts, a part dedicated to JavaScript (1 session) and a part dedicated to NodeJS (2 sessions). **This document describes the second part of the practice, dedicated to NodeJS** .

Its goals are:

1. Introduce the procedures and tools necessary in order to work in the TSR laboratory.
2. Introduce basic techniques for programming and development in JavaScript and NodeJS.

INTRODUCTION

Network tools

The practices are carried out on virtual machines using JavaScript, i.e. the NodeJS environment. This eliminates the risk of conflicts (e.g., trying to use the same port number in their processes) that exists when multiple students use concurrently poliLabs sessions, since some sessions may be supported by the same server computer.

Students can install a similar Node.js environment on their own computers under Windows, LINUX or MacOS. See http://nodejs.org

Procedures

In general, we write the JavaScript code with any editor in a file with the extension js (x.js), and we execute that code with the command **node x[.js] [args]**

To minimize possible errors, it is recommended:

- use strict mode: `node --use_strict file.js`
- properly document each interface function
  - meaning and type of each argument, as well as any additional restrictions
  - what kind of operational errors can appear, and how they will be managed
  - the return value

Practice 1 (second part): NodeJS

For this second part we have 2 sessions. A work to be done is proposed for each of the sessions. However, the student can manage his/her time and dedicate more or less to each session than we propose to end up completing all the aspects that are mentioned.

- **Session 1.2: NodeJS modules.**
- **Session 1.3: Reverse proxy.**

Session 1.2: NodeJS modules.

NodeJS includes a wide variety of modules that provide useful functionality for working with files, with the network, with security aspects, etc. In this session we are going to work with some of the most relevant modules.

Modules to be covered: **fs, events, http, net.**

During this session you will have to practice with the code that is provided as an example for each of these modules, and answer the questions or carry out the activities that are proposed.

1.2.1 FS MODULE: FILE ACCESS

All the methods corresponding to operations on files appear in the fs.js module. The operations are asynchronous, but for each asynchronous function **xx** there is usually the synchronous variant **xxSync**. The following programs can be copied and executed.

Read the following sections A, B and C of this section. For each of them copy the code, analyze it and run it until you understand it. At the end there are a series of questions that you must answer.

A. Asynchronously reading the content of a file (`read1.js`)

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

B. Asynchronously write the content to a file (`write1.js`)

```
const fs = require ( 'fs' );
fs . writeFile ( '/tmp/f' , 'contents of new file' , 'utf8' ,
  function ( err ) {
  if ( err ) {
    return console . log ( err );
  } }
  console . log ( 'write completed' );
});
```

C. Adapted reads and writes. Use of modules.

We define the **`fiSys.js` `module`**, based on **`fs.js`**, to access files along with some examples of its use.

(**`fiSys.js`**)

```
//fiSys module
//Example of functions module adapted for the use of files.
//(More functions could have been defined.)

const fs=require("fs");

function readFile(file,callbackError,callbackRead){
        fs.readFile(file,"utf8",function(error,data){
               if(error) callbackError(file);
               else callbackRead(data);
        });
}


function readFileSync(file){
        var result; // It will return undefined if any error occurs in reading
        try{
                result=fs.readFileSync(file,"utf8");
        }catch(e){};
        return result;
}


function writeFile(file,data,callbackError,callbackWrite){
        fs.writeFile(file,data,function(error){
                if(error) callbackError(file);
                else callbackWrite(file);
        });
}

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

Reading program using the “fiSys” module (**`read2.js`**)

```
//File reads

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


//For asynchronous reading (/proc/loadavg is only available in Linux systems):
console.log("Invoke asynchronous read");
fiSys.readFile("/proc/loadavg",cbError,format);
console.log("Asynchronous read invoked\n\n");

//Synchronous reading
console.log("Invoke synchronous read");
const data=fiSys.readFileSync("/proc/loadavg");
if(data!=undefined)format(data);
        else console.log(data);
console.log("Synchronous reading finished\n\n");

//- - - - - - - - - - - -
function format(data){
        const separator=" "; // space
        const tokens = data.toString().split(separator);
        const min1 = parseFloat(tokens[0])+0.01;
        const min5 = parseFloat(tokens[1])+0.01;
        const min15 = parseFloat(tokens[2])+0.01;
        const result=min1*10+min5*2+min15;
        console.log(result);
}

function cbError(file){
        console.log("READ ERROR in "+file);
}
```

Writing program using the fiSys module (**`write2.js`**)

```
//Asynchronous writing of files

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

fiSys.writeFile('text.txt','content of the new file',cbError,cbWrite);

function cbWrite(file){
        console.log("writing done to: "+file);
}

function cbError(file){
        console.log("WRITE ERROR in "+file);
}
```

Questions

1. Reason out how many iterations of the event loop (turns) occur in each of the analyzed programs. 2
2. In section C a user module has been developed. Describe the steps that have been taken in the code to create a module, as opposed to the code required to program an application. `exports.functionName = functionName` has been used to export certain functions so they can be used from other files
3. Write 2 new versions of section A. One through promises and the other through async/await. Use the theory material as a base.

1.2.2 EVENTS MODULE

The "events" module provides some functionality for using events in NodeJS. It is worth noting that several NodeJS modules use this module to handle events. Let us consider the following two programs that you must analyze and understand. Besides, you must complete the third one with your own code.

A. Program that uses the "events" module. (`emitter1.js`)

Analyze and run the program until you understand how it works.

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

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

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

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

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

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

B. Program that uses the “events” module. (**emitter2.js**)

Analyze this other example (emitter2.js). When raising events, associated values (arguments to the event 'listener') can be generated.

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

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

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

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

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

Exercise

Let us consider this initial code of the “emitter3” program. You must complete it so that it meets certain specifications that we detail below.

Code to complete (**emitter3.js**)

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

function rand () { // should return random values in range [2000,5000) (ms)
... // Math.floor(x) returns the integer part of the value x
... // Math.random() returns a value in the range [0,1)
}

function handler ( e , n ) { // e is the event, n the associated value
return ( inc ) => {..} // listener receives a value (inc)
}

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

function stage () {
...
}

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

Complete the code (adding code in place of the ellipses) without removing code so that it does the following:

The code must call the "stage" function a first time. Within the “stage” function, the call to “stage” must be reprogrammed so that the program executes a series of “stages”.

Each stage must start with a delay of between 2 and 5 seconds. Each stage should do the following:

- Emit the events e1 and e2, passing the value of the variable "inc" as the associated value.
- Increase variable “inc” one unit
- Show by console the delay in the execution of the stage.

Example run (initial snippet only)

```
e1 -- > 0
e2 -- > 0
stage 1 started after 3043 ms
e1 -- > 1
e2 -- > 01
stage 2 started after 3869 ms
e1 -- > 3
e2 -- > 012
stage 3 started after 2072 ms
e1 -- > 6
e2 -- > 0123
stage 4 started after 2025 ms
e1 -- > 10
e2 -- > 01234
stage 5 started after 2325 ms
```

Questions

The code that has been proposed for "emitter3", must reprogram each new stage within the stage itself. Discuss the implications of programming all the stages at once with code similar to the following:

```
t = 0

for (let i=0; i<=10; i++) {

      t = t + rand();

      setTimeout(stage, t);

}
```

1.2.3 HTTP MODULE

Module with functions for development of Web servers (HTTP servers)

You are asked to analyze and verify the operation of the following program that uses the “http” module.

Web server (`ejemploSencillo.js`) that greets the client

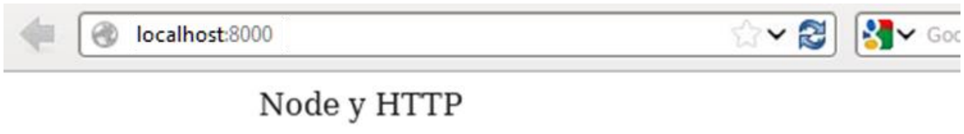
Code	Comment
`build http = require ( 'http' );`	Import http module
`function dd ( i ) { return ( i < 10 ? "0" : "" )+ i ;}`	dd(8) -> "08" dd(16) -> "16"

`const server = http.createServer(`
`    function (req,res) {`
`        res.writeHead(200,{'Content-Type':'text/html'});`
`        res.end('<marquee>Node y Http</marquee>');`
`        var d = new Date ();`
`        console . log ( 'Someone has accessed this site at ' +`
`            d . getHours () + ":" +`
`            dd ( d . getMinutes ()) + ":" +`
`            dd ( d.getSeconds ( ) ));`
	create the server and associate this function which returns a fixed response
	and also writes the time to the console

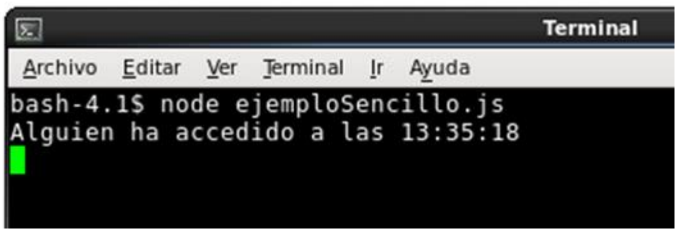
`}). listen ( 8000 );`	The server listens on port 8000

Run the server and use a web browser as a client

- Access the URL `http://localhost:8000`
- Check in the browser the response from the server, and in the console the message written by the server



Node y HTTP



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

1.2.4 NET MODULE

This module contains the API to use basic TCP/IP sockets.

Two programs are provided, client and server. Analyze the code of both and run them several times to see how they work.

Client (`netClient.js`)

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

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

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

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

Server (`netServer.js`)

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

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

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

Exercise

Modify the code of these client and server programs to achieve a service that provides the load on the server computer. Using "netClient" as a base, create a "netClientLoad" program. Using the “netServer” program, create a program that we will call “netServerLoad”.

To carry out this activity you must complete 2 parts:

Part 1: Modify the "netClientLoad" program you just created, so that it uses the command line arguments correctly.

Part 2: Update the “netServerLoad” program. To this end, use the “getLoad()” function that you can embed in your code

Activity Part 1 – Accessing Command Line Arguments

The shell collects all the arguments on the command line and passes them to the JS application packed in an array called process.argv (short for 'argument values'), so we can compute their length and access each argument by position.

- `process.argv.length:` number of arguments passed on the command line
- `process.argv[i]` : Returns the i-th argument. If we have used the command " `node program arg1 …`" then `process.argv[0]` contains the string '`node`', `process.argv[1]` contains the string '`program`', `process.argv[2]` contains '`arg1`', etc.

Note that **args=process.argv.slice(2) can be used** to discard 'node' and the program path, so that only the actual arguments to the application will remain in **args wrapped and accessible as elements of an array,** starting from position 0.

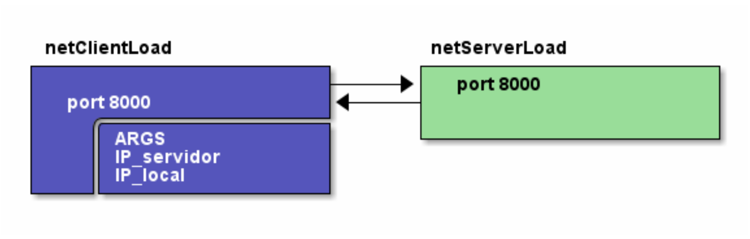
Part 2 of the activity. Check the load of the computer

You can embed the "getLoad()" function in the netServerLoad code, so that it gets its current load every time a client requests the service. Note that it only works on Linux:

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

As a curiosity, so that we understand the code of this function a little, we can observe what it does: This function reads data from the file /proc/loadavg. This pseudo-file contains information regarding the load of the Linux system. To provide the current load, the function filters the values of interest (adds a hundredth to them to avoid confusion between the value 0 and an error), and processes them by calculating a weighted average (weight 10 to the last minute load, weight 2 to the last 5 minutes, weight 1 to the last 15)

General description of both programs.



- **`netServerLoad`** server receives no command line arguments and will listen on a certain port (port 8000 for example).
- The **`netClientLoad client`** must receive the IP address of the server and its local IP as command line arguments. It will connect to the server at the server IP address and server port.
- Protocol: When the client sends a request to the server, including its own IP, the server calculates its load and returns a response to the client including the server's own IP and the load level calculated with the **getLoad function**.
- It may be a good idea for both programs to exchange data using JSON. Check the API “JSON.stringify()” and “JSON.parse()”
- You have to make sure that the client ends by getting the payload (e.g. with **`process.exit()`**) or by closing its connection.
- You can find out the IP from the portal web interface, or using the **ip addr command**, or **ifconfig**
- Complete both programs, put them on different machines collaborating with a partner (or connecting via ssh), and have them communicate via port 8000: **`netServerLoad`** should calculate the load as a response to each request received from the client, and **`netClientLoad`** should show the response on screen.

SESSION 1.3: REVERSE PROXY

An intermediary or proxy is a server that, when invoked by the client, redirects the request to a third party, and subsequently routes the final response back to the client.

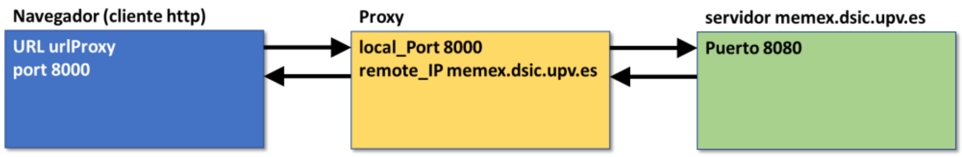
- From the client's point of view, it is a normal server (it hides the server that actually completes the service)
- May reside on a different machine than the client and server
- The broker can modify ports and addresses, but does not alter the body of the request or response.

This is the functionality that a redirector provides, such as an HTTP proxy, ssh gateway, or similar service. More information at http://en.wikipedia.org/wiki/Proxy_server

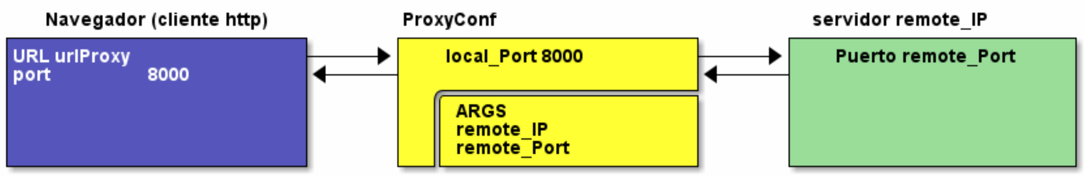
In this practice session we are going to work with 3 proxy versions. We give you the first version already implemented and you have to complete the other 2. At the end you have to answer the questions that we ask.

The three versions to consider are the following:

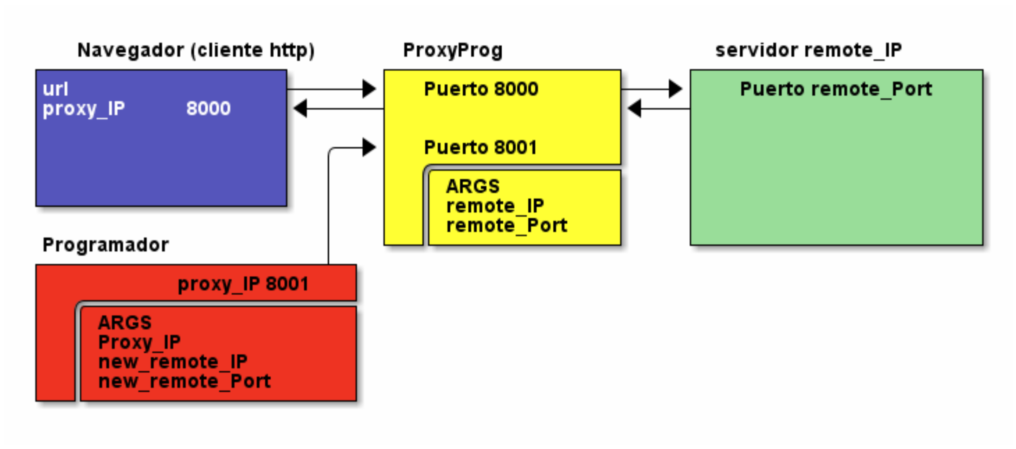
1. Basic Proxy (**Proxy**). When the http client (e.g. browser) contacts the proxy on port 8000, the proxy redirects the request to the memex web server (158.42.185.55 port 8080), and then returns the response from the server to the caller.



2. Configurable proxy (**ProxyConf**): instead of fixed remote IP and port in the code, you receive them as arguments on the command line



3. Programmable proxy (**ProxyProg**). The IP and port values of the server are initially taken from the command line, but are later modified by receiving messages from the programmer on port 8001



basic Proxy code : analyze it until you understand how it works

Program (Proxy.js)

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

const LOCAL_PORT  = 8000;
const LOCAL_IP  = '127.0.0.1';
const REMOTE_PORT = 80;
const REMOTE_IP = '158.42.4.23'; // www.upv.es

const server = net.createServer(function (socket) {
      const serviceSocket = new net.Socket();
      serviceSocket.connect(parseInt(REMOTE_PORT),
         REMOTE_IP, function () {
          socket.on('data', function (msg) {
               serviceSocket.write(msg);
          });
          serviceSocket.on('data', function (data) {
             socket.write(data);
          });
      });
}). listen ( LOCAL_PORT , LOCAL_IP );
console . log ( "TCP server accepting connection on port: " +
LOCAL_PORT );
```

Comments

Use one socket to talk to the client (socket) and another to talk to the server (serviceSocket)

1.- read a message (`msg`) from the client
2.- open a connection with the server
3.- write a copy of the message
4.- wait for the response from the server and return a copy to the client

1.- Basic proxy: In this first part you don't have to program anything. You have to check its operation. Check the proxy operation using a web browser pointing to http://proxy_address:8000/

Perform different tests and analyze the code provided to you.

You can also test the proxy to intermediate a netClientLoad client and a netServerLoad server.

2.- Configurable proxy: Based on the basic proxy, we should write a configurable proxy (ProxyConf.js). The only thing to do is parse the command line to get from it the address of the server to which the proxy will connect.

Once we have it done we can do several tests, the same as the tests we did in the previous version. Do at least these two:

- 1. Intermediate a WEB server. For example, intermediate access to the UPV server (we still use port 8000 as the local port for handling requests)

- 2. Intermediate access between netClientLoad and netServerLoad.
    - If ProxyConf is running on the same machine than netServerLoad and we have a port usage collision -> modify the netServerLoad code to use another port. You can also modify the netServerLoad code so that it takes as an argument the port it should listen on.
    - If the error "EADDRINUSE" appears when executing the program, it indicates that we are referencing a port that is already in use by another program

3. Programmable proxy :

Using the configurable proxy as a base, we create the programmable proxy (ProxyProg.js)

We have to implement the code of the programmer (programmer.js) and make some changes to the code of the new proxy. To implement the scheduler we can take the client program netClientLoad as a basis and make the necessary modifications. For its part, to make the new programmable Proxy receive requests from the programmer, we can take part of the server code that we have made (netServerLoad)

We have to complete both programs so that the following aspects are considered:

- The programmer must receive the IP address of the proxy, and the new IP and port values corresponding to the server, on the command line. With the IP of the proxy and the default port of the proxy (port 8001), it will contact the proxy to send it the data from the remote server.

- The scheduler will encode the values and send them as a message to the proxy, after which it terminates. For its part, the proxy will receive this message to update the address of the server it will contact from that moment on.

- The `programmer.js` should send messages with content like the following:

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

You can experiment using two different WEB servers as servers and check how the programmer modifies the destination server. You could also use 2 netServerLoad servers and 1 netClientLoad client, plus the scheduler.

Questions

1. What advantages do you observe in the interaction between a client and a server when using JSON compared to not using it?
2. Imagine a new proxy that has a pool of servers to send requests to. How could we get this new proxy to take care of sending requests to the server that is least loaded at any given time? Discuss how you would develop this new proxy.