

TSR: Actividades del Tema 2

ACTIVIDAD 1

OBJETIVO: Profundizar en la gestión del paso de argumentos en la invocación de funciones en JavaScript.

ENUNCIADO: Tomando como base el siguiente código¹:

```
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) Describa cuál es la salida proporcionada por el programa anterior. Justifique si tiene o no algún efecto el pasar más de un argumento en la línea 12.

¹ Todos los ficheros fuente listados o mencionados en este documento se encuentran en un archivo "acts.zip" disponible en PoliformaT.

- b) Suponga que ahora reemplazamos la línea 12 original del programa anterior por la siguiente, ¿qué salida proporciona el programa en este caso? ¿por qué?

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

- c) Suponga que ahora reemplazamos la línea 12 original del programa por la siguiente, ¿qué salida proporciona el programa ahora? ¿por qué?

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

- d) A partir de los resultados obtenidos en los apartados anteriores, justifique si JavaScript acepta y puede tener algún efecto que se pasen más argumentos de los que espera una función determinada.

ACTIVIDAD 2

OBJETIVO: Adquirir soltura en la programación de eventos de JavaScript.

ENUNCIADO: Sea el programa siguiente:

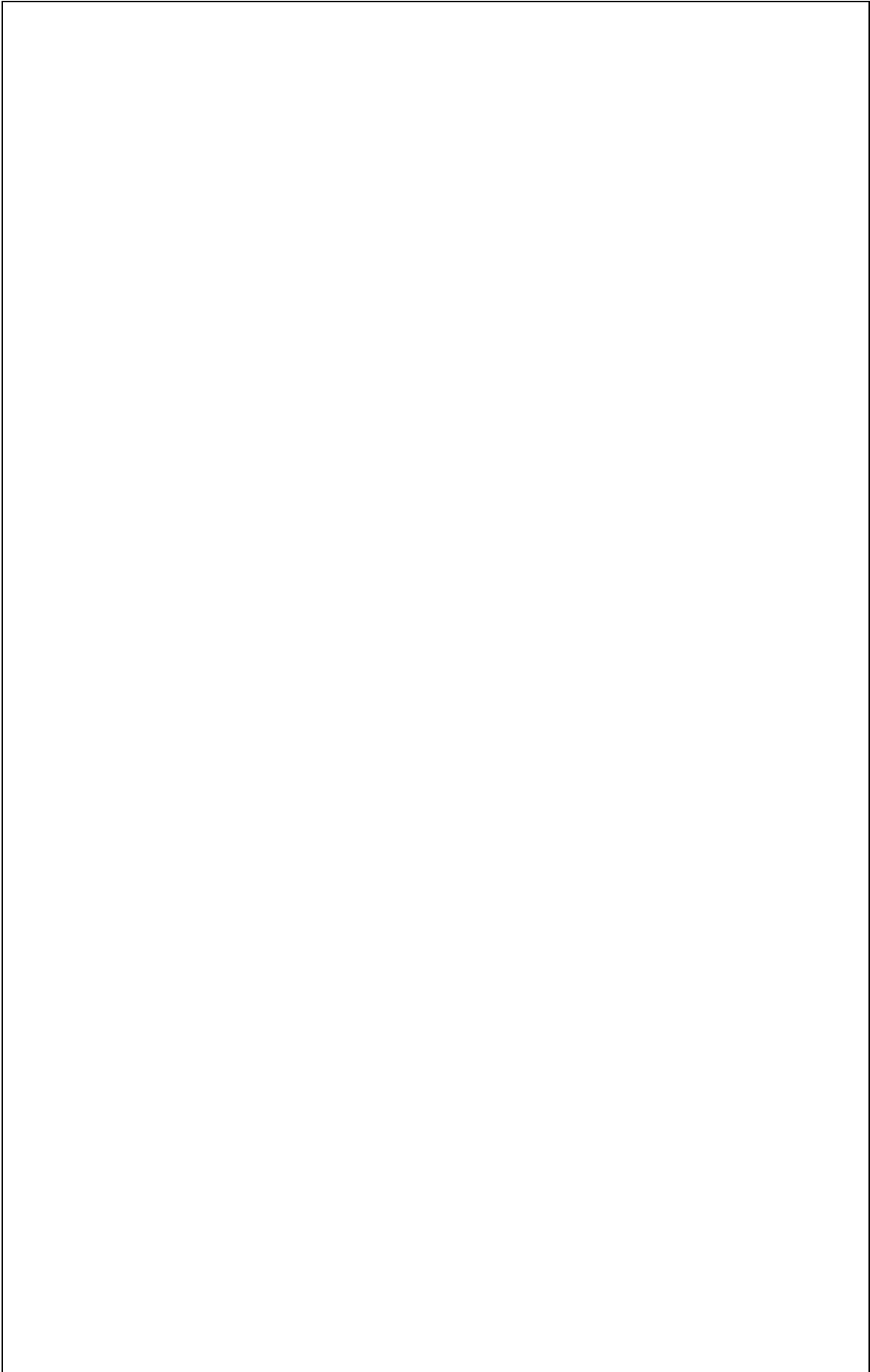
```
1  const ev = require('events')
2  const emitter = new ev.EventEmitter
3  const e1 = "print"
4  const e2 = "read"
5  const books = [ "Walk Me Home", "When I Found You", "Jane's Melody", "Pulse" ]
6
7  // Constructor for class Listener.
8  function Listener(n1,n2) {
9      this.num1 = 0
10     this.name1 = 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 const 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 let counter=0;
36 function generateEvent2() {
37     // This second argument provides the argument for the "e2" listener.
38     emitter.emit(e2,books[counter++ % books.length])
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 )
```

Además del generador de eventos, en este programa se ha creado un objeto “listener” que ofrece un método para cada evento que puede disparar el generador. Este programa también ilustra algunos aspectos más:

- El evento “read” en este caso se genera con un argumento adicional (el título de la novela a leer). El código necesario para ello se muestra en las líneas:
 - 46: Para determinar cada cuánto se genera ese evento (3 segundos en este ejemplo).
 - 35-39: Declaración de la variable “counter” necesaria para mantener el número de eventos generados y, en base a su valor, determinar qué mensaje acompaña al evento como argumento.
 - 30: Ahora el “listener” para ese evento debe tener un parámetro.
 - 20-23: Y la función utilizada por ese “listener” también.
- El objeto Listener, cuyo constructor se muestra en las líneas 8 a 13, mantiene ahora el número de veces que ha llegado a procesar cada uno de los eventos. El incremento de tales contadores se realiza en cada una de las funciones instaladas como “listener” de un evento (en las líneas 16 y 22, respectivamente).

Tomando ese programa como base, se pide que el alumno desarrolle otro programa en el que:

- Se generen los eventos siguientes:
 - “uno”: Cada tres segundos. Sin argumentos.
 - “dos”: Inicialmente cada dos segundos. Sin argumentos.
 - “tres”: Cada diez segundos. Sin argumentos.
 - Haya un objeto Listener con un método para cada evento generado. Al recibir cada evento, el Listener tendrá que hacer lo siguiente:
 - “uno”: Escribir la cadena “Listener activo: X eventos de tipo uno.” en su salida estándar. En ese mensaje, en lugar de una ‘X’ deberá mostrarse el número de eventos de tipo uno recibidos hasta el momento.
 - “dos”: Escribir la cadena “Evento dos.” en su salida estándar si el número de eventos “dos” recibidos hasta ahora es superior al número de eventos “uno”. Cuando eso ya no suceda, escribirá “Hay más eventos de tipo uno”.
 - “tres”: Escribir un mensaje “Evento tres.” por salida estándar. Además, con cada ejecución de este manejador, se triplicará la duración del intervalo entre dos eventos consecutivos de tipo “dos”, hasta que dicho valor sea 18 segundos. A partir de ese momento, los eventos de tipo “dos” se programarán cada 18 segundos.
- La operación “setInterval()” retorna un objeto que debe ser empleado como único argumento de “clearInterval()”. Para modificar la frecuencia de un evento, conviene utilizar “clearInterval()” antes de establecer la nueva frecuencia.



ACTIVIDAD 3

OBJETIVO: Entender las clausuras y el paso de funciones como argumento en JavaScript.

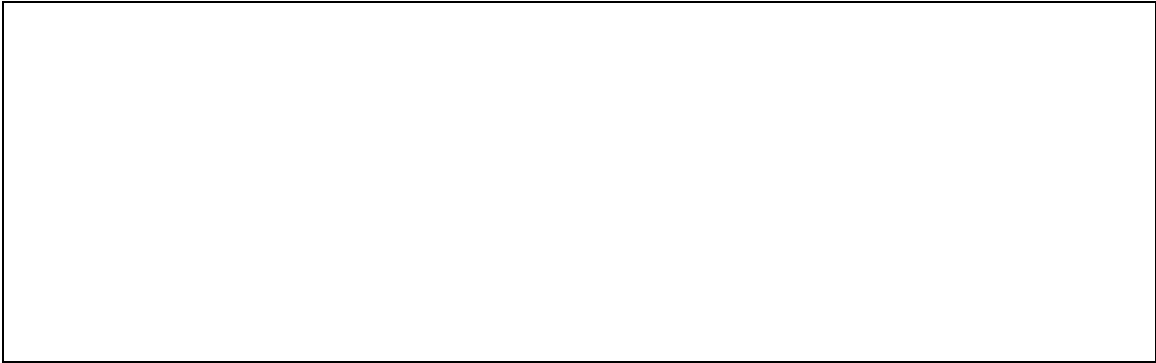
ENUNCIADO: Sea el programa siguiente:

```
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))
```

Ejecute ese programa y diga cuál es el resultado de la ejecución de cada una de las líneas siguientes, justificando por qué se da en cada caso:

a) línea 25.

b) línea 26.

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

c) línea 27.

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

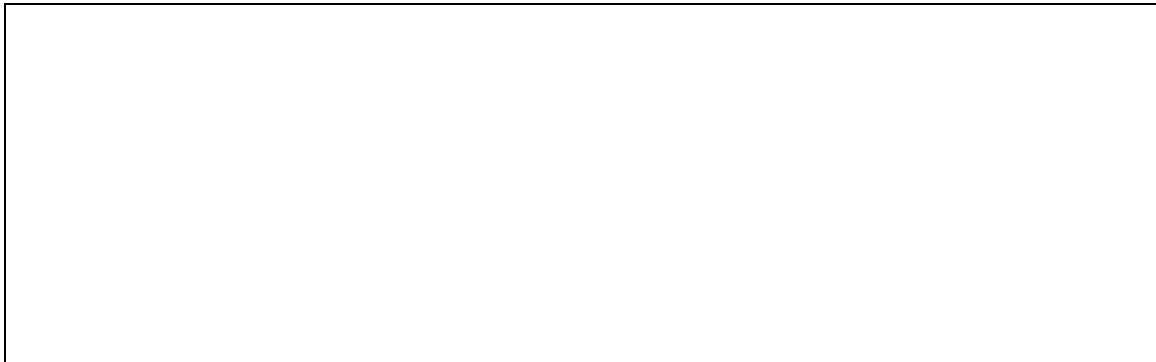
d) línea 28.

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

e) línea 29.

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

f) línea 30.



ACTIVIDAD 4

OBJETIVO: Extender el servidor web mínimo que se ha utilizado para ilustrar la funcionalidad del módulo HTTP.

ENUNCIADO: En una de las secciones de la presentación se mostró el siguiente programa:

```
1  const 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/')
```

Este servidor retorna la cadena “Hello World” a los navegadores que accedan a la URL <http://127.0.0.1:1337/> del ordenador en que se ejecute. No es capaz de devolver nada más.

Un servidor web debería ser capaz de devolver el contenido de los ficheros HTML especificados en la URL utilizada, para que el navegador solicitante pueda mostrarlos al usuario.

Para lograr esa funcionalidad se deben conocer algunos detalles adicionales que se describen seguidamente:

- El primer argumento (“request” en este ejemplo, que es un objeto de la clase ClientRequest) del “callback” utilizado en `http.createServer()` mantiene una propiedad “url” con la parte de la URL que sigue al nombre de máquina y puerto. Así, por ejemplo, si en el navegador se escribió lo siguiente:

<http://127.0.0.1:1337/dir1/pagina.html>

entonces la propiedad **request.url** contendría la cadena “dir1/pagina.html”.

- Para acceder a los ficheros conviene utilizar el módulo “fs”. En el ejemplo citado en el punto anterior, el servidor debería leer el contenido del fichero “pagina.html” ubicado en el directorio “dir1”. Eso se puede realizar utilizando un código similar al siguiente (asumiendo que se hizo un “const fs=require(‘fs’)” al empezar el programa):

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

Pero esto también conlleva problemas, pues el nombre de fichero utilizado como primer argumento debería ser un nombre de ruta absoluto. La función empleada como segundo argumento es un “callback” cuyo primer parámetro será un indicativo del error que haya podido darse y el segundo parámetro mantendría el contenido completo del fichero.

En nuestro caso, si hubiese un error habría que rellenar el objeto “**response**” con:

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

Ya que el identificador de error a devolver en la cabecera de la respuesta HTTP (ServerResponse) en este caso es el 404. La segunda sentencia proporciona el texto del mensaje de error. Por el contrario, cuando no haya ningún error, convendría rellenar el objeto “response” con:

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

En esa situación, el 200 en la cabecera indica que la petición ha podido resolverse sin problemas. La segunda sentencia vuelca el contenido del fichero (“**content**” era el segundo parámetro del “callback” y recogía el contenido del fichero en el método `readFile()` en la ServerResponse).

- Para formar correctamente el nombre de ruta absoluto del fichero HTML a devolver al navegador, se podrá utilizar el método `join()` del módulo PATH. En NodeJS existe un atributo “global” llamado “__dirname” que contiene el nombre de ruta absoluto del directorio actual. Así, se añadirá una sentencia “const path=import(‘path’)” en la parte inicial del programa y se utilizará una invocación similar a la siguiente:

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

para obtener el nombre de ruta absoluto.

- En la mayoría de los servidores web, cuando únicamente se utiliza la dirección del ordenador en la URL (sin especificar ningún nombre de fichero), se suele acceder a un fichero “index.html”. Esto tendrá que programarse explícitamente en caso de que se quiera obtener esa funcionalidad.
- Por otra parte, una URL puede especificar únicamente la dirección de un sitio web (<https://intranet.upv.es>) o la dirección completa de una página dentro del sitio (<http://www.upv.es/organizacion/escuelas-facultades/index-es.html>), pero también puede contener más información como, por ejemplo, parámetros de consulta

(considérese: https://intranet.upv.es/pls/soalu/sic_menu.Personal?P_IDIOMA=c, en esta página la información de consulta es “?P_IDIOMA=c”. El método `parse()` del módulo “url” permite obtener el objeto codificado en la cadena de una URL. La propiedad `query` almacena la información correspondiente a los parámetros de consulta.

Por ejemplo (asumiendo un “`const url=require('url')`” previo):

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

proporciona “P_IDIOMA=c”.

- La cadena correspondiente a los parámetros de consulta puede ser mucho más compleja (por ejemplo, considérese la siguiente URL incompleta:

[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&...)).

Conviene utilizar algún método que permita extraer parámetros concretos, y esto lo permite el método `parse()` del módulo “`querystring`”.

Por ejemplo (con un “`require`” previo del módulo “`querystring`”):

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

proporciona “Valencia”.

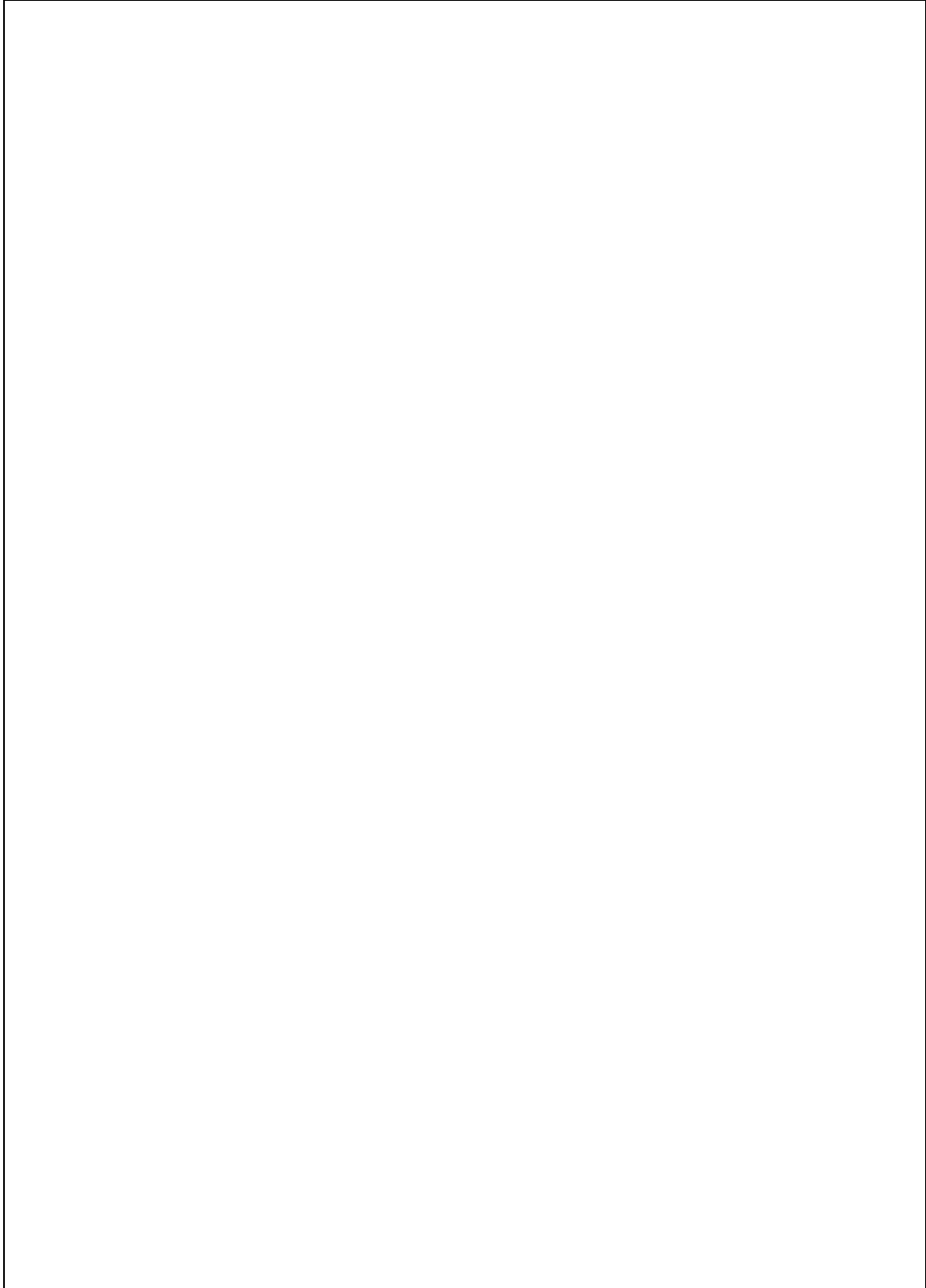
- Para ilustrar un uso básico de los parámetros de consulta, considérese el siguiente servidor:

```
1 const http = require('http')
2 const url = require('url')
3 const qs = require('querystring')
4 http.createServer( function(request,response) {
5   let query = url.parse(request.url).query
6   let info = qs.parse(query).info
7   let x = 'equis'
8   let y = 'y griega'
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')
```

Se pide extender el programa mostrado al empezar este enunciado para que sea capaz de gestionar un parámetro “consulta” en la URL utilizada y, en función de su valor, mostrar diferentes informaciones:

- Si el valor del parámetro es “time”, el servidor proporcionará la fecha y hora actual.
- Si el valor del parámetro es “dir”, el servidor proporcionará el nombre de su directorio y un listado con los nombres de todos los ficheros contenidos en el mismo.

- Cualquier otro valor de parámetro será interpretado como un nombre de fichero. Si existe un fichero con ese nombre en el directorio actual, el servidor lo leerá y proporcionará su contenido. En caso contrario, proporcionará un mensaje de error o aviso, indicando que no puede facilitar la información solicitada.



ACTIVIDAD 5

OBJETIVO: Entender que los “callbacks” no siempre son asincrónicos.

ENUNCIADO: Sólo hay un hilo de ejecución en Node.js. Esto implica que no tendremos por qué preocuparnos sobre la protección de variables compartidas con locks o cualquier otro mecanismo de control de concurrencia.

Sin embargo, hay algunos casos en los que necesitaremos ser cuidadosos.

Un buen principio para razonar sobre la lógica de un programa asincrónico es considerar que TODOS sus callbacks se ejecutarán en un turno posterior al que ahora ejecuta el código que los pasa como argumentos.

Considere el código que se muestra a continuación:

```
1  const fs = require('fs')
2  const path = require('path')
3  const 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(let 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 (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()

```

Cuando sea ejecutado², esperaremos esta salida:

```

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

```

TODOS los mensajes “processed” aparecen tras TODOS los mensajes “processing”, tal como se esperaba (los “callbacks” parecen ser llamados en turno futuro).

Sin embargo, considere esta variación:

Sustituya la línea 4 del código anterior (`var rolodex={};`) por la siguiente:

```

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

```

La salida que obtendríamos sería:

```

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

```

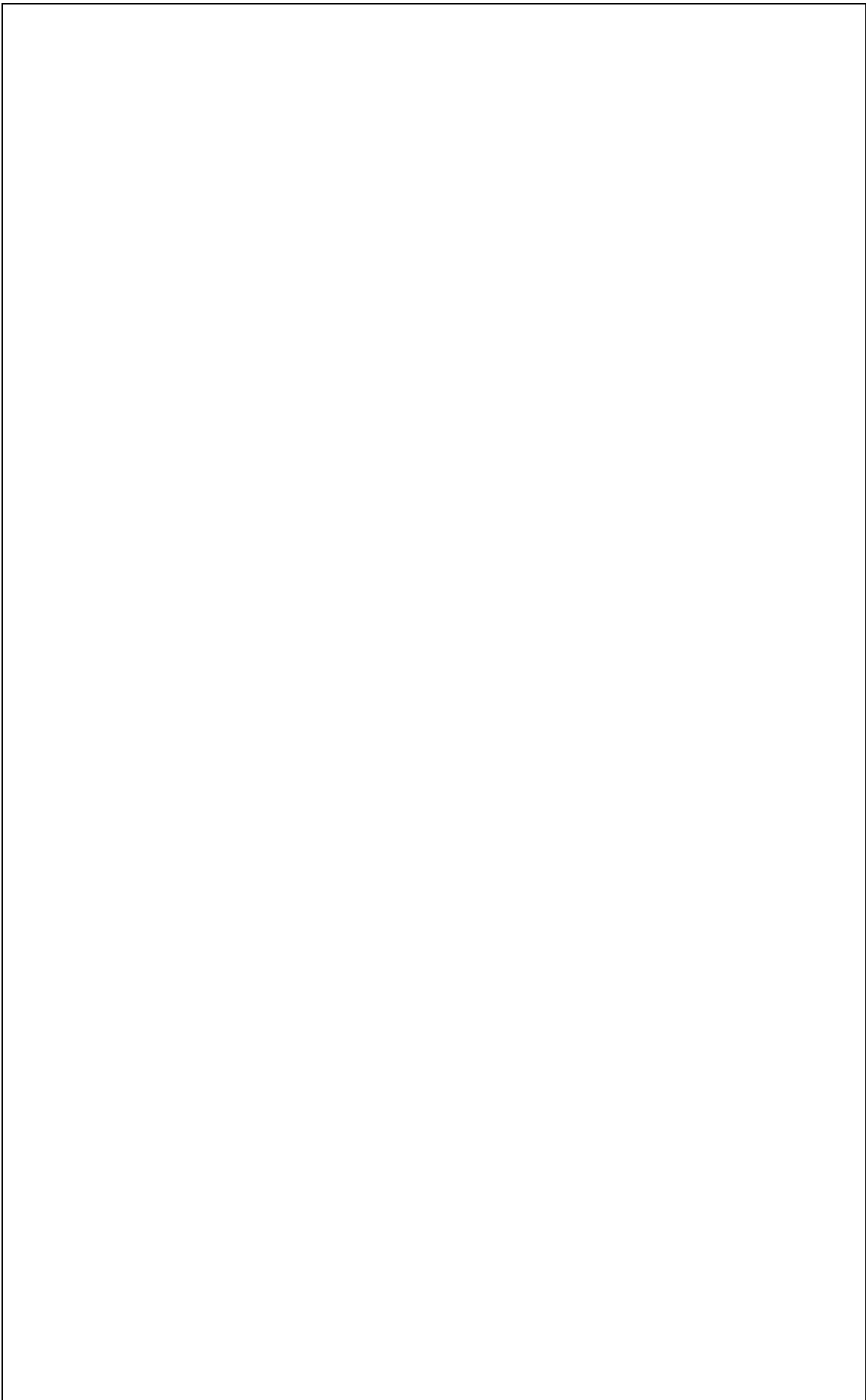
² Para ejecutar el programa correctamente deberá existir un fichero llamado “rolodex” en el mismo directorio. Ese fichero debe contener algunas líneas de texto. En ellas se buscarán las cadenas contenidas en el vector “testNames”.

Observe que ahora NO TODOS los mensajes “processed” se muestran tras TODOS los mensajes “processing”. La razón es que uno de los callbacks ha sido ejecutado EN EL MISMO turno que la función que lo pasó.

Dependiendo de la situación, esto podría introducir problemas difíciles de percibir, en caso de que el código que establece los callbacks y uno o más de los callbacks interfirieran en su acceso a una misma parte del estado del proceso, dejándolo inconsistente.

Esta situación (inconveniente) siempre aparecerá si los callbacks confían en que su invocador preparará sus contextos antes de que ellos inicien su ejecución.

Modifique el programa anterior, utilizando promesas, para garantizar que se respete el orden de ejecución esperado.



ACTIVIDAD 6

OBJETIVO: Utilizar adecuadamente “callbacks” y clausuras.

ENUNCIADO: Para acceder a ficheros, Node.js proporciona el módulo “fs”. Escriba un programa que reciba un número variable de nombres de ficheros desde la línea de órdenes y que escriba en pantalla el nombre del fichero más grande de todos ellos, así como su longitud en bytes. Para ello, utilice la función `fs.readFile()` del módulo “fs”, cuya documentación está disponible en https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback. No utilice las variantes sincrónicas de esa función u otras funciones del módulo “fs”.

Para gestionar los argumentos recibidos desde la línea de órdenes tendrá que utilizar el vector `process.argv` (https://nodejs.org/api/process.html#process_process_argv).

Si se reciben varios argumentos, necesitará utilizar clausuras para que el “callback” acceda correctamente a la posición adecuada del vector `process.argv`].

ACTIVIDAD 7

OBJETIVO: Profundizar en la conversión de “callbacks” en promesas.

ENUNCIADO: El programa que se muestra seguidamente implanta un servidor de ficheros (para ficheros de texto únicamente) que debería proporcionar tres operaciones a sus clientes: UPLOAD, DOWNLOAD y REMOVE. Los programas necesarios para solicitar esas operaciones (Act7upload.js, Act7download.js y Act7remove.js) se encuentran en PoliformaT donde también se incluye una versión de este programa servidor que utiliza promesas en lugar de “callbacks” (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 const net = require('net')
10 const fs = require('fs')
11 // Default port number.
12 let 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 const 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         const 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                         let 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                         const m2 = JSON.stringify(msg2)
51                         c.write(m2)
```

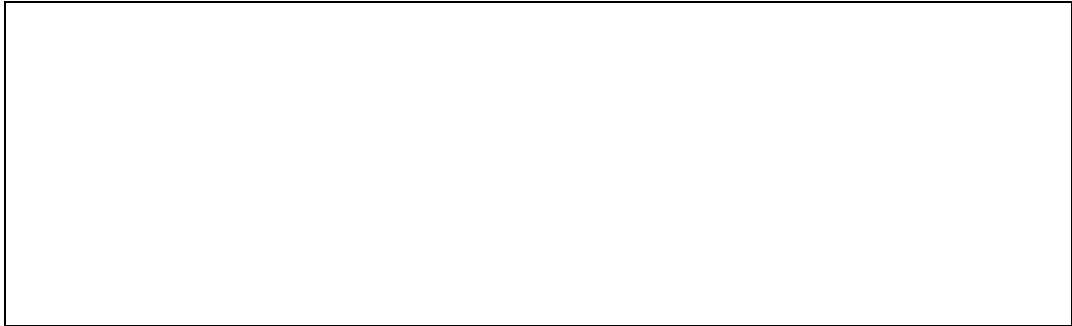
```

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                 let 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                 const 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 })

```

Se pide lo siguiente:

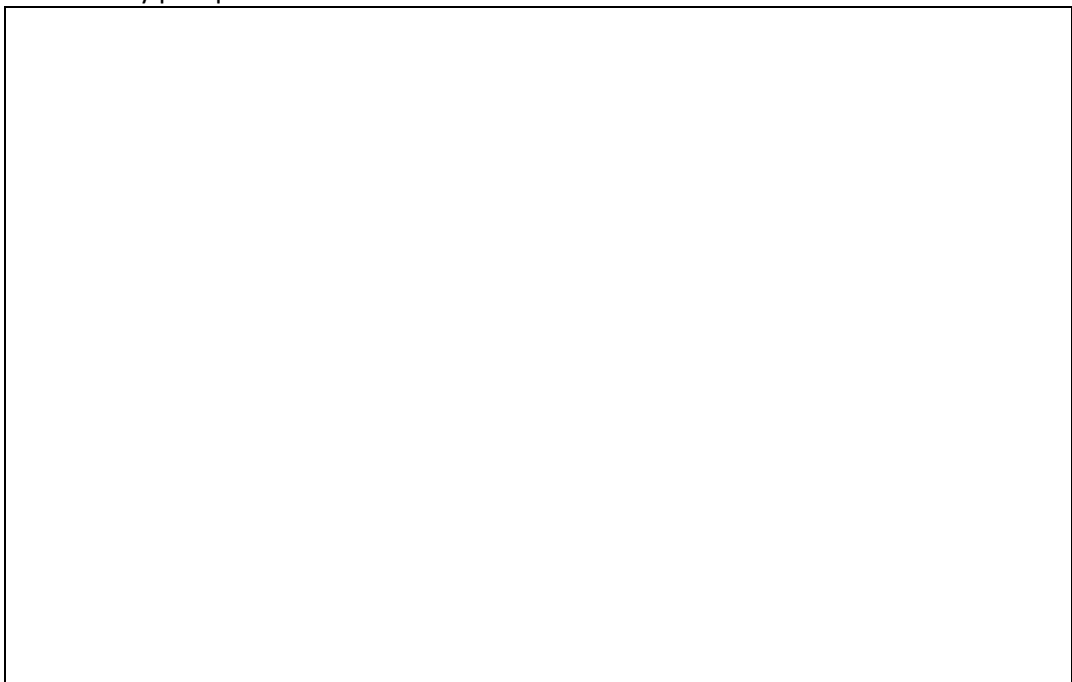
- Implante el código necesario para que la operación REMOVE realice su trabajo correctamente. Indique en su respuesta entre qué líneas del programa original deben añadirse las líneas que compongan su solución.



- b) Realice una ampliación equivalente en el programa basado en promesas (Act7serverPromise.js).



- c) Explique en qué programa ha tenido que realizar más trabajo para elaborar la solución solicitada y por qué motivo.



ACTIVIDAD 8

OBJETIVO: Ejecutar código de manera interactiva en un servidor remoto usando los módulos “net” y “repl”.

ENUNCIADO: EL módulo REPL (Read-Eval-Print Loop) representa el shell de Node. El shell se puede activar directamente en la línea de comandos de una terminal escribiendo:

```
> node
```

Además, el módulo “repl”, si se usa en el código de un programa Node, permite invocar el shell y ejecutar sentencias interactivamente. Considere el siguiente programa:

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

Un ejemplo de ejecución de este programa:

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

Como se aprecia, en el contexto del shell invocado con “repl” existe una función referenciada como “show” que es equivalente a “console.log”. En esta ejecución también se advierte la aparición del valor “undefined”, esto se puede evitar estableciendo una propiedad del módulo (consultar <https://nodejs.org/api/repl.html>).

La shell del módulo “repl” puede ser invocada remotamente. Considere el código de los programas, repl_client.js y repl_server.js, que se muestran a continuación:

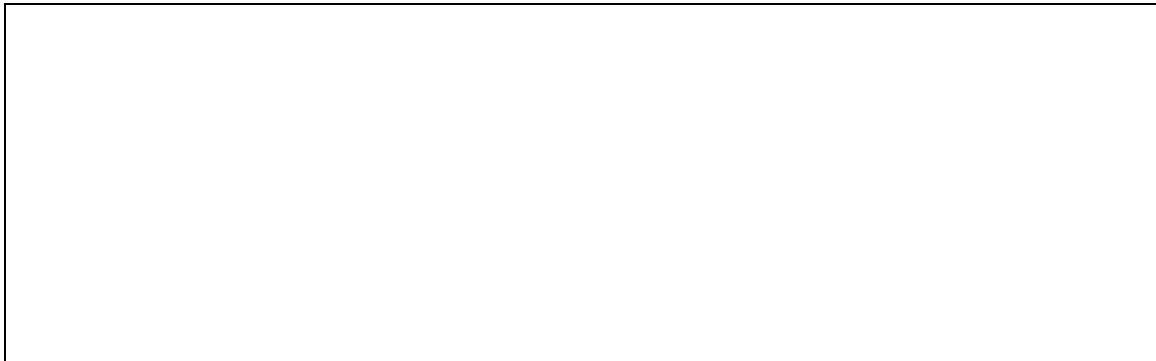
```
1 /* repl_client.js */
2
3 const net = require('net')
4 const sock = net.connect(8001)
5
6 process.stdin.pipe(sock)
7 sock.pipe(process.stdout)
```

```
1  /* repl_server.js */
2
3  const net = require('net')
4  const 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)
```

Se pide estudiar el funcionamiento de estos programas, consultando, si es preciso, la API de los módulos utilizados, y ejecutando los programas, interactuando con el shell remoto accesible desde el programa cliente.

- a) Explique el funcionamiento de cliente y servidor, precisando cómo fluye la información y dónde se realiza el procesamiento.

- b) Si en la terminal donde se ejecuta el cliente se escribe “`console.log(process.argv)`”, ¿qué se obtiene?, ¿por qué?



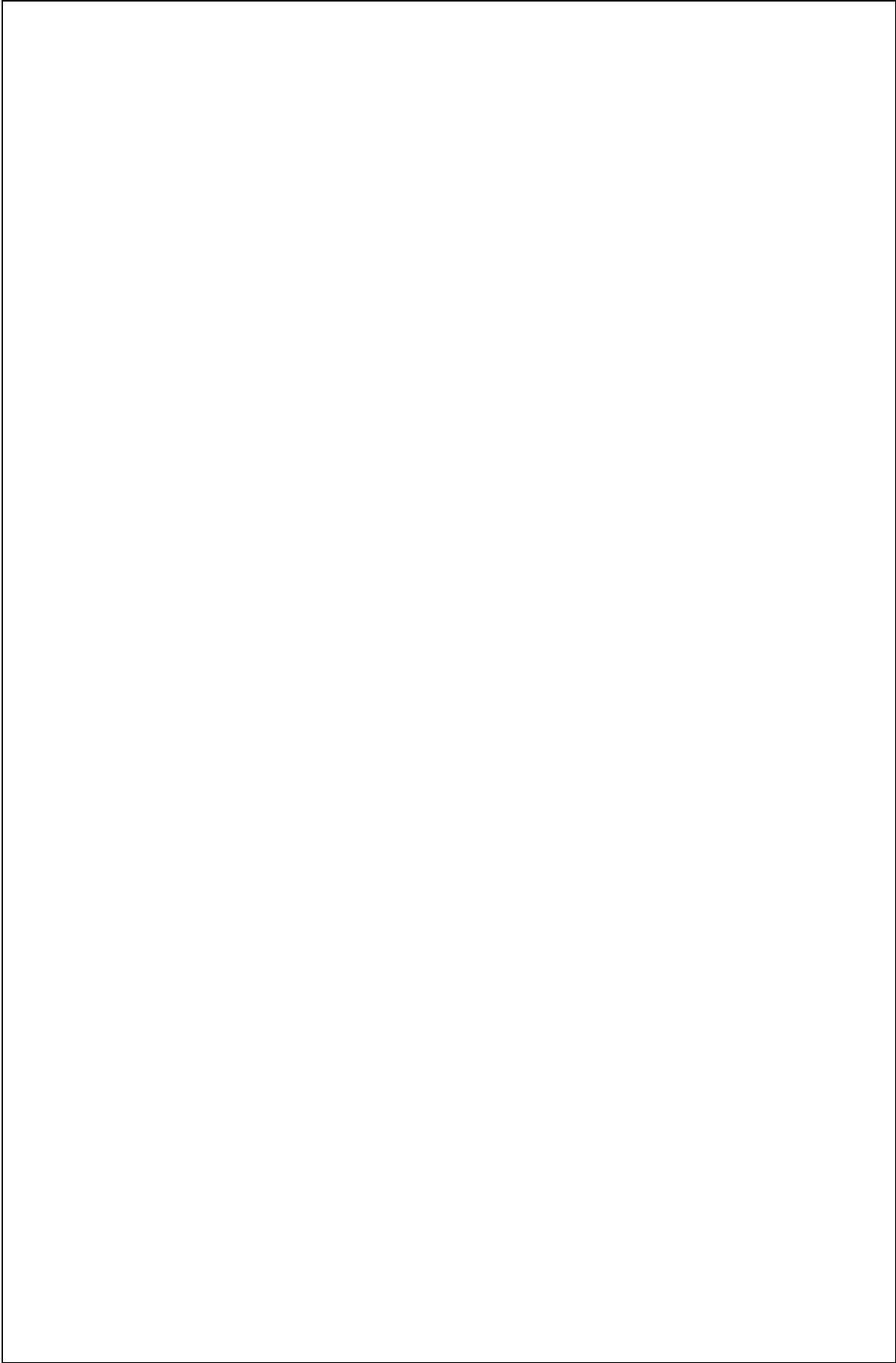
- c) Modifique el programa “`repl_server.js`” para que, desde una terminal donde se ejecute el programa `repl_client` se pueda desarrollar una sesión interactiva como la mostrada en la siguiente figura:

A screenshot of a Windows command prompt window titled "Node.js command prompt - node repl client". The window shows an interactive Node.js REPL session. The user enters various commands, and the REPL outputs the results. The commands and outputs are as follows:

```
$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) {console.log(x)}\r\nrepl.start('\>').context.show = f'
$node> logBase2(512)
9
$node> _
```

No se debe modificar el programa cliente. En el servidor se tendrán que modificar los parámetros necesarios (cambio del prompt, supresión de “undefined”, activación de colores) así como añadir a su contexto las funciones necesarias:

- **factorial**: función tal que dado n devuelve $n!$
- **fibonacci**: función tal que dado n devuelve el término n -ésimo de la sucesión de Fibonacci.
- **logaritmo**: función tal que dado n devuelve una función para calcular el logaritmo en base n (consultar el apartado 2.4.3, Clausuras, de la guía de estudio de este tema).
- **leeFichero**: función tal que dado un nombre de fichero, si existe el fichero, devuelve su contenido como String (ayuda: usar la función “`readFileSync`” del módulo “`fs`”). Si el fichero no existe, se generará el error `ENOENT`.



ACTIVIDAD 9

OBJETIVO: Comunicación en una aplicación multiusuario en red, y uso del módulo “Socket.IO”. En particular, se pretende comunicar mediante sockets todas las instancias de una sencilla aplicación ejecutable en red (cada instancia corresponde a un usuario activo) de manera que todos los usuarios de la aplicación puedan colaborar en un objetivo común.

ENUNCIADO: Considérese la siguiente aplicación de dibujo en una página web, constituida por un fichero HTML, “index.html”, donde se carga un fichero js, “script.js”. Los ficheros son:

```
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   let 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 });

```

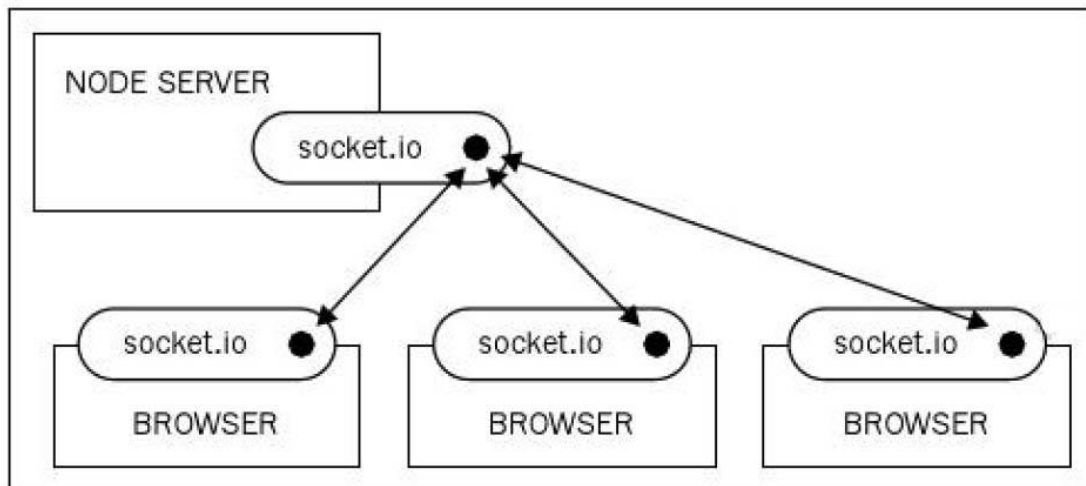
Esta aplicación, usando un objeto canvas y dando respuesta a los eventos de ratón, permite al usuario dibujar con trazo libre (véase, como ejemplo, la siguiente figura), pero es una aplicación monousuario. El objetivo de esta actividad es modificarla de forma que múltiples usuarios puedan cargarla en sus navegadores y el dibujo sea compartido (es decir, los trazos efectuados por cualquier usuario en su lienzo de dibujo se muestren inmediatamente en los lienzos del resto de usuarios conectados). La figura muestra un ejemplo con dos usuarios, pero la solución al problema no estará limitada a un número máximo de usuarios.



En la solución considerada se utilizará el módulo “Socket.IO”. Como no es un módulo estándar, preinstalado, se tendrá que instalar, mediante la orden:

```
> npm install socket.io
```

“Socket.IO” proporciona sockets bidireccionales, y se integra adecuadamente con los diferentes navegadores de Internet. Un diseño adecuado para la aplicación multiusuario considerada se muestra en la siguiente figura:



Es un diseño adecuado dado que se requiere que cada “browser” (navegador de cliente) comunique con un “node server” (servidor) al que envía sus acciones (trazos de dibujo) y del que recibe las acciones de los demás clientes. Como se aprecia, el cometido del servidor será retransmitir los mensajes recibidos de cada cliente al resto de clientes. Este tipo de comunicación se conoce como “broadcasting”.

Usando el módulo “Socket.IO”, para retransmitir mensajes hay que añadir el flag “broadcast” en las llamadas a los métodos “emit” y “send”. Por ejemplo, el siguiente fragmento de código correspondería a un servidor que retransmitiera mensajes a todos, excepto al socket que los envió:

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

Para la solución de la actividad, es necesario implementar un nuevo módulo, el servidor, y modificar el fichero cliente, “script.js”, y la página web que lo carga. En esta última, “index.html” solamente se necesita añadir la siguiente línea en la sección de includes:

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

Suponiendo que el fichero “socket.io.js” esté en el mismo directorio que “index.html”.

El servidor estará a la escucha en un determinado puerto, al que se conectará el socket (tipo “socket.io”) de cada script cliente. Las modificaciones a efectuar en “script.js” serán las siguientes:

- Declarar el socket y conectarlo al servidor.
- Modificar la función de callback del documento (variable “doc”) cuando se produce el evento “mousemove” para que, además de seguir dibujando el trazo del usuario local, envíe la información de dicho trazo a través del socket. La información a transmitir sería:

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

Y el evento asociado a este envío podría ser “mousemove” (si se elige también este nombre de evento para ser escuchado en el servidor).

- Por otra parte, el socket del cliente deberá estar a la escucha de los envíos del servidor. Estos envíos corresponderán a notificaciones de trazos de dibujo hechos por otros usuarios. Tendrán que tener un nombre de evento, por ejemplo, “moving”. La función de callback asociada a este evento “moving” en el socket del cliente deberá procesar adecuadamente los datos recibidos (el correspondiente objeto “data” tendrá las propiedades “x”, “y”, “drawing” e “id”, dado el formato indicado antes).

En este callback, en primer lugar, deberá comprobarse si los datos proceden de un usuario nuevo para, en tal caso, registrarlo:

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

A continuación, se dibujará el trazo correspondiente a los datos recibidos (desde la última posición del usuario, “clients[data.id]”, hasta su nueva posición, “data”):

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

Por último, el callback actualizará el estado del usuario:

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

Se pide implementar en el script del cliente, “script.js”, las modificaciones descritas:



En el servidor se ha de escribir el código necesario para:

- Establecer un socket (tipo “socket.io”) que escuche en el mismo puerto indicado en el script cliente.
- Para el objeto “io.sockets”, implementar un callback que responda al evento “connection” por parte de algún cliente.
- En este callback, para el socket identificado del cliente, implementar otro callback que responda al evento “mousemove”.
- En respuesta a este evento “mousemove”, se transmitirá (con *broadcasting*) un mensaje con el evento “moving” y los mismos datos que acompañen al evento “mousemove”.

Se pide implementar el servidor:



ACTIVIDAD 10

OBJETIVO: Comprobar si hay concurrencia en los servidores asincrónicos.

ENUNCIADO: El código siguiente muestra un cliente y un servidor:

```
1 // Client code.
2 const net = require('net')
3
4 // Check that an argument is given.
5 if (process.argv.length !== 3) {
6   console.log('An argument is required!!')
7   process.exit()
8 }
9
10 // Obtain the argument value.
11 const info = process.argv[2]
12
13 // The server is in our same machine.
14 const client = net.connect({port: 9000},
15   function() { // 'connect' listener
16     console.log('client connected')
17     console.log('sent:', info)
18     // This is sent to the server.
19     client.write(info+'\n')
20   });
21
22 client.on('data', function(data) {
23   // Write the received data to stdout.
24   console.log('received:\n'+data.toString())
25   // Close this connection.
26   client.end()
27 })
28
29 client.on('end', function() {
30   console.log('client disconnected')
31 })
```

```
1 // Server code.
2 const net = require('net')
3 let text = ""
4
5 const 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     // Read what the client sent.
12     c.on('data', function(data) {
```

```

13 // Print data to stdout.
14 console.log(data.toString())
15 text += data
16 // Print text to stdout.
17 console.log('text =\n'+text)
18 // Send the result to the client.
19 c.write(text + " ")
20 // Close connection.
21 c.end()
22 })
23 })
24
25 server.listen(9000,
26 function() { //listening' listener
27   console.log('server bound')
28 })

```

El cliente envía un texto variable (especificado como argumento al invocarlo) al servidor y el servidor concatena el texto recibido en una variable global. Es decir, la “invocación” (pues hay una petición y una respuesta) realizada por cada cliente logra que se añada una línea a un texto mantenido por el servidor y éste devuelve como resultado a cada cliente el valor actual de dicho texto.

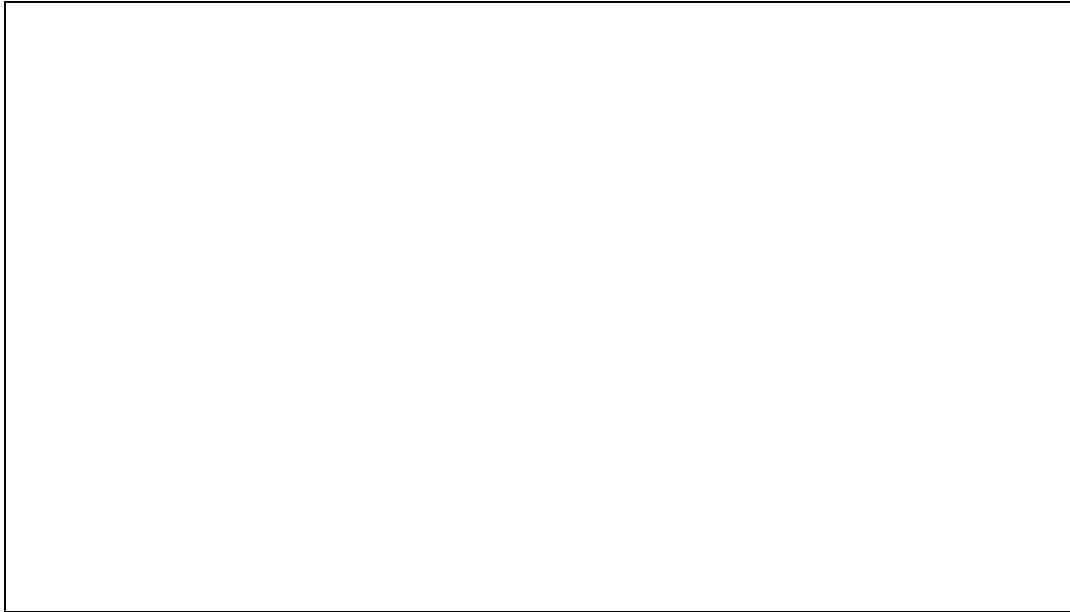
Se pide lo siguiente:

1. Una vez iniciado el proceso servidor mediante “`node Act10server`”, en un mismo ordenador, lance 4 procesos cliente (utilizando “`node Act10client '...' &`” para cada uno de ellos en algún sistema UNIX e indicando, en lugar de los puntos suspensivos, la línea de texto que el cliente enviará al servidor).

Compruebe si ha habido “condiciones de carrera” en esa ejecución (¿Cuál debería ser el valor final de la variable “text”? ¿Ha sido ese el valor obtenido? ¿Se obtiene el mismo valor si se repite la ejecución de los 4 procesos cliente?). Explique por qué.

2. ¿Qué ocurrirá si se sustituye la línea `"text += data;"` del servidor por esta otra: `"text += parseInt(data) ;"`? ¿Por qué ocurre eso?

(En esta variante debe suponerse que el argumento recibido por cada proceso cliente y enviado al servidor tendrá que ser una representación válida de un número entero, y no cualquier línea de texto).



3. Los programas mostrados en el enunciado y utilizados en los apartados 1 y 2 permiten comprobar fácilmente si hay "condiciones de carrera" o no.

Modifique los programas cliente y servidor para que sean capaces de hacer lo siguiente:

- El cliente deberá recibir dos argumentos desde la línea de órdenes. El primer argumento será un número entero (que podemos interpretar como un identificador del cliente) y el segundo argumento será una línea de texto (interpretable como la información que ese cliente tiene que proporcionar al servidor).
- El cliente construirá un objeto con los valores de esos 2 argumentos. Y lo serializará, usando la función **stringify()** del módulo JSON.
- El cliente mostrará en consola el objeto serializado, y lo enviará al servidor.
- El servidor utilizará un vector como variable global donde almacenar los datos que le envíen los clientes.
- Cada vez que el servidor reciba datos de un cliente, reconstruirá el objeto recibido, usando la función **parse()** del módulo JSON.
- A continuación, el servidor almacenará en el vector los datos recibidos de la siguiente forma: el primer dato del objeto (que hemos supuesto que representa un identificador del cliente) se usará como índice del vector, y el segundo dato del objeto se almacenará en el vector en la posición fijada por este índice.
- Por último, el servidor enviará al cliente el contenido completo del vector.
- El cliente, al recibir el vector, lo mostrará en consola.

Una vez realizada esta extensión, lance de nuevo 4 clientes con diferentes valores para sus argumentos, y compruebe de nuevo si hay condiciones de carrera y si el orden de ejecución de las peticiones coincide con el que se esperaba o no. Justifique los resultados obtenidos.