

TSR: Seminar 1. Exercises

ACTIVITY 1

GOAL: To delve in the JavaScript argument passing mechanisms.

EXERCISE: Please consider the following program¹ and answer the questions being shown afterwards:

```
1 function table(x) { // Prints column x of a (1..10) multiplication table
2     for (var j=1; j<11; j++)
3         console.log("%d * %d = %d", x, j, x*j);
4     console.log("");
5 }
6
7 function allTables() {
8     for (var i=1; i<11; i++)
9         table(i);
10 }
11
12 table(5, 4, 1);
```

- a) Describe the output provided by that program. Justify whether the usage of multiple arguments in the call made in line 12 has any effect or not.

¹ All program files listed or mentioned in this document can be downloaded from the PoliformaT site. They are maintained in the "acts.zip" file.

- b) Let us assume that the original line 12 is replaced with the following one. In that case, which is the output of the resulting program? Why?

12	<code>table(table(2));</code>
----	-------------------------------

- c) Let us assume, again, that the original line 12 is replaced with the following one. In this case, which is the output of the resulting program? Why?

12	<code>allTables(table(30),table(20),table(10));</code>
----	--

- d) Considering your answers to the previous questions, please justify whether JavaScript tolerates additional arguments in function calls and whether using those arguments may have any effect.

ACTIVITY 2

OBJECTIVE: Improve your ability in JavaScript event-driven programming.

EXERCISE: Considering the following program:

```
1 var ev = require('events');
2 var emitter = new ev.EventEmitter;
3 var e1 = "print";
4 var e2 = "read";
5 var books = [ "Walk Me Home", "When I Found You", "Jane's Melody", "Pulse" ];
6
7 // Constructor for class Listener.
8 function Listener(n1,n2) {
9   this.num1 = 0;
10  this.name1 = n1;
11  this.num2 = 0;
12  this.name2 = n2;
13 }
14
15 Listener.prototype.event1 = function() {
16   this.num1++;
17   console.log( "Event " + this.name1 + " has happened " + this.num1 + " times." );
18 }
19
20 Listener.prototype.event2 = function(a) {
21   console.log( "Event " + this.name2 + " (with arg: " + a + ") has happened " +
22     ++this.num2 + " times." );
23 }
24
25 // A Listener object is created.
26 var lis = new Listener(e1,e2);
27
28 // Listener is registered on the event emitter.
29 emitter.on(e1, function() {lis.event1({});});
30 emitter.on(e2, function(x) {lis.event2(x)});
31 // There might be more than one listener for the same event.
32 emitter.on(e1, function() {console.log("Something has been printed!!");});
33
34 // Auxiliary function for generating e2.
35 var counter=0;
36 function generateEvent2() {
37   emitter.emit(e2,books[counter++ % books.length]);
38 }
39
40 // Generate the events periodically...
41 // First event generated every 2 seconds.
42 setInterval( function() {
43   emitter.emit(e1);
44 }, 2000 );
45 // Second event generated every 3 seconds.
46 setInterval( generateEvent2, 3000 );
```

Note that it extends the example shown in Section 3.2 of this seminar. Besides the event emitter, this new program implements a “listener” that provides a method for each one of the events. This program also shows some other aspects:

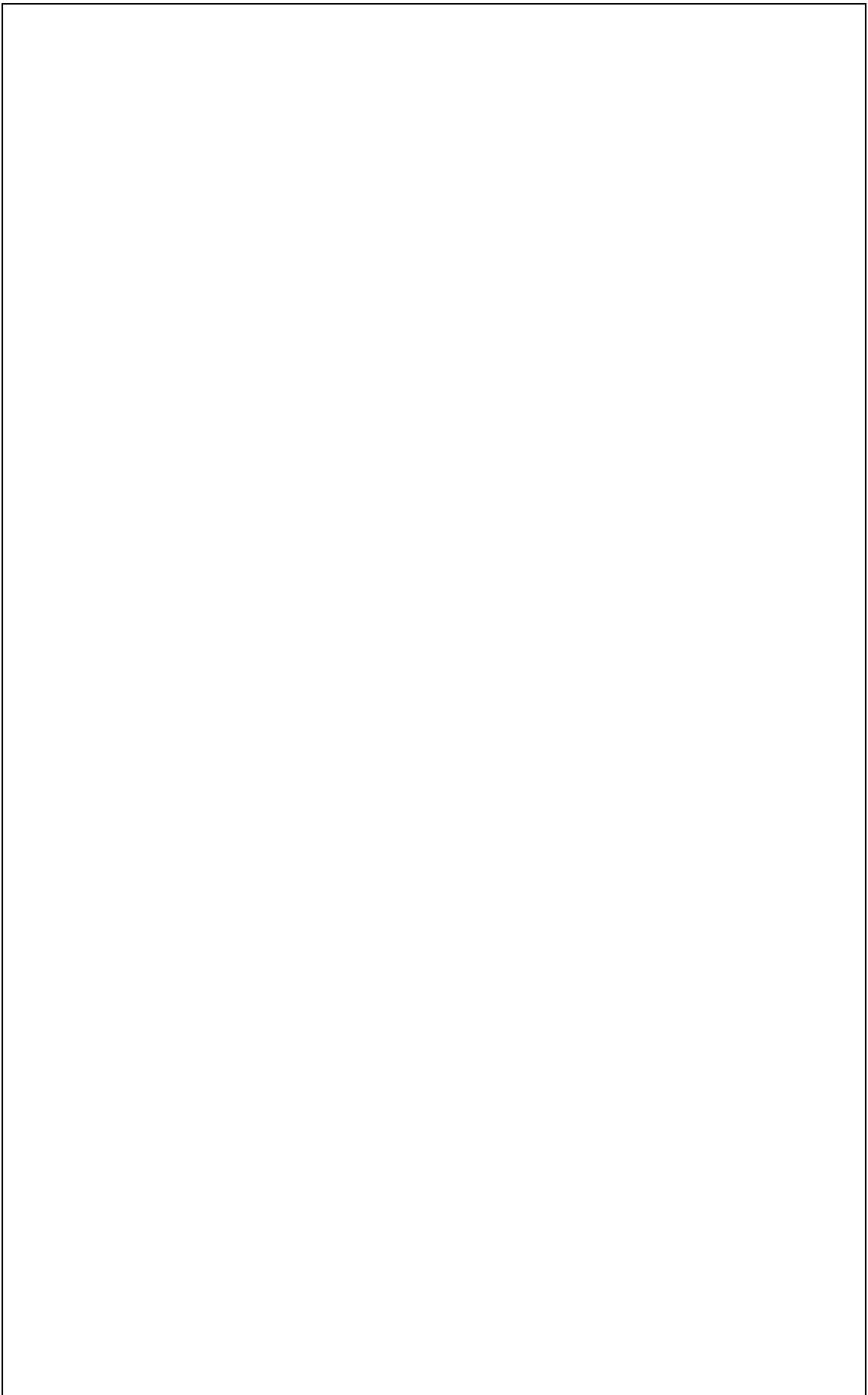
- The “read” event is generated with an additional argument (title to be read). The sentences needed to this end are shown in these lines:
 - 46: Frequency of this event (1 event per 3 seconds).
 - 35-38: Declaration of a variable (“counter”) that maintains the number of generated events and, depending on it, selects the message to be used as an argument.
 - 20-23, 30: The “listener” for this event needs a parameter.
- The Listener object, whose constructor is in lines 8 to 13, holds now the amount of times that it has processed each kind of event. Those counters are increased in each one of the functions as “listener” of an event (in lines 16 and 22, respectively).

Taking this program as a base, please develop another extended version where:

- Three events should be emitted:
 - “one”: Every three seconds. Without arguments.
 - “two”: Initially, every two seconds. Without arguments.
 - “three”: Every ten seconds. Without arguments.
- There should be a Listener object with a method for each emitted event. When an event is thrown, Listener should do the following:
 - “one”: Write the string “Active listener.” to its standard output.
 - “two”: Write the string “Event two.” to its standard output when the number of “two” events is greater than the number of “one” events. Otherwise, it should write “I have received more events of type ‘one’.”.
 - “three”: Write the string “Event three.” to its standard output. Additionally, on each execution of this handler, the length of the event “two” interval will be doubled, until such length were greater than 9 seconds. Once this happens, the “two” events should happen every 9 seconds.

The “setInterval()” operation returns an object that should be used as the single argument of “clearInterval()”. In order to modify the frequency of an event, please use “clearInterval()” before setting the new frequency.





ACTIVITY 3

GOAL: To understand and use closures and how to pass functions as arguments in JavaScript.

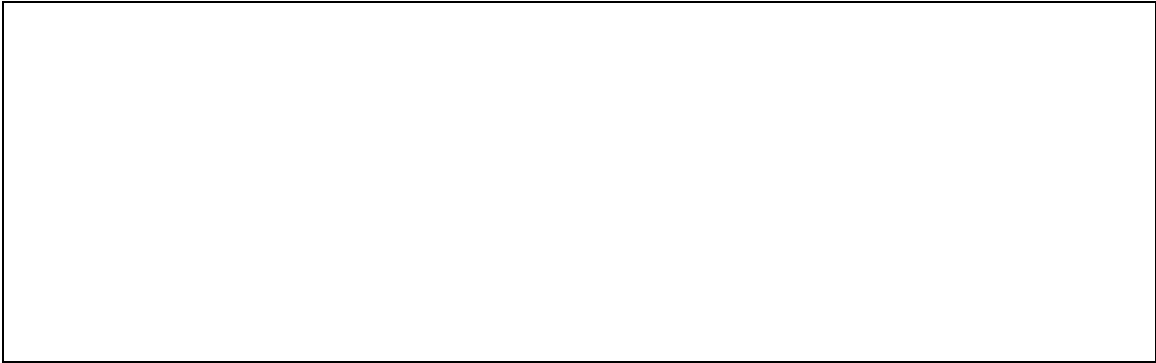
EXERCISE: Consider this program:

```
1 function a3(x) {  
2     return function(y) {  
3         return x*y;  
4     };  
5 }  
6  
7 function add(v) {  
8     var sum=0;  
9     for (var i=0; i<v.length; i++)  
10         sum += v[i];  
11     return sum;  
12 }  
13  
14 function iterate(num, f, vec) {  
15     var amount = num;  
16     var result = 0;  
17     if (vec.length<amount)  
18         amount=vec.length;  
19     for (var i=0; i<amount; i++)  
20         result += f(vec[i]);  
21     return result;  
22 }  
23  
24 var myArray = [3, 5, 7, 11];  
25 console.log(iterate(2, a3, myArray));  
26 console.log(iterate(2, a3(2), myArray));  
27 console.log(iterate(2, add, myArray));  
28 console.log(add(myArray));  
29 console.log(iterate(5, a3(3), myArray));  
30 console.log(iterate(5, a3(1), myArray));
```

Run the program and explain the result of the execution in each of the following lines:

a) line 25.

b) line 26.

A large, empty rectangular box with a thin black border, intended for a drawing or answer corresponding to line 26.

c) line 27.

A large, empty rectangular box with a thin black border, intended for a drawing or answer corresponding to line 27.

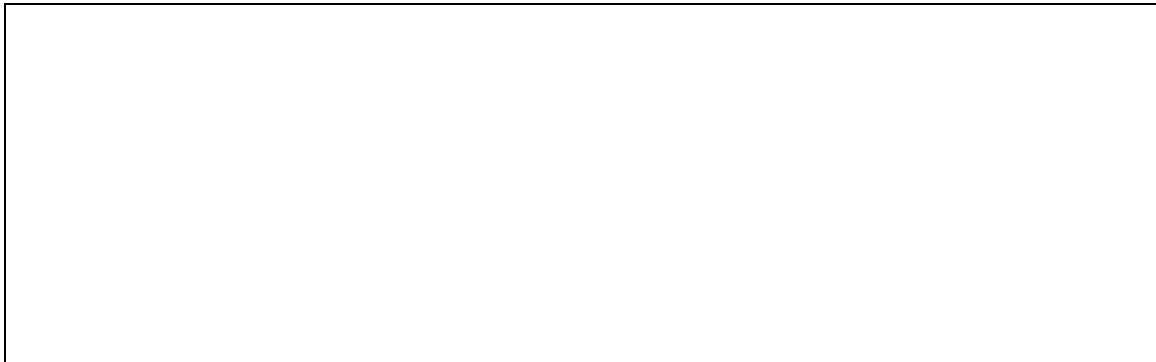
d) line 28.

A large, empty rectangular box with a thin black border, intended for a drawing or answer corresponding to line 28.

e) line 29.

A large, empty rectangular box with a thin black border, intended for a drawing or answer corresponding to line 29.

f) line 30.



ACTIVITY 4

GOAL: To extend the minimal HTTP web server presented in the Seminar.

EXERCISE: In one of the presentation sections, we have shown this program:

```
1 var http = require('http');
2 http.createServer(function (request, response) {
3   // response is a ServerResponse.
4   // Its writeHead() method sets the response header.
5   response.writeHead(200, {'Content-Type': 'text/plain'});
6   // The end() method is needed to communicate that both the header
7   // and body of the response have already been sent. As a result, the response can
8   // be considered complete. Its optional argument may be used for including the last
9   // part of the body section.
10  response.end('Hello World\n');
11  // listen() is used in an http.Server in order to start listening for
12  // new connections. It sets the port and (optionally) the IP address.
13 }).listen(1337, "127.0.0.1");
14 console.log('Server running at http://127.0.0.1:1337/');
```

This server replies with the “Hello World” string to the browsers that access the <http://127.0.0.1:1337/> URL in the local computer. It does not return anything else.

A web server should be able to return the contents of HTML files specified in the URL being used. To achieve this functionality other mechanisms are needed:

- The first argument (“request” in this example: a ClientRequest object) in the “callback” used in `http.createServer()` has a “url” property. It maintains the fragment of the URL that follows the computer name and port. Thus, for instance, when a user introduces in the browser this string:
<http://127.0.0.1:1337/dir1/page.html>
the `request.url` property holds the string “dir1/page.html”.
- Module “fs” provides multiple operations for accessing the file system. In this example, the server should read the contents of file “page.html” in directory “dir1”. This can be achieved using a code similar to (assuming a “`var fs=require('fs');`” at the beginning of the program):

```
fs.readFile( "dir1/page.html", function (error,content){...} );
```

But the name being used as a first argument should be an absolute pathname. Moreover, the second argument is a “callback” function whose first parameter communicates the errors and the second one provides the contents of the file being read.

In this case, when there is any error, the “**response**” object should be filled using:

```
response.writeHead(404);  
response.write('not found');
```

The error identifier to be returned in the ServerResponse header should be 404 (File not found). The second sentence provides the error message text. On the other hand, when no errors occur, “**response**” should be filled with:

```
response.writeHead(200);  
response.write(content);
```

A value 200 in the header means that no error arose. The second sentence dumps the file contents into the ServerResponse. Note that “**content**” received the contents of the file in the `readFile()` operation.

- In order to build an absolute pathname, we should use the `join()` operation of the `PATH` module. To this end, there is a “global” property “`__dirname`” that stores the absolute pathname of the current working directory. Thus, we need a “`var path=require('path');`” in the beginning of our program and:

```
path.join(__dirname, request.url)
```

in order to return that absolute pathname.

- In most web servers, when the user writes only a server address in a URL (without any pathname or file name) the server returns the contents of a root “index.html” file. Please, write the appropriate code to implement this same behaviour.
- Moreover, a URL may specify only the address of a web site (<https://intranet.upv.es>) or the full path of any of the pages from that site (<http://www.upv.es/organizacion/escuelas-facultades/index-en.html>), but also some other information as, for instance, some query parameters. Consider: https://intranet.upv.es/pls/soalu/sic menu.Personal?P_IDIOMA=c; in that page the query information is “P_IDIOMA=c”. The `parse()` method from the “url” module provides access to the components of an URL. The “query” property holds the information for query parameters.

For instance (assuming a previous statement “`var url=require('url');`”):

```
url.parse(https://intranet.upv.es/pls/soalu/sic menu.  
Personal?P\_IDIOMA=c).query
```

returns “P_IDIOMA=c”.

- The string that maintains query parameters may be much more complex (e.g., consider the following URL fragment:
[http://www.booking.com/searchresults.es.html?src=index& ...
&ss=Valencia&checkin_monthday=1&checkin_year_month=2015-
8&checkout_monthday=2&checkout_year_month=2015-8& ...](http://www.booking.com/searchresults.es.html?src=index&...&ss=Valencia&checkin_monthday=1&checkin_year_month=2015-8&checkout_monthday=2&checkout_year_month=2015-8&...)).

So, it is convenient to know and use some other methods for extracting concrete parameters from the query string. This is done by the `parse()` method from the “`querystring`” module.

For instance (assuming a “`var qs = require(“querystring”);`”):

```
qs.parse(http://www.booking.com/searchresults.es.html?src=index& ...  
&ss=Valencia&checkin\_monthday=1&checkin\_year\_month=2015-8& ...).ss
```

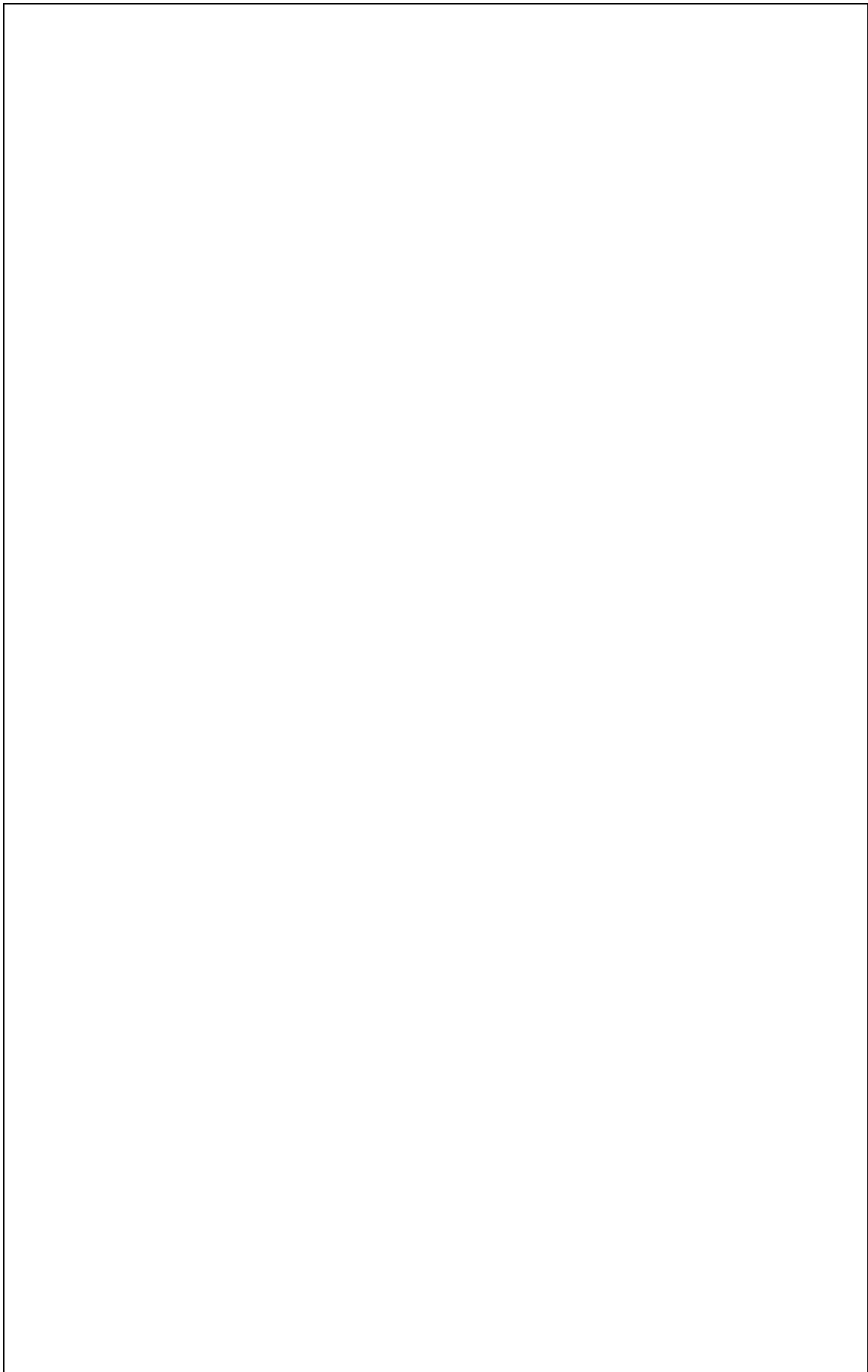
returns “Valencia”.

- In order to illustrate the basic usage of query parameters, please consider the following simple server:

```
1 var http = require('http');  
2 var url = require('url');  
3 var qs = require('querystring');  
4 http.createServer( function(request,response) {  
5   var query = url.parse(request.url).query;  
6   var info = qs.parse(query).info;  
7   var x = '2015';  
8   var y = '1492';  
9   response.writeHead(200, {'Content-Type':'text/plain'});  
10  switch( info ) {  
11    case 'x': response.end('Value = ' + x); break;  
12    case 'y': response.end('Value = ' + y); break;  
13  }  
14 }).listen('1337');
```

Please extend the program shown in the first box of this exercise in order to manage a parameter called “`query`” in the URL and, depending on its value, return different data:

- If the “`query`” value is “`time`”, the server returns the current date and time.
- If the “`query`” value is “`dir`”, the server returns the directory name where the server has been started and a list with all the names of the files located in that directory.
- Any other value for the “`query`” parameter should be interpreted as a file name. If that file exists in that directory, the server reads it and returns its contents. Otherwise, it returns an error message saying that the file does not exist.



ACTIVITY 5 (Optional since it requires promises)

OBJECTIVE: To understand that async callbacks are sometimes sync. Usage of promises.

EXERCISE: There is just one thread in Node.js. This implies we do not have to worry about protecting shared variables with mutexes or other concurrency control mechanisms.

However, there are cases in which we need to be careful.

A good practice to reason about the logic of an asynchronous program is to consider ALL callbacks happening in a future turn from the one executing the code passing them.

Consider this program:

```
1 fs = require('fs');
2 path = require('path');
3 os = require('os');
4 var rolodex={};
5
6 function contentsToArray(contents) {
7   return contents.split(os.EOL);
8 }
9 function parseArray(contents,pattern,cb) {
10   for(var i in contents) {
11     if (contents[i].search(pattern) > -1)
12       cb(contents[i]);
13   }
14 }
15
16 function retrieve(pattern,cb) {
17   fs.readFile("rolodex", "utf8", function(err,data){
18     if (err) {
19       console.log("Please use the name of an existant file!!");
20     } else {
21       parseArray(contentsToArray(data),pattern,cb);
22     }
23   });
24 }
25
26 function processEntry(name, cb) {
27   if (rolodex[name]) {
28     cb(rolodex[name]);
29   } else {
30     retrieve( name, function (val) {
31       rolodex[name] = val;
32       cb(val);
33     });
34   }
35 }
36
37 function test() {
38   for (var n in testNames) {
```

```

39     console.log ('processing ', testNames[n]);
40     processEntry(testNames[n], function generator(x) {
41         return function (res) {
42             console.log('processed %s. Found as: %s', testNames[x], res);
43         }}(n))
44     }
45 }
46
47 var testNames = ['a', 'b', 'c'];
48 test();

```

When we run it², we should expect the following output:

```

processing a
processing b
processing c
processed a...
processed b...
processed c...

```

ALL “processed” messages appear after ALL “processing” messages, as per our expectations (callbacks seem to be called in a future turn).

Consider, however this variation:

Replace line 4 in the previous program (`var rolodex={};`) with the following one:

```

4 var rolodex={a: "Mary Duncan 666444888"};

```

The output we would get now is this:

```

processing a
processed a...
processing b
processing c
processed b...
processed c...

```

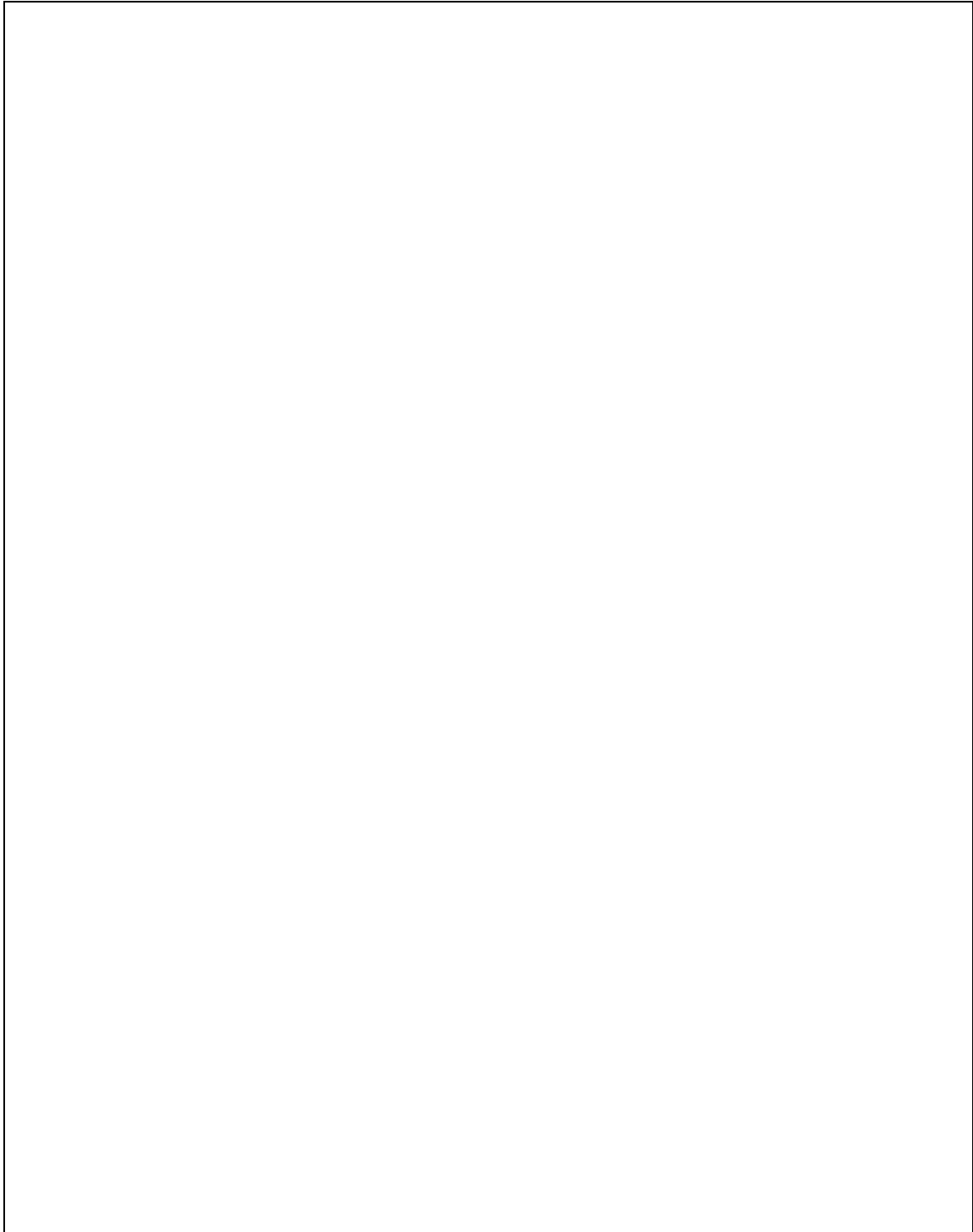
Notice, that now not ALL “processed” messages come after ALL “processing” messages. The reason is that one of the callbacks has been executed IN THE SAME turn as the call passing it.

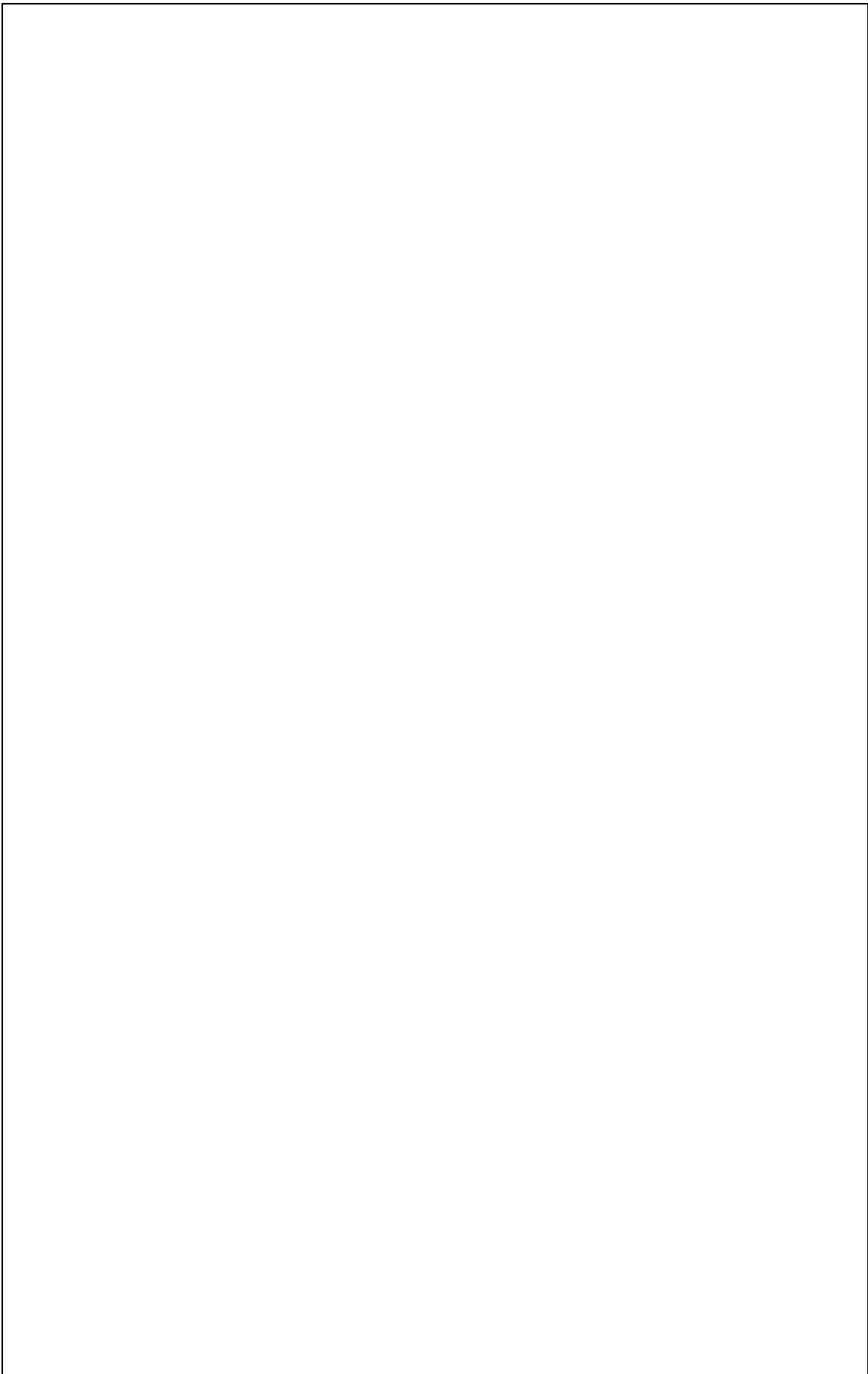
² In order to correctly execute this program we need a file named “rolodex” in the same folder. That file should contain some lines of text. In those lines, the program looks for the strings held in the “testNames” array.

Depending on the situation, this may introduce hard-to-see problems, if the code setting up the callbacks and one or more callbacks interfere with the same state of the process, potentially rendering it inconsistent.

This will always happen if the callbacks rely on the main code preparing their context before they start execution.

Modify this program, using promises, to ensure that the expectations about order of execution are met.





ACTIVITY 6

GOAL: To use callbacks and closures appropriately.

EXERCISE: In Node.js the “fs” module provides multiple operations to manage files.

Write a program that accepts a variable amount of file names from the command line and that writes to screen the name of the largest file in that set, and its file length. To this end, please use the `fs.readFile()` function from that module, whose documentation can be found in https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback. Do not use any of the synchronous variants for that function or any other functions from that module.

In order to manage the received arguments from the command line, you need the `process.argv` array (https://nodejs.org/api/process.html#process_process_argv).

In case of receiving multiple arguments (i.e., the regular case), consider using closures in order to ensure that your callback accesses the correct slot in the `process.argv[]` array.

ACTIVITY 7 (Parts b and c are optional since they need promises)

GOAL: To delve in the conversion of callbacks into promises.

EXERCISE: The program that is shown in the following box implements a text file server that provides three operations to its clients: UPLOAD, DOWNLOAD and REMOVE. This file and the programs needed to request those operations (Act7upload.js, Act7download.js and Act7remove.js) may be downloaded from PoliformaT where you may also find a version of this server program implemented using promises (Act7serverPromise.js).

```
1 // File Act7server.js
2 // This is a simple file server that understands these requests:
3 // - DOWNLOAD: Receives the name of a file whose contents should be returned in the reply message.
4 // - UPLOAD: Receives the name and contents of a file that should be stored in the server directory.
5 // - REMOVE: Removes a given file if its name exists in the current directory.
6 // The port number to be used by the server should be passed as an argument from the command line.
7 // Requests and replies are received and sent using TCP channels.
8
9 var net = require('net');
10 var fs = require('fs');
11 // Default port number.
12 var port = 9000;
13
14 // Check command line arguments.
15 if (process.argv.length > 2)
16     port = process.argv[2];
17
18 // Create the server.
19 var server = net.createServer( function(c) {
20     console.log("Server connected!");
21     // Manage end events.
22     c.on("end", function() {
23         console.log("Server disconnected!");
24     });
25     // Manage message receptions.
26     c.on("data", function(m) {
27         // Get the message object.
28         var msg = JSON.parse(m);
29         switch (msg.type) {
30             // Download management.
31             case 'DOWNLOAD':
32                 // Read the file named in the request message.
33                 fs.readFile(msg.name,
34                     // readFile() callback.
35                     function(err,result) {
36                         var msg2 = {};
37                         // If error, print an error message at the server's
38                         // console and return an error reply.
39                         if (err) {
40                             console.log( "%s trying to apply %s on %s",
41                                 err, msg.type, msg.name );
42                             msg2 = { type:'ERROR', data:err.code };
43                             // Otherwise, print a message and return an OK reply.
44                         } else {
45                             console.log( "%s successfully applied on %s",
46                                 msg.type, msg.name );
47                             msg2 = { type:'OK', data: result.toString() };
48                         }
49                         // Build and send the reply message.
50                         var m2 = JSON.stringify(msg2);
51                         c.write(m2);
```

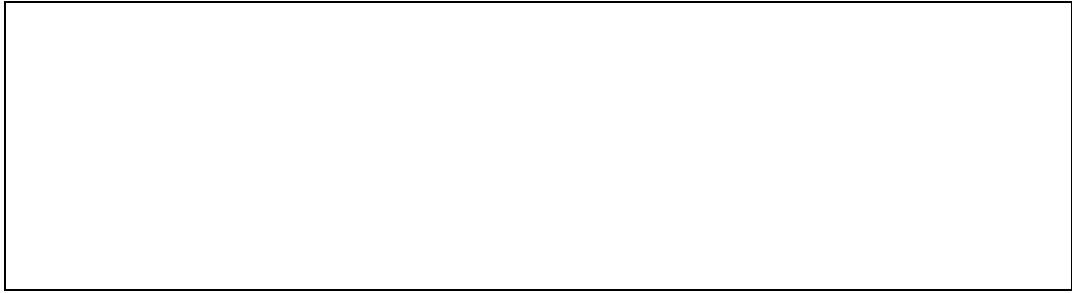
```

52         } // End of callback.
53     ); // End of writeFile() call.
54     break;
55     // Upload management.
56     case 'UPLOAD':
57         // Write the received file contents in the local directory.
58         fs.writeFile(msg.name, msg.data,
59             // writeFile() callback.
60             function(err,result) {
61                 var msg2 = {};
62                 // If error, print an error message at the server's
63                 // console and return an error reply.
64                 if (err) {
65                     console.log( "%s trying to apply %s on %s",
66                                 err, msg.type, msg.name );
67                     msg2 = { type:'ERROR', data:err.code };
68                     // Otherwise, print a message and return an OK reply.
69                 } else {
70                     console.log( "%s successfully applied on %s",
71                                 msg.type, msg.name );
72                     msg2 = { type:'OK' };
73                 }
74                 // Build and send the reply message.
75                 var m2 = JSON.stringify(msg2);
76                 c.write(m2);
77             } // End of callback.
78     ); // End of writeFile() call.
79     break;
80     // Remove management.
81     case 'REMOVE':
82         // Remove the file whose name is received in this message.
83         // Not implemented yet!!
84         break;
85     };
86 });
87 // Manage errors on the current connection.
88 c.on("error", function(e) {
89     console.log("Error: %s", e);
90 });
91 }); // End of createServer();
92
93 // Listen to the given port.
94 server.listen(port, function() {
95     console.log("Server bound to port %s", port );
96 });
97
98 // If any error arises in this socket, report it.
99 server.on("error", function(e) {
100     console.log("Server error: %s", e.code);
101 });

```

Please complete the following tasks:

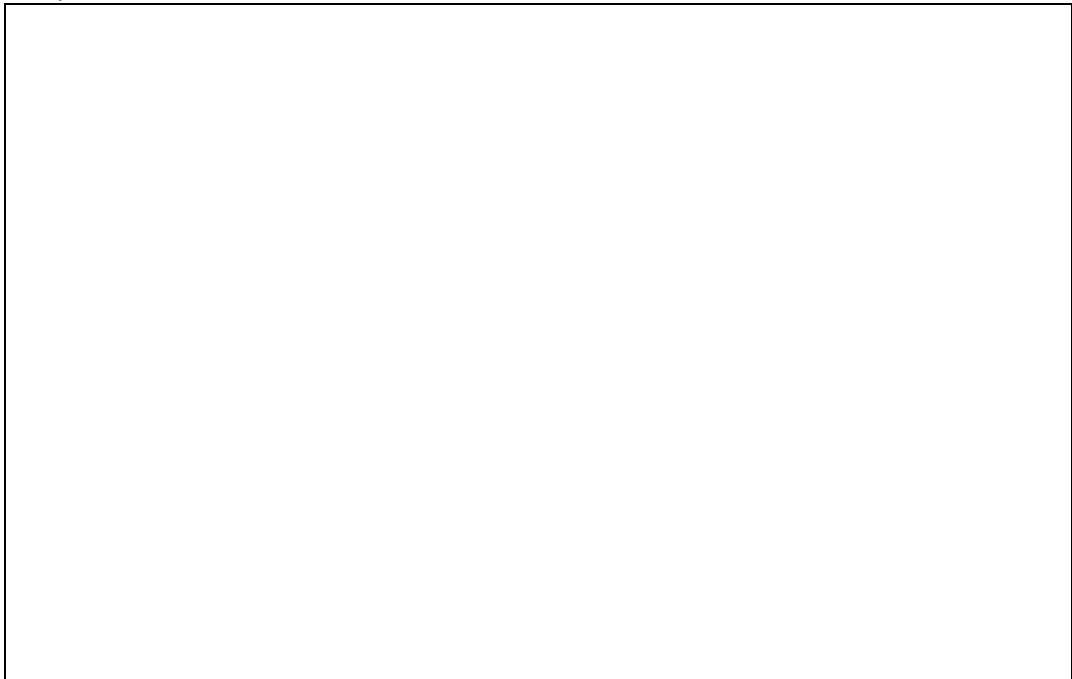
- a) To insert the code fragments needed to correctly implement the REMOVE operation. Please specify between which pairs of lines should be placed each code fragment.



- b) Implement and explain an equivalent solution for the program based on promises (Act7serverPromise.js).



- c) Explain in which program you have needed more work to develop your solution and why.



ACTIVITY 8 (Optional since it requires promises)

GOAL: To delve in the conversion of callbacks into promises.

EXERCISE: Taking as a base your complete solution based on promises to Activity 7, adapt such solution creating the promises using their constructor, instead of using the `promisify()` method.

ACTIVITY 9

GOAL: To execute JavaScript code in an interactive way on a remote server using the “net” and “repl” modules.

EXERCISE: The REPL (Read-Eval-Print Loop) module represents the Node shell. This shell can be directly activated on the command line writing:

```
$ node
```

Additionally, the “repl” module (when it is used from a Node program) makes possible the invocation of that shell, executing JavaScript statements in an interactive way.

Consider the following program:

```
1  /* repl_show.js */
2
3  var repl = require('repl')
4  var f = function(x) {console.log(x)}
5  repl.start('$> ').context.show = f
```

An example of its execution could be:

```
> node repl_show
$> show(5*7)
35
undefined
$> show('juan '+'luis')
juan luis
undefined
```

As it can be seen, it has implemented a “show” function that is equivalent to the native “console.log()” function from node. That execution also shows that “undefined” values are printed to the standard output. This can be avoided assigning a value to one property in that module (take a look at <https://nodejs.org/api/repl.html> to this end).

The “repl” shell may be also invoked remotely. For instance, consider the following repl_client.js and repl_server.js programs:

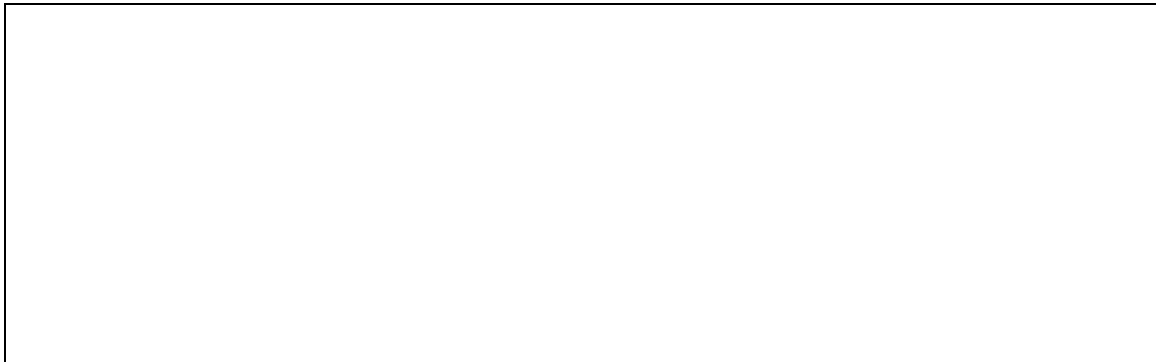
```
1  /* repl_client.js */
2
3  var net = require('net')
4  var sock = net.connect(8001)
5
6  process.stdin.pipe(sock)
7  sock.pipe(process.stdout)
```

```
1  /* repl_server.js */
2
3  var net = require('net')
4  var repl = require('repl')
5
6  net.createServer(function(socket){
7    repl
8    .start({
9      prompt: '>',
10     input: socket,
11     output: socket,
12     terminal: true
13   })
14   .on('exit', function(){
15     socket.end()
16   })
17 }).listen(8001)
```

Please analyse the functionality of both programs, reading the API of those two modules, starting those programs and using the remote shell from the client program.

a) Explain the functionality of both processes, client and server, describing the information flow and discussing which of those processes directly executes the statements.

b) If we write “`console.log(process.argv)`” in the client terminal... what is printed?, why?

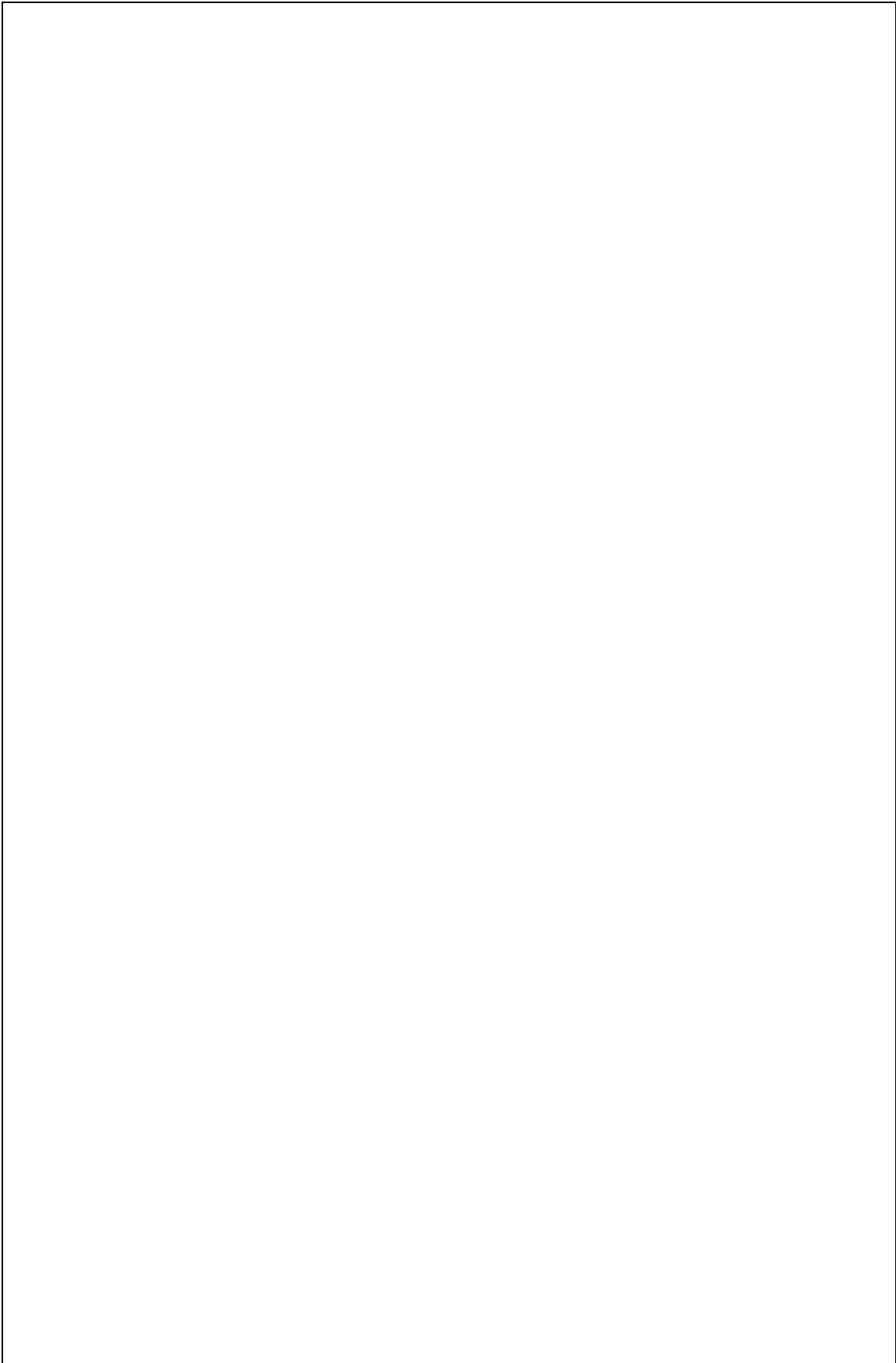


c) Modify the “`repl_server.js`” program to be able to accept and run as shown the interactive session in the `repl_client` process depicted in the following figure:

```
ca. Node.js command prompt - node repl_client
$node> factorial(4)
24
$node> factorial(12)
479001600
$node> fibonacci(6)
13
$node> fibonacci(32)
3524578
$node> var logBase3 = logaritmo(3)
$node> logBase3
[Function]
$node> logBase3(81)
4
$node> var logBase2 = logaritmo(2)
$node> logBase2(4096)
12
$node> console.log('Hola')
Hola
$node> fibonacci(17)
2584
$node> leeFichero("repl_show.js")
'var repl = require(\'repl\')\r\nvar f = function(x) {consol
e.log(x)}\r\nrepl.start(\'>\').context.show = f'
$node> logBase2(512)
9
$node> _
```

No modification should be applied to the client program. In the server, you should update the relevant parameters (modification of its prompt, avoidance of any “undefined” printing, colour activation) and also include in its program several other functions:

- **factorial**: a function that, given an integer n , returns $n!$ as its result.
- **fibonacci**: a function that, given an integer n , returns the n -th term of the Fibonacci series as its result.
- **logaritmo**: a function that, given an integer n , returns a function for computing the logarithm in base n as its result (take a look at the second section, Closures, from the Student Guide of this seminar for additional details on this).
- **leeFichero**: this function should be a wrapper to the “`readFileSync()`” function from the “`fs`” module. If the file whose name is given as its argument exists, it returns its contents. Otherwise, it returns the error `ENOENT`.



ACTIVITY 10

GOAL: To use the “Socket.IO” module in order to build a networked multi-user application.

EXERCISE: Please consider the following drawing application that consists of an HTML file (“index.html”) and a JavaScript file (“script.js”). These files are:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <title>Node.js Multiuser Drawing Game</title>
6   </head>
7   <body>
8     <div id="cursors">
9       <!-- The mouse pointers will be created here -->
10    </div>
11    <canvas id="paper" width="800" height="400"
12      style="border:1px solid #000000;">
13      Your browser needs to support canvas for this to work!
14    </canvas>
15    <hgroup id="instructions">
16      <h1>Draw anywhere inside the rectangle!</h1>
17      <h2>You will see everyone else who's doing the same.</h2>
18      <h3>Tip: if the stage gets dirty, simply reload the page</h3>
19    </hgroup>
20    <!-- JavaScript includes. -->
21    <script src="http://code.jquery.com/jquery-1.8.0.min.js"></script>
22    <script src="script.js"></script>
23  </body>
24 </html>
```

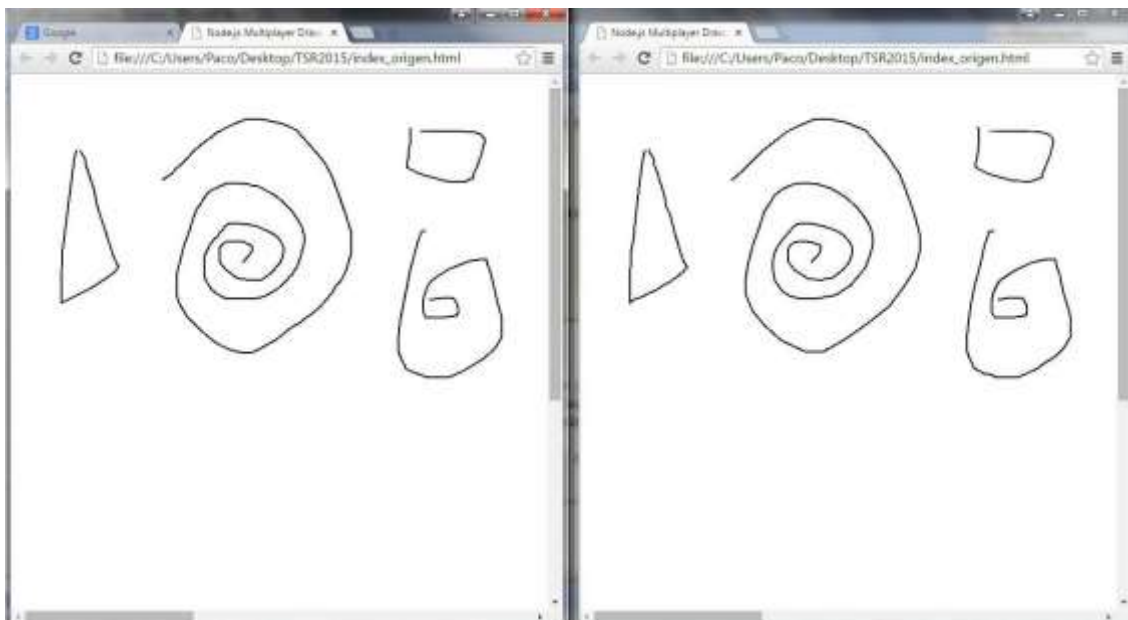
```
1 $(function(){
2   // This demo depends on the canvas element
3   if(!('getContext' in document.createElement('canvas'))){
4     alert('Sorry, it looks like your browser does not support canvas!');
5     return false;
6   }
7   var doc = $(document),
8       win = $(window),
9       canvas = $('#paper'),
10      ctx = canvas[0].getContext('2d'),
11      instructions = $('#instructions'),
12      id = Math.round($.now()*Math.random()), // Generate an unique ID
13      drawing = false, // A flag for drawing activity
14      clients = {},
15      cursors = {},
16      prev = {};
17   canvas.on('mousedown',function(e){
18     e.preventDefault();
19     drawing = true;
```

```

20     prev.x = e.pageX;
21     prev.y = e.pageY;
22   });
23   doc.bind('mouseup mouseleave',function(){
24     drawing = false;
25   });
26   doc.on('mousemove',function(e){
27     // Draw a line for the current user's movement
28     if(drawing){
29       drawLine(prev.x, prev.y, e.pageX, e.pageY);
30       prev.x = e.pageX;
31       prev.y = e.pageY;
32     }
33   });
34   function drawLine(fromx, fromy, tox, toy){
35     ctx.moveTo(fromx, fromy);
36     ctx.lineTo(tox, toy);
37     ctx.stroke();
38   }
39 });

```

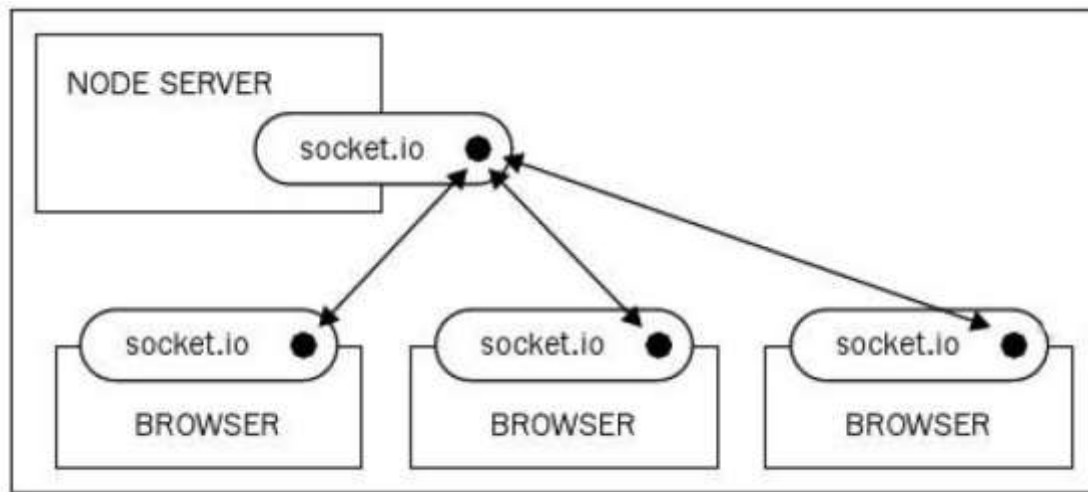
This application needs a canvas object and is driven by mouse-related events. It is a single-user drawing application. Our goal is to extend it, allowing that multiple users load it in their browsers, sharing the same drawing board (i.e., the lines drawn by a user will be seen by all users). The next figure shows an example with two users, but the solution should not limit the maximal number of users.



We should use the “Socket.IO” module to this end. Since it is not a standard JavaScript module, we need to install it using the “npm” command, as follows:

```
npm install socket.io
```

“Socket.IO” provides bidirectional sockets and may be easily integrated in Internet browsers. The overall design for the proposed application is shown in this figure:



Each browser interacts with a central server node, sending its drawing actions to that server and receiving the actions applied by the remaining users. Thus, the task of the server consists in forwarding the messages sent by each client to all remaining clients. This is an example of “broadcasting” communication.

When we use the “Socket.IO” module in order to forward messages, we should set the “broadcast” flag in the calls to the “emit()” and “send()” methods. For instance, the following code snippet corresponds to a server that forwards messages to all known sockets except that used for receiving the message to be forwarded:

```
1 var io = require('socket.io').listen(8080);
2
3 io.on('connection', function (socket) {
4   socket.broadcast.emit('user connected');
5 });
```

In our solution to this activity, we should implement a new file (that of the server process) and update the client JavaScript file “script.js” and the HTML file that loads it. In such “index.html” file we need to add the following line in the “includes” section:

```
<script src="socket.io.js"></script>
```

...assuming that “socket.io.js” and “index.html” are placed in the same directory.

The server should listen to a given port, to which we will connect the “socket.io” sockets of each client script. The extensions to be applied to the “script.js” file are:

- Declare a socket and connect it to the server.
- Modify the document (“doc” variable) “mousemove” callback to, besides plotting the line for the local user, send that drawing information through the socket. The information to be transmitted is:

```
{ 'x': e.pageX, 'y': e.pageY, 'drawing': drawing, 'id': id }
```

And the event associated to this sending could be “mousemove” (assuming that we have chosen this same event name in the server).

- Besides this, the client socket should listen to server messages. These messages correspond to the drawing notifications sent by other users. They should have an event name; e.g., “moving”. The callback associated to this “moving” event in the client socket should adequately process the received data (such incoming “data” object should have “x”, “y”, “drawing” and “id” properties, as we have shown before).

In this callback we should check first whether the incoming data belong to a new user. In that case, it should be registered:

```
if ( !(data.id in clients) )  
    cursors[data.id] = $('<div class="cursor">').appendTo('#cursors');
```

Next, we should draw the line corresponding to the received data (from the last position for that user, “clients[data.id]”, if any, to its current position, “data”):

```
if ( data.drawing && clients[data.id] )  
    drawLine(clients[data.id].x, clients[data.id].y, data.x, data.y);
```

Finally, the callback updates the user state:

```
clients[data.id] = data;
```



In the server, we should write the code being needed to:

- Use a “socket.io” socket that listens to the port assumed in the client script.
- For that “io.sockets” object, implement a callback that manages the “connection” event originated by any client.
- In this callback, for the identified client socket, implement another callback that manages the “mousemove” event.
- As an answer to this “mousemove” event, we should send (i.e., broadcast) a message with the “moving” event and the same data received in the “mousemove” event.

Please, implement this server:

ACTIVITY 11

GOAL: Check whether there is any concurrency in asynchronous servers.

EXERCISE: The following programs are a client and a server:

```
1 // Client code: Act11client.js
2 var net = require('net');
3
4 if (process.argv.length != 3) { // Check that an argument is given.
5     console.log('An argument is required!!');
6     process.exit();
7 }
8
9 var info = process.argv[2]; // Obtain the argument value.
10
11 var client = net.connect({port: 9000}, // The server is in our same machine.
12     function() { // 'connect' listener
13         console.log('client connected');
14         console.log('sent:', info);
15         client.write(info+'\n'); // This is sent to the server.
16     });
17
18 client.on('data', function(data) {
19     console.log('received:\n' +data.toString()); // Write the received data to stdout.
20     client.end(); // Close this connection.
21 });
22
23 client.on('end', function() {
24     console.log('client disconnected');
25 });
```

```
1 // Server code: Act11server.js
2 var net = require('net');
3 var text = "";
4
5 var server = net.createServer(
6     function(c) { // 'connection' listener
7         console.log('server connected');
8         c.on('end', function() {
9             console.log('server disconnected');
10         });
11         c.on('data', function(data) { // Read what the client sent.
12             console.log( data.toString() ); // Print data to stdout.
13             text += data;
14             console.log('text =\n'+text); // Print text to stdout.
15             c.write(text + " "); // Send the result to the client.
16             c.end(); // Close connection.
17         });
18     });
```

```
19
20 server.listen(9000,
21   function() { //listening' listener
22     console.log('server bound');
23   });
```

In this case, instead of exchanging the words “Hello” and “world”, the client sends a text string (received as an argument from the command line) to the server and the server appends the received text to a global variable and returns the resulting string. Therefore, each client “invocation” (since there are request and reply messages) made by each client appends a text line to a sequence of lines maintained by the server (and “shared” by all clients) and the server returns the current contents of that sequence.

Please complete the following tasks:

1. Once you have started a server process using “`node Act11server`”, in the same computer, start 4 client processes (using “`node Act11client '...' &`” for each one in any UNIX system using a text string instead of the ellipsis: ‘...’). Check whether there have been any “race conditions” in that execution (Which should be the final value for the “text” string? Have you obtained that value? Do we obtain the same value if we restart all these processes?). Explain your answers.

2. What happens if you replace the line `"counter += data;"` in the server code with this other: `"counter += parseInt(data) ;"`? Why?

(In this variant we should assume that the received argument in client processes is a valid representation of an integer number.)

3. The programs shown in this wording and used in its parts 1 and 2 allow us to verify whether "race conditions" are possible or not.

Modify both programs in order to achieve the following:

- The client should receive two arguments from the command line. The first one is an integer number (to be interpreted as the client identifier) and the second one is a text string (the information to be forwarded to the server).
- The client builds an object with the values of those two arguments and serialises it using the **JSON.stringify()** function.
- The client prints the serialised object to the screen and sends it to the server.
- The server uses an array (as a global variable) to store the data sent by clients.
- Each time the server receives a client message, it rebuilds that incoming object using the **JSON.parse()** function.
- The server stores the contents of such object in the array as follows: the first object property (i.e., the client identifier) is used as the array index and the second object property (the string) is stored in that array slot.
- Finally, the server sends the complete array to the client as its reply.
- The client, once the array is received, prints it to the screen.

Once you have applied this extension, start again four clients with different strings as their arguments and check whether there have been any race conditions. Justify the obtained results.

