NIST: Unit 2. Exercises

This bulletin contains several exercises. Those exercises may be classified, according to their relation with the goals of Unit 2, in these groups:

a) Mandatory (most important goals in Unit 2): 2, 6 and 10.
b) Recommended (other important goals in Unit 2): 1, 3, 4, 5, 11, 12 and 13.
c) Complementary (other aspects of Unit 2; unimportant for the assessments): 7, 8 and 9.

ACTIVITY 1

GOAL: To delve in the JavaScript argument passing mechanisms.

EXERCISE: Please consider the following program[1] and answer the questions being shown afterwards:

```
 1 function table(x) {  // Prints column x of a (1..10) multiplication table
 2         for (let 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 (let 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  table(table(2));
```

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  allTables(table(30),table(20),table(10));
```

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: Consider the following program:

```
 1 const ev = require('events')
 2 const emitter = new ev.EventEmitter
 3 const e1 = "print"
 4 const e2 = "read"
 5 const books = [ "Walk Me Home", "When I Found You", "Jane's Melody", "Pulse" ]
 6 
 7 // Function that creates the intended event listeners.
 8 function createListener( eventName ) {
 9         let num = 0
10         return function (arg) {
11                 let book = ""
12                 if (arg)
13                         book = ", now with book title '" + arg + "',"
14                 console.log("Event " + eventName + book + " has " +
15                         "happened " + ++num + " times.")
16         }
17 }
18 
19 // Listeners are registered in the event emitter.
20 emitter.on(e1, createListener(e1))
21 emitter.on(e2, createListener(e2))
22 // There might be more than one listener for the same event.
23 emitter.on(e1, () => console.log("Something has been printed!!"))
24 
25 function emitE2() {
26         let counter=0
27         return function () {
28                 // This second argument provides the argument for the "e2" listener.
29                 emitter.emit(e2,books[counter++ % books.length])
30         }
31 }
32 // Generate the events periodically...
33 // First event generated every 2 seconds.
34 setInterval( () => emitter.emit(e1), 2000 )
35 // Second event generated every 3 seconds.
36 setInterval( emitE2(), 3000 )
```

This program behaves as an event emitter and it adds some new aspects to what has been explained in Unit 2:

- The “read” event is generated with an additional argument (title to be read). The sentences needed to this end are shown in these lines:
  - 36: Frequency of this event (1 event per 3 seconds).
  - 25-31: A closure maintains the number of generated events. Thus, such counter determines which message is provided as an argument when the event is emitted. Note that the program **calls** this function in line 36.

- 12-14: The “listener” for this event needs a parameter.

Taking this program as a base, please develop a different program 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 function for each emitted event. When each event is thrown, they should be managed as follows:
  - “one”: Write the string “Event one.” 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 interval for event “two” will be tripled, until such length becomes 18 seconds. Once this happens, those events must occur every 18 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          let sum=0
 9          for (let i=0; i<v.length; i++)
10                  sum += v[i]
11          return sum
12  }
13
14  function iterate(num, f, vec) {
15          let amount = num
16          let result = 0
17          if (vec.length<amount)
18                  amount=vec.length
19          for (let i=0; i<amount; i++)
20                  result += f(vec[i])
21          return result
22  }
23
24  let 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.

c) line 27.

d) line 28.

e) line 29.

f) line 30.

ACTIVITY 4

GOAL: To adequately deal with the execution of a sequence of asynchronous operations.

EXERCISE: Implement a short version of the UNIX “**cat**” command. It should show in its standard output the contents of a sequence of files. Those file names should be received as command-line arguments. There is no limit in the amount of file names that can be processed in a single command execution.

Assume that all files are text files. The output should respect the order in which those file names have been specified in the command line. This also implies that the output from the i-th file should not be shown until all lines from file i-1 have been already shown.

Please develop two versions of this program:

1) The first should be based on the fs.readFileSync() operation.

2) The other must use fs.readFile() instead.

Compare the performance of both versions when they need to process a large amount of files.

ACTIVITY 5

OBJECTIVE: To understand that asynchronous callbacks are sometimes synchronous.

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:

```
const fs = require('fs')
const path = require('path')
const os = require('os')
var rolodex={}

function contentsToArray(contents) {
    return contents.split(os.EOL)
}
function parseArray(contents,pattern,cb) {
    for(let i in contents) {
        if (contents[i].search(pattern) > -1)
            cb(contents[i])
    }
}

function retrieve(pattern,cb) {
    fs.readFile("rolodex", "utf8", function(err,data){
        if (err) {
            console.log("Please use the name of an existant file!!")
        } else {
            parseArray(contentsToArray(data),pattern,cb)
        }
    })
}

function processEntry(name, cb) {
    if (rolodex[name]) {
        cb(rolodex[name])
    } else {
        retrieve( name, function (val) {
            rolodex[name] = val
            cb(val)
        })
    }
}

function test() {
    for (let 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  const testNames = ['a', 'b', 'c']
48  test()
```

When we run it[2], 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

GOAL: To adequately manage a sequence of asynchronous operations.

EXERCISE: Implement a short version of the **grep** command from UNIX systems. To this end, a NodeJS program that accepts at least two command-line arguments should be written. Its first argument is the word (let us call it “W1”, hereafter) to be found and all remaining arguments are the names of multiple text files where that word is looked for.
Each time “W1” is found, our program should write a line with the following information:

file-name: line-number: contents of that file line

The program must accept a list of file names of any arbitrary length, and it should use fs.readFile() in order to read each file.

Hints: Use operation “split()” in order to break the file contents into an array of text lines and “includes()” in order to find whether “W1” is contained in a file line. Note that both “split()” and “includes()” are standard methods available in all JavaScript String objects.

ACTIVITY 8

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:

```
/* repl_show.js */

const repl = require('repl')
const f = function(x) {console.log(x)}
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:

```
/* repl_client.js */

const net = require('net')
const sock = net.connect(8001)

process.stdin.pipe(sock)
sock.pipe(process.stdout)
```

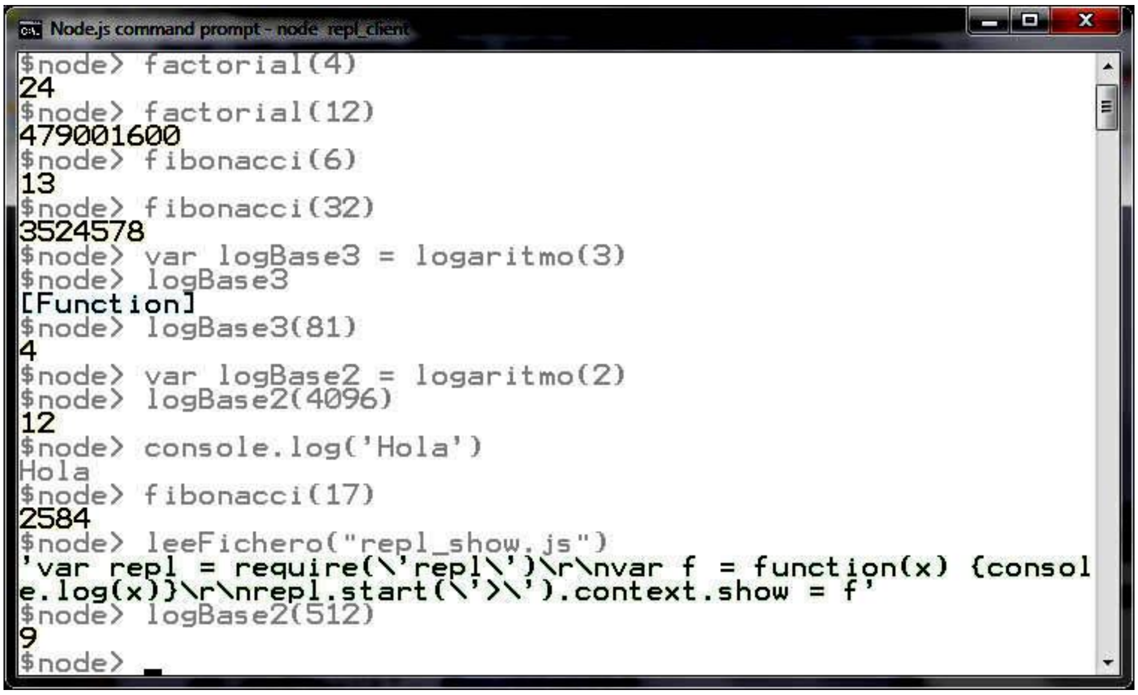
```
/* repl_server.js */

const net = require('net')
const repl = require('repl')

net.createServer(function(socket){
  repl
  .start({
    prompt: '>',
    input: socket,
    output: socket,
    terminal: true
  })
  .on('exit', function(){
    socket.end()
  })
}).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:



```
$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 section 2.4.3, Closures, from the Student Guide of this unit 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 9

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

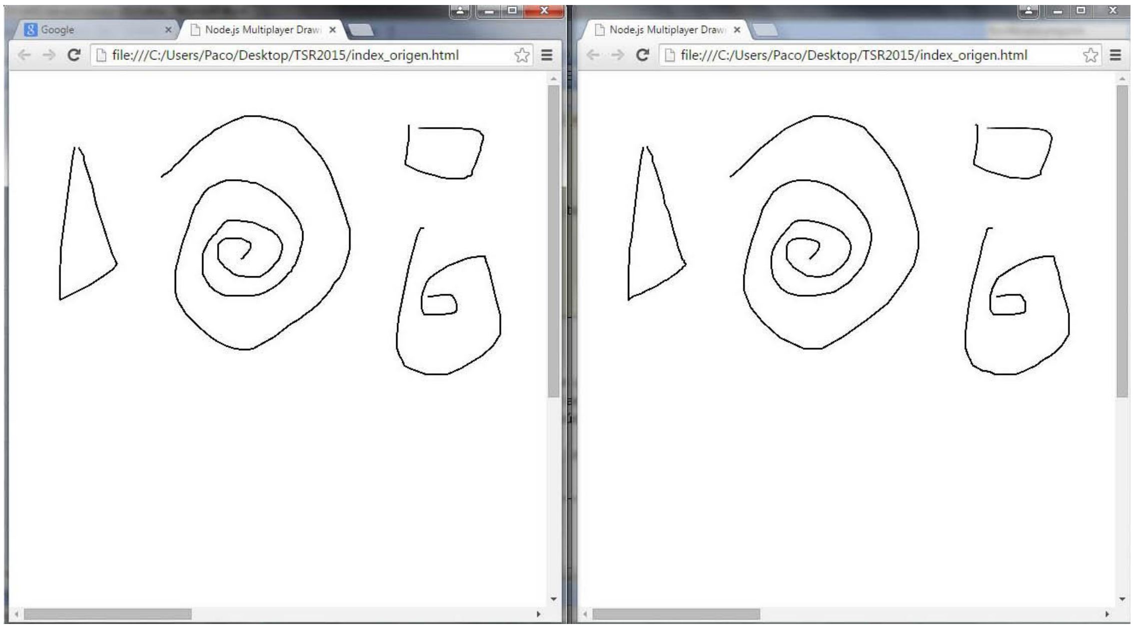
EXERCISE: Please consider the following drawing application that consists of an HTML file ("index.html") and a JavaScript file ("script.js"). These files are:

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

```javascript
$(function(){
  // This demo depends on the canvas element
  if(!('getContext' in document.createElement('canvas'))){
    alert('Sorry, it looks like your browser does not support canvas!');
    return false;
  }
  let doc = $(document),
      win = $(window),
      canvas = $('#paper'),
      ctx = canvas[0].getContext('2d'),
      instructions = $('#instructions'),
      id = Math.round($.now()*Math.random()), // Generate an unique ID
      drawing = false, // A flag for drawing activity
      clients = {},
      cursors = {},
      prev = {};
  canvas.on('mousedown',function(e){
    e.preventDefault();
    drawing = true;
```

```
        prev.x = e.pageX;
        prev.y = e.pageY;
    });
    doc.bind('mouseup mouseleave',function(){
        drawing = false;
    });
    doc.on('mousemove',function(e){
        // Draw a line for the current user's movement
        if(drawing){
            drawLine(prev.x, prev.y, e.pageX, e.pageY);
            prev.x = e.pageX;
            prev.y = e.pageY;
        }
    });
    function drawLine(fromx, fromy, tox, toy){
        ctx.moveTo(fromx, fromy);
        ctx.lineTo(tox, toy);
        ctx.stroke();
    }
});
```

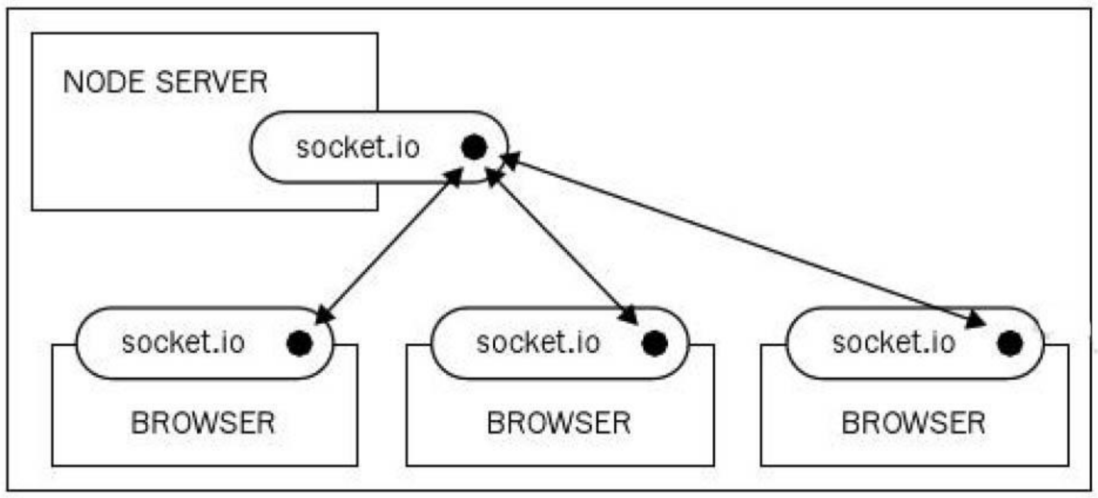
This application needs a canvas object and is driven by mouse-related events. It is a single-user drawing application. Our goal is to extend it, allowing that multiple users load it in their browsers, sharing the same drawing board (i.e., the lines drawn by a user will be seen by all users). The next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


Google
Node.js Multiplayer Draw
file:///C:/Users/Paco/Desktop/TSR2015/index_origen.html
Node.js Multiplayer Draw
file:///C:/Users/Paco/Desktop/TSR2015/index_origen.html

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:

```
const io = require('socket.io').listen(8080);

io.on('connection', function (socket) {
   socket.broadcast.emit('user connected');
});
```

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 10

GOAL: Introduce an asynchronous management in applications that initially had a synchronous behaviour. To this end, use callbacks (or, alternatively, promises) in an adequate way.

EXERCISE: This is an exercise to put into practice the concepts and constructs related to asynchrony in NodeJS. The program to be developed, in a first approximation, must read the content of a text file (using the fs module), order its lines alphabetically, and write the result to a new file. Features related to synchronism and the number of files to sort will be gradually incorporated. Complex aspects or combinations are resolved from other simpler solutions; Therefore, the exercise is organized in 5 stages:

1. **Synchronous** version: sR->S->sW (sR =Synchronous Read, S =Sort, sW =Synchronous Write). It is the starting point, for a single file.
2. **"Semisynchronous"** version: sR->S->W->E where W is asynchronous and a final E operation is required.
3. **Asynchronous** version: R->S->W->E where R is also asynchronous. An event-oriented alternative can be proposed.
4. **Asynchronous with two files**. There will be 2 branches Ri-> Si ->Wi->Ei (1<=i<=2) independent of each other. Add an additional **message** when **both finish**.
5. **Generalize** for an indeterminate number of files.

The full statement (in the next page) adds some details. In order not to deviate from the topic of interest, the possibility of errors is not considered. The ordering of the file is also not a concern (to this end, use the function sort_lines() whose code is in the solution example of the first section)

Stages

1. (already resolved) **Synchronous version**: sR ->S->sW (sR =Synchronous Read, S =Sort, sW =Synchronous Write). Synchronous variants of reading and writing to files are taken. W doesn't start until S ends, S doesn't start until R ends.
   Solution:

```
// sort_lines function
var fs = require( 'fs' )
file = process.argv[ 2 ]

function sort_lines(mystring)
{ return mystring.toString().split("\n").sort().join("\n")}

var r=fs.readFileSync(file, 'utf-8')
fs.writeFileSync( file + "2" , sort_lines ( r ), 'utf-8' )
```

2. **"Semisynchronous"** version: The writing will be asynchronous. Therefore, in order to know when it has finished, we will add an E operation to the end: console.log("End"), after the writing. We don't use prefixes for async operations in our execution schemes. The ordering will be sR ->S->W->E.

3. **Asynchronous** version: R->S->W->E The reading will also be asynchronous, so that the precedence relationship forces (S+W) to be a callback of R, and that E remains a callback of W. This transformation of sequentiality in nesting can be a problem.
   - Alternatively, it can be **event** oriented, with the completion of R being represented as an event triggering (S+W), the completion of W being an event triggering E.

4. **Two files**. Now, there will be 2 branches Ri-> Si ->Wi->Ei (1<=i<=2). The approach assumes that one branch has NO dependency on the other, and that independence must be used in the solution. This assumes that e.g. an ordering R1-R2-S2-W2-S1-E2-W1-E1 is just as valid as R2-R1-S2-S1-W2-W1-E2-E1 or any other interleaving that respects the relationships expressed for each branch. The solution cannot limit this because it would lose performance; Can you argue it?, and give counterexamples?
   - Add an **additional completion message** when both branches finish (there will be an "End 1", "End 2" and "End All").

5. **Generalize** for an indeterminate number of files. If in the previous section you have created a solution for exactly two files, you may have opted for an "artisan" version. Switching to 3, 4, or 5 files may test that craft, but generalizing in the way indicated means changing your approach. A vector of functions...? You have to imagine it before you write it. Remember that you still need the completion message. Alternatively it can be resolved with promises.

ACTIVITY 11

GOAL: To delve in the knowledge of the ‘net’ module.

EXERCISE: Consider these three files:

```
// file: proxy.js
const net = require('net')

const LOCAL_PORT  = 8000
let remotePort = process.argv[3] || 8001
let remoteIP = process.argv[2] || '127.0.0.1'

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

```
// File: worker.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(parseInt(data+'')*3+'')
            })
    })

server.listen(parseInt(process.argv[2]) || 8001,
    function() { //listening listener
        console.log('server bound')
    })
```

```
// File: client.js
const net = require('net')

const client = net.connect(parseInt(process.argv[2]) || 8000,
    function() { //connect listener
        console.log('client connected')
        client.write(process.pid+'')
    })

client.on('data',
    function(data) {
        console.log(data.toString())
    })

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

The first file, proxy.js, behaves as an intermediary between the other two. Therefore, communication is initiated by a client process, who sends a message to the proxy. That message is forwarded by the proxy to the worker. The worker processes that request and returns a reply to the proxy. Finally, the proxy returns that reply to the client.
In the current version, once those interactions are completed, the client remains alive but it is unable to do anything else.
Please, answer the following parts:

1. Using the original code shown in the figures, start one process of each kind, in this order: worker, proxy and client. Once the client has shown its received reply, kill the proxy. Explain what happens in the other two agents.
2. In order to achieve client termination, that agent should close its connection once it has received its reply. Extend "client.js" in order to implement that behaviour.
3. Once you have extended client.js, describe what happens (i.e. whether the execution completes successfully or there is any error; in case of error, explain which error occurs and how could it be avoided) when those three agents are started in the following orders:
    a. Worker, proxy, client.
    b. Proxy, worker, client.
    c. Proxy, client, worker.
    d. Client, proxy, worker.
4. Extend all three programs in order to adequately manage the 'error' event in their connections. Repeat part 3 with those new programs and describe whether those execution orders cause the same process abortions or not. Explain the new scenarios if any of them has changed. After completing each start sequence, try to start new clients and check whether they behave as intended or not.
   Hint: For "serviceSocket", set its 'error' event listener before it connects to the worker.
5. The connections managed by the "net" module are "transient" since they rely on TCP sockets. Unit 3 will describe the ZeroMQ library. ZeroMQ is (weakly) "persistent". A persistent communication channel allows message sending even when the other communication side is not connected yet. Describe what is needed in the "net" module in order to implement "persistent" channels on top of TCP connections.

ACTIVITY 12

OBJECTIVE: Exercise on NET server, objects and arrays, JSON functions, FS module, and periodic events (setInterval).

EXERCISE: We want to implement a **net** server that receives electoral results, which will be transmitted to it through an indeterminate number of **net** clients. The server must account for, adequately accumulate, and keep in files all the information that is transmitted to it.

Any **net client** will be able to send to the server objects that contain the votes obtained by each political party in a certain electoral college. A couple of examples of this class of objects:

{place:madrid, pp:3532, psoe:2056, up:3077, cs:1540}

{place:barcelona, pp:1056, psoe:1403, up:2056, cs:1986, erc:2389}

The objects will always have a **place property** that identifies the electoral college, and a variable number of properties such that its identifier is that of a political party and its value is the votes obtained by that party.

These objects, serialized with **JSON.stringify, are the data that the net** server will receive. The server has to process the data received, saving them properly in memory and on disk:

- In memory it must keep a variable of type array, let's call it **votes** , which uses the **place property** (of the objects received) as the index of the array, and stores in the position of the array thus indexed an object with all the votes received by each party in that district.
- On disk, periodically (every 20 seconds), you must save a text file (extension txt) for each entry in the **votes array**. The name of the file will be that of the index of the array (the **place property**) and the content of the file will be the value stored in that position of the array, serialized with JSON.

As an example, consider that during the first 20 seconds of server execution, the following data has been collected from several **net clients** :

{place:madrid, pp:3500, psoe:2000, up:3000, cs:1500}

{place:barcelona, pp:1000, psoe:1500, up:2000, cs:2000, erc:3000}

{place:madrid, pp:2000, psoe:3000, up:1000, cs:500}

{place:valencia, pp:2500, psoe:1500, up:2000, cs:2500}

{place:madrid, pp:4000, psoe:3000, up:2000, cs:2000}

{place:barcelona, psoe:500, up:400, cs:200, erc:300}

The server's **votes** variable must hold the following information:

votes['madrid'] = {pp:9500, psoe:8000, up:6000, cs:4000}

votes['barcelona'] = {pp:1000, psoe:2000, up:2400, cs:2200, erc:3300} votes['valencia'] = {pp:2500, psoe:1500, up:2000, cs:2500}

And the following files will have been written on disk:

file name	***File content***
madrid.txt	{“pp”:9500, “psoe”:8000, “up”:6000,” cs”:4000}
barcelona.txt	{“pp”:1000, “psoe”:2000, “up”:2400, “cs”:2200, “erc”:3300}
valencia.txt	{ “pp”:2500, “psoe”:1500, “up”:2000, “cs”:2500}

You must implement that **net** server, based on, and keeping in the solution, the following code:

```
const net = require('net')
const fs = require('fs')
var votes = {}

var server = net.createServer(function(c) {
        c.on('data', function(data){
                // TO COMPLETE
        })
})

server.listen(9000,
  function() { console.log('server bound')
})

function save() {
        // TO COMPLETE
        console.log('data saved to disk')
}

// TO COMPLETE
```

ACTIVITY 13

OBJECTIVE: Exercise on callbacks, closures, objects and arrays, JSON functions, FS module, and interaction with process.stdin.

EXERCISE: We want to implement a server that allows obtaining electoral results from the reading of a set of text files (where those data are stored) and to develop an interactive session, with the user of the application, to consult the results of each electoral college.

It is considered that the server runs in a directory where the text files with the electoral results are located. The names and contents of the files follow the format described in the previous exercise, like this, for example:

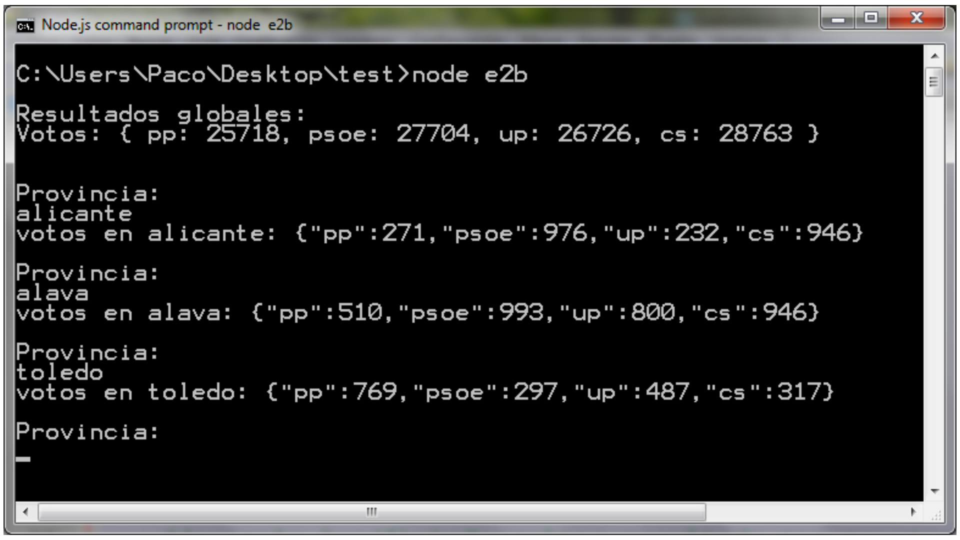
file name	***File content***
madrid.txt	{“pp”:9500, “psoe”:8000, “up”:6000,” cs”:4000}
barcelona.txt	{“pp”:1000, “psoe”:2000, “up”:2400, “cs”:2200, “erc”:3300}
valencia.txt	{ “pp”:2500, “psoe”:1500, “up”:2000, “cs”:2500}

The server, first of all, must read all the files, saving the electoral results in an array, let 's call it **votes**, with the same criteria as in the previous exercise.

The server, secondly (that is, once all the files have been read), must:

- Show the global results (the total votes obtained by each party in all provinces), stored in another array called **total_votes** .
- Start an interactive session, using **process.stdin**, to view the results in each province.

The following screenshot serves as a reference of the functionality of the application:



```
C:\Users\Paco\Desktop\test>node e2b

Resultados globales:
Votos: { pp: 25718, psoe: 27704, up: 26726, cs: 28763 }

Provincia:
alicante
votos en alicante: {"pp":271,"psoe":976,"up":232,"cs":946}

Provincia:
alava
votos en alava: {"pp":510,"psoe":993,"up":800,"cs":946}

Provincia:
toledo
votos en toledo: {"pp":769,"psoe":297,"up":487,"cs":317}

Provincia:
_
```

In this example, the user has written the names of 3 provinces (*alicante* , *alava* , *toledo*), and the rest of the output has been generated by the server to be implemented.

It is requested to implement the server, based on, and keeping in the solution, the following code:

```
const fs = require('fs')
var total_votes = {}

fs.readdir('.', function (err, files) {
var count = files.length
var votes = {}
    for (var i=0; i < files.length; i++) {
        // TO COMPLETE
    )}
})
```