

# TSR: Activitats del Seminari 1

---

## ACTIVITAT 1

OBJECTIU: Aprofundir en la gestió del pas d'arguments en la invocació de funcions en JavaScript.

ENUNCIAT: Prenent com a base el següent codi<sup>1</sup>:

```
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) Descriga quin és l'eixida proporcionada pel programa anterior. Justifique si té o no algun efecte el passar més d'un argument en la línia 12.

---

<sup>1</sup> Tots els fitxers font llistats o esmentats en aquest document es troben en un arxiu "acts.zip" que es pot baixar des de PoliformaT.

- b) Suppose que ara canviem la línia 12 original del programa anterior per la següent, quina eixida proporciona el programa en aquest cas? per què?

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

- c) Suppose que ara canviem la línia 12 original del programa per la següent, quina eixida proporciona el programa ara? per què?

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

- d) A partir dels resultats obtinguts en els apartats anteriors, justifique si en JavaScript s'accepta i pot tenir algun efecte que es passen més arguments dels quals espera una funció determinada.

## ACTIVITAT 2

OBJECTIU: Adquirir habilitat en la programació d'esdeveniments de JavaScript.

ENUNCIAT: Siga el programa següent:

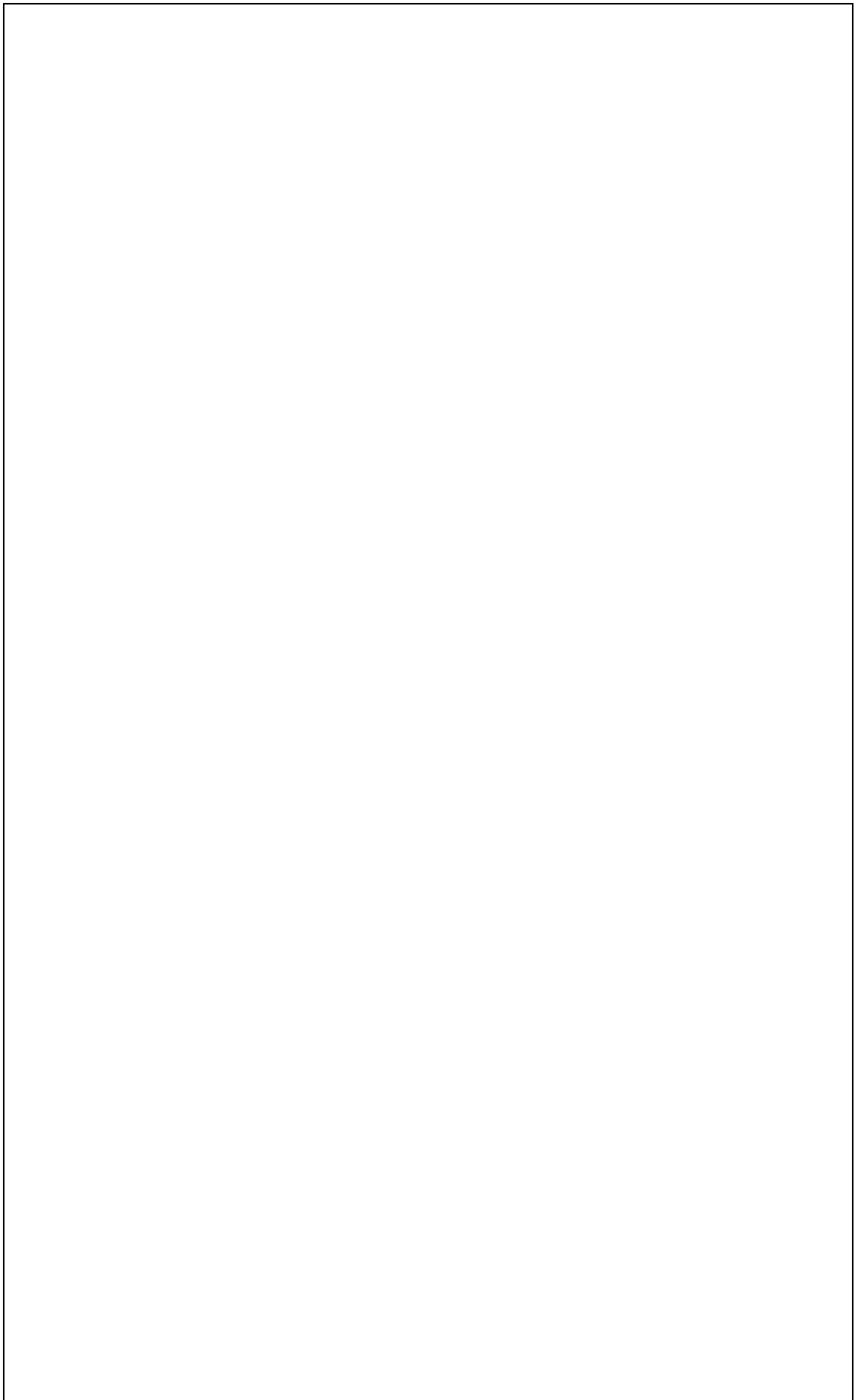
```
1 var ev = require('events');
2 var emitter = new ev.EventEmitter;
3 var i1 = "print";
4 var i2 = "read";
5 var books = [ "Walk Me Home", "When I Found You", "Jane's Melody", "Pulse" ];
6
7 // Constructor for class Listener.
8 function Listener(n1,n2) {
9   this.num1 = 0;
10  this.name1 = n1;
11  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(i1,i2);
27
28 // Listener is registered on the event emitter.
29 emitter.on(i1, function() {lis.event1()});
30 emitter.on(i2, function(x) {lis.event2(x)});
31 // There might be more than one listener for the same event.
32 emitter.on(i1, function() {console.log("Something has been printed!!");});
33
34 // Auxiliary function for generating i2.
35 var counter=0;
36 function generateEvent2() {
37   emitter.emit(i2,books[counter++ % books.length]);
38 }
39
40 // Generate the events periodically...
41 // First event generated every 2 seconds.
42 setInterval( function() {
43   emitter.emit(i1);
44 }, 2000 );
45 // Second event generated every 3 seconds.
46 setInterval( generateEvent2, 3000 );
```

Observe que amb ell s'amplia lleugerament l'exemple mostrat en l'secció 3.2 del seminari. A més del generador d'esdeveniments, en aquest nou programa s'ha creat un objecte "listener" que ofereix un mètode per a cada esdeveniment que pot disparar el generador. Aquest programa també il·lustra alguns aspectes més:

- L'esdeveniment "read" en aquest cas es genera amb un argument addicional (el títol de la novel·la a llegir). El codi necessari per a fer això es mostra a les línies:
  - 46: Per a determinar cada quant es genera l'esdeveniment (3 segons en aquest exemple).
  - 35-38: Declaració de la variable "counter" necessària per a mantenir el nombre d'esdeveniments generats i, sobre la base del seu valor, determinar quin missatge acompanya a l'esdeveniment com a argument.
  - 30: Ara el "listener" per a l'esdeveniment ha de tenir un paràmetre.
  - 20-23: I la funció utilitzada pel "listener" també.
- L'objecte Listener, el constructor del qual es mostra en les línies 8 a 13, manté ara el nombre de vegades que ha arribat a processar cadascun dels esdeveniments. L'increment dels comptadors es fa en cadascuna de les funcions instal·lades com a "listener" d'un esdeveniment (en les línies 16 i 22, respectivament).

Prenent aquest programa com a base, es demana que l'alumne desenvolupi un altre programa en el qual:

- Es generen els esdeveniments següents:
  - "un": Cada tres segons. Sense arguments.
  - "dos": Inicialment cada dos segons. Sense arguments.
  - "tres": Cada deu segons. Sense arguments.
- Hi haja un objecte Listener amb un mètode per a cada esdeveniment generat. En rebre cada esdeveniment, el Listener haurà de fer el següent:
  - "un": Escriure la cadena "Listener actiu: X esdeveniments de tipus un." en la seua eixida estàndard. Al missatge, en lloc d'una 'X' haurà de mostrar-se el nombre d'esdeveniments de tipus un rebuts fins al moment.
  - "dos": Escriure la cadena "Esdeveniment dos." en la seua eixida estàndard si el nombre d'esdeveniments "dos" rebuts fins ara és superior al nombre d'esdeveniments "un". Quan això ja no succeïska, escriurà "Hi ha més esdeveniments de tipus un".
  - "tres": Escriure un missatge "Esdeveniment tres." per eixida estàndard. A més, amb cada execució d'aquest manejador, es triplicarà la durada de l'interval entre dos esdeveniments consecutius de tipus "dos", fins que aquest valor siga 18 segons. A partir d'aquest moment, els esdeveniments de tipus "dos" es programaran cada 18 segons.  
L'operació "setInterval()" retorna un objecte que ha de ser emprat com a únic argument de "clearInterval()". Per a modificar la freqüència d'un esdeveniment, convé utilitzar "clearInterval()" abans d'establir la nova freqüència.



### ACTIVITAT 3

OBJECTIU: Entendre les clausures i el pas de funcions com a argument en JavaScript.

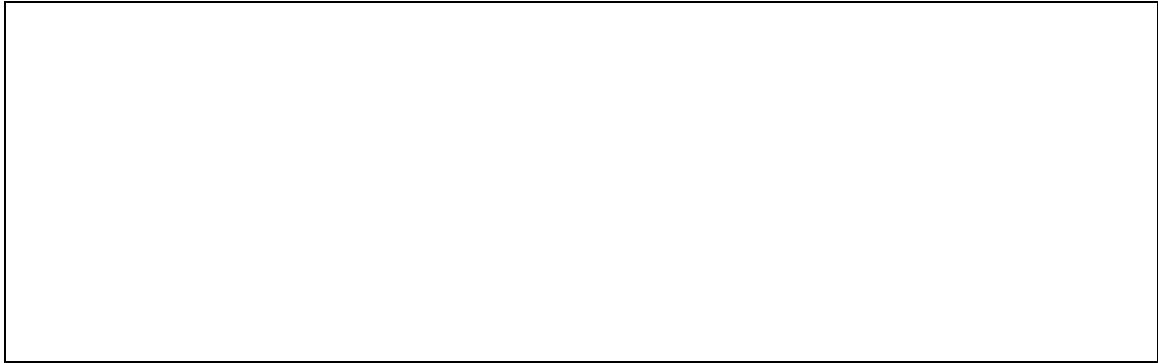
ENUNCIAT: Siga el programa següent:

```
1 function a3(x) {  
2     return function(i) {  
3         return x*i;  
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));
```

Execute el programa i diga quin és el resultat de l'execució de cadascuna de les línies següents, justificant per què es dona en cada cas:

a) línia 25.

b) línia 26.

A large, empty rectangular box with a thin black border, intended for a response.

c) línia 27.

A large, empty rectangular box with a thin black border, intended for a response.

d) línia 28.

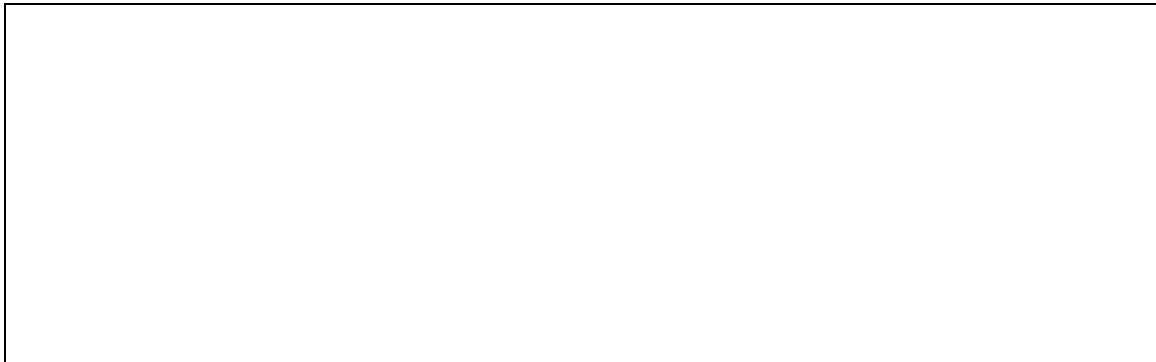
A large, empty rectangular box with a thin black border, intended for a response.

e) línia 29.

A large, empty rectangular box with a thin black border, intended for a response.



f) línia 30.



## ACTIVITAT 4

OBJECTIU: Estendre el servidor web mínim que s'ha utilitzat per a il·lustrar la funcionalitat del mòdul HTTP.

ENUNCIAT: En una de les seccions de la presentació es va mostrar el següent programa:

```
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/');
```

Aquest servidor retorna la cadena “Hello World” als navegadors que accedisquen a la URL <http://127.0.0.1:1337/> de l'ordinador en què s'executa. No és capaç de retornar res més.

Un servidor web hauria de ser capaç de retornar el contingut dels fitxers HTML especificats en la URL utilitzada, perquè el navegador sol·licitant pugui mostrar-los a l'usuari.

Per a aconseguir aquesta funcionalitat s'han de conèixer alguns detalls addicionals que es descriuen seguidament:

- El primer argument (“request” en aquest exemple, que és un objecte de la classe ClientRequest) del “callback” utilitzat en `http.createServer()` manté una propietat “url” amb la part de la URL que segueix al nom de màquina i port. Així, per exemple, si en el navegador es va escriure el següent:

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

la propietat **request.url** contindria la cadena “dir1/pagina.html”.

- Per a accedir als fitxers convé utilitzar el mòdul “fs”. En l'exemple citat en el punt anterior, el servidor hauria de llegir el contingut del fitxer “pagina.html” situat en el directori “dir1”. Això es pot fer utilitzant un codi similar al següent (assumint que es va fer un “var fs=require('fs');” en començar el programa):

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

Però això també comporta problemes ja que el nom de fitxer utilitzat com a primer argument hauria de ser un nom de ruta absolut. La funció emprada com a segon argument és un “callback” el primer paràmetre del qual serà un indicatiu de l'error que haja pogut donar-se i el segon paràmetre mantindria el contingut complet del fitxer.

En el nostre cas, si hi haguera un error caldria emplenar l'objecte “response” amb:

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

Ja que l'identificador d'error a retornar en la capçalera de la resposta HTTP (ServerResponse) en aquest cas és el 404. La segona sentència proporciona el text del missatge d'error. Per contra, quan no hi haja cap error, convindria emplenar l'objecte “response” amb:

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

En aquesta situació, el 200 en la capçalera indica que la petició ha pogut resoldre's sense problemes. La segona sentència bolca el contingut del fitxer (“content” era el segon paràmetre del “callback” i arreplegava el contingut del fitxer en el mètode `readFile()` en la `ServerResponse`.

- Per a formar correctament el nom de ruta absolut del fitxer HTML a retornar al navegador, es podrà utilitzar el mètode `join()` del mòdul `PATH`. En NodeJS existeix un atribut “global” anomenat “\_\_dirname” que conté el nom de ruta absolut del directori actual. Així, s'afegirà una sentència “var path=import('path');” en la part inicial del programa i s'utilitzarà una invocació similar a la següent:

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

per a obtenir el nom de ruta absolut.

- En la majoria dels servidors web, quan únicament s'utilitza l'adreça de l'ordinador en la URL (sense especificar cap nom de fitxer), se sol accedir a un fitxer “index.html”. Això haurà de programar-se explícitament en cas que es vulga obtenir aquesta funcionalitat.
- D'altra banda, una URL pot especificar únicament l'adreça d'un lloc web (<https://intranet.upv.es>) o l'adreça completa d'una pàgina dins del lloc (<http://www.upv.es/organizacion/escuelas-facultades/index-es.html>), però també pot contenir més informació com, per exemple, paràmetres de consulta (considere's: [https://intranet.upv.es/pls/soalu/sic\\_menu.personal?p\\_idioma=c](https://intranet.upv.es/pls/soalu/sic_menu.personal?p_idioma=c), en aquesta pàgina la informació de consulta és “?P\_IDIOMA=c”. El mètode `parse()` del mòdul “url”

permet obtenir l'objecte codificat en la cadena d'una URL. La propietat query emmagatzema la informació corresponent als paràmetres de consulta.

Per exemple (assumint un “var url=require('url');” previ):

```
url.parse(https://intranet.upv.es/pls/soalu/sic\_menu.personal?p\_idioma=c).query
```

proporciona “P\_IDIOMA=c”.

- La cadena corresponent als paràmetres de consulta pot ser molt més complexa (per exemple, considere's la següent URL incompleta:

```
http://www.booking.com/searchresults.es.html?src=index& ...  
&ss=València&checkin\_monthday=1&checkin\_year\_month=2015-  
8&checkout\_monthday=2&checkout\_year\_month=2015-8& ...).
```

Convindria utilitzar algun mètode que permetia extraure paràmetres concrets, i això ho fa el mètode parse() del mòdul “querystring”.

Per exemple (amb un “require” previ del mòdul “querystring”):

```
querystring.parse(http://www.booking.com/searchresult  
s.es.html?src=index& ...  
&ss=València&checkin\_monthday=1&checkin\_year\_month=20  
15-8& ...).ss
```

proporciona “València”.

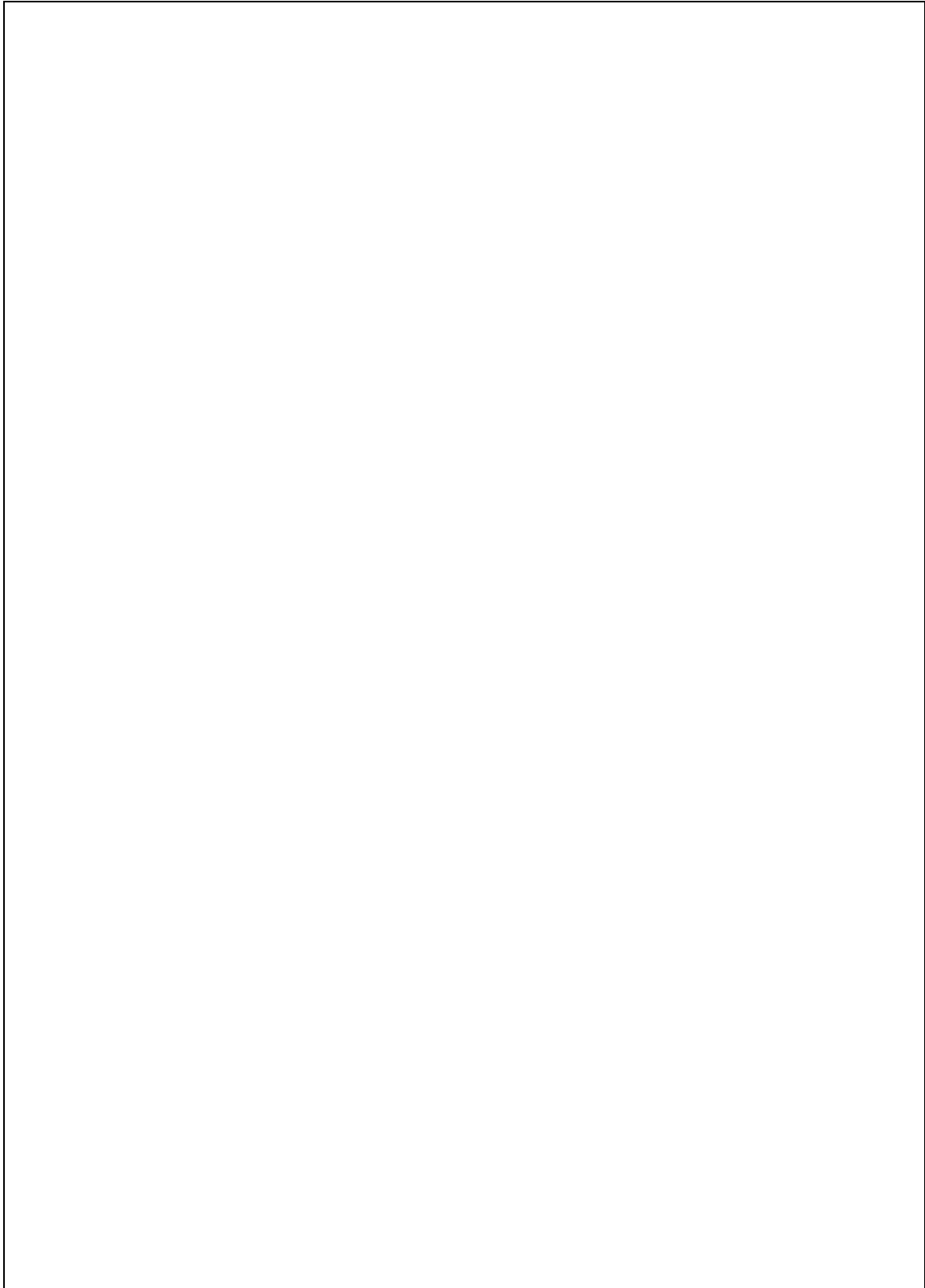
- Per a il·lustrar un ús bàsic dels paràmetres de consulta, considere's el següent senzill servidor:

```
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 = 'ics';  
8   var y = 'i grega';  
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');
```

Es demana estendre el programa mostrat en començar aquest enunciat perquè siga capaç de gestionar un paràmetre “consulta” en la URL utilitzada i, en funció del seu valor, mostrar diferents informacions:

- Si el valor del paràmetre és “time”, el servidor proporcionarà la data i hora actual.
- Si el valor del paràmetre és “dir”, el servidor proporcionarà el nom del seu directori i un llistat amb els noms de tots els fitxers continguts en el mateix.
- Qualsevol altre valor de paràmetre serà interpretat com un nom de fitxer. Si existeix un fitxer amb aqueix nom en el directori actual, el servidor ho llegirà i proporcionarà el

seu contingut. En cas contrari, proporcionarà un missatge d'error o avís, indicant que no pot facilitar la informació sol·licitada.



## ACTIVIDAD 5

OBJECTIU: Entendre que els “callbacks” no sempre són asincrònics.

ENUNCIAT: Només hi ha un fil d'execució en Node.js. Això implica que no caldrà preocupar-nos sobre la protecció de variables compartides amb locks o qualsevol altre mecanisme de control de concurrència.

No obstant això, hi ha alguns casos en els quals necessitarem ser acurats.

Un bon principi per a raonar sobre la lògica d'un programa asincrònic és considerar que TOTS els seus callbacks s'executaran en un torn posterior al que ara executa el codi que els passa com a arguments.

Considere el codi que es mostra a continuació:

```
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();

```

Quan siga executat<sup>2</sup>, esperarem aquesta eixida:

```

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

```

TOTS els missatges “processed” apareixen després de TOTS els missatges “processing”, tal com s'esperava (els “callbacks” semblen ser cridats en torn futur).

No obstant això, considere aquesta variació:

Substituïska la línia 4 del codi anterior (var rolodex={}; ) per la següent:

```

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

```

L'eixida que obtindríem seria:

```

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

```

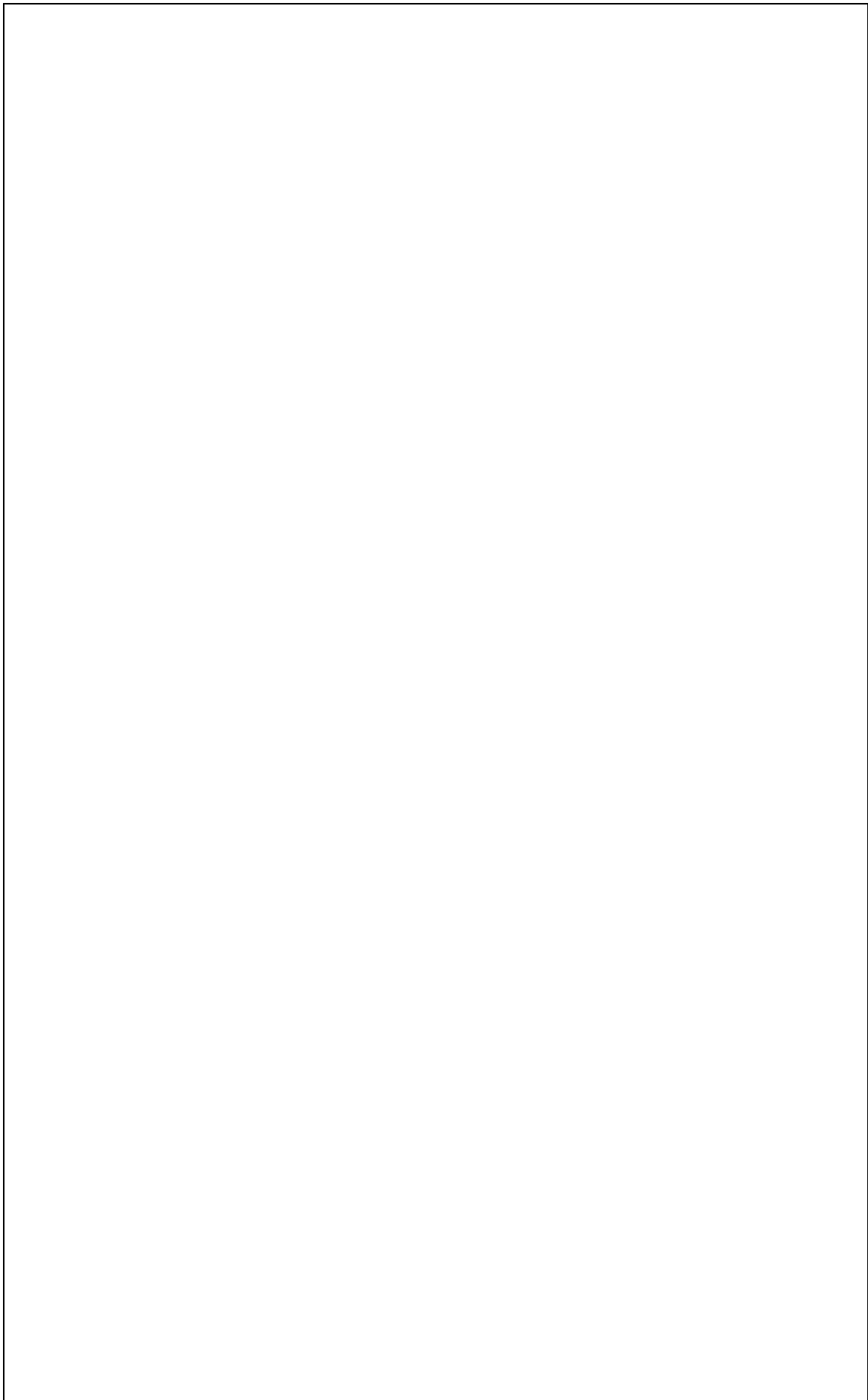
<sup>2</sup> Per a executar el programa correctament haurà d'existir un fitxer anomenat “rolodex” en el mateix directori. El fitxer ha de contenir algunes línies de text. En elles se cercaran les cadenes contingudes en el vector “testNames”.

Observe que ara NO TOTS els missatges “processed” es mostren després de TOTS els missatges “processing”. La raó és que un dels callbacks ha sigut executat EN EL MATEIX torn que la funció que el va passar.

Depenent de la situació, això podria introduir problemes difícils de percebre, en cas que el codi que estableix els callbacks i un o més dels callbacks interferiren en el seu accés a una mateixa part de l'estat del procés, deixant-la inconsistent.

Aquesta situació (inconvenient) sempre apareixerà si els callbacks confien que el seu invocador prepararà els seus contextos abans que ells inicien la seua execució.

Modifique el programa anterior, utilitzant promeses, per a garantir que es respecte l'ordre d'execució esperat.





## ACTIVITAT 6

OBJECTIU: Utilitzar adequadament “callbacks” i clausures.

ENUNCIAT: Per a accedir a fitxers, Node.js proporciona el mòdul “fs”. Escriga un programa que reba un nombre variable de noms de fitxers des de la línia d'ordres i que escriga en pantalla el nom del fitxer més gran de tots ells, així com la seua longitud en bytes. Per a fer això, utilitze la funció `fs.readFile()` del mòdul “fs”, la documentació de la qual està disponible en [https://nodejs.org/api/fs.html#fs\\_fs\\_readfile\\_filename\\_options\\_callback](https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback). No utilitze les variants sincròniques de la funció o altres funcions del mòdul “fs”.

Per a gestionar els arguments rebuts des de la línia d'ordres haurà d'utilitzar el vector `process.argv` ([https://nodejs.org/api/process.html#process\\_process\\_argv](https://nodejs.org/api/process.html#process_process_argv)).

Si es reben diversos arguments, necessitarà utilitzar clausures perquè el “callback” accedisca correctament a la posició adequada del vector `process.argv`.

## ACTIVITAT 7

OBJECTIU: Aprofundir en la conversió de “callbacks” en promeses.

ENUNCIAT: El programa que es mostra seguidament implanta un servidor de fitxers (per a fitxers de text únicament) que hauria de proporcionar tres operacions als seus clients: UPLOAD, DOWNLOAD i REMOVE. Els programes necessaris per a sol·licitar aquestes operacions (Act7upload.js, Act7download.js i Act7remove.js) es troben en PoliformaT on també s'inclou una versió d'aquest programa servidor que utilitza promeses en lloc 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 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);
```

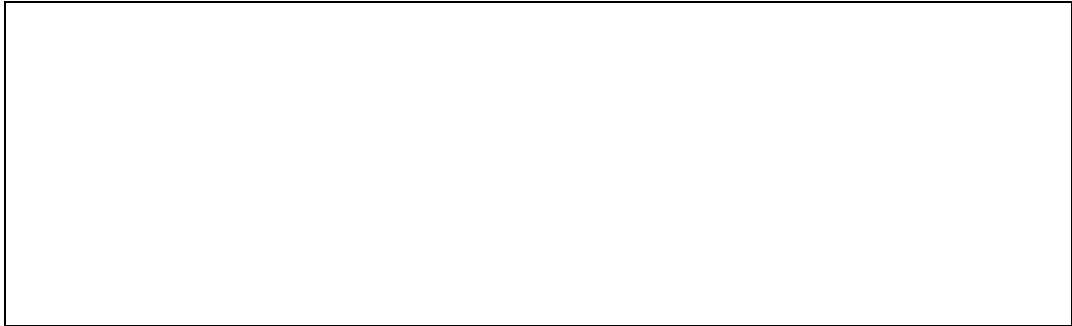
```

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(i) {
89     console.log("Error: %s", i);
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(i) {
100     console.log("Server error: %s", i.code);
101 });

```

Es demana el següent:

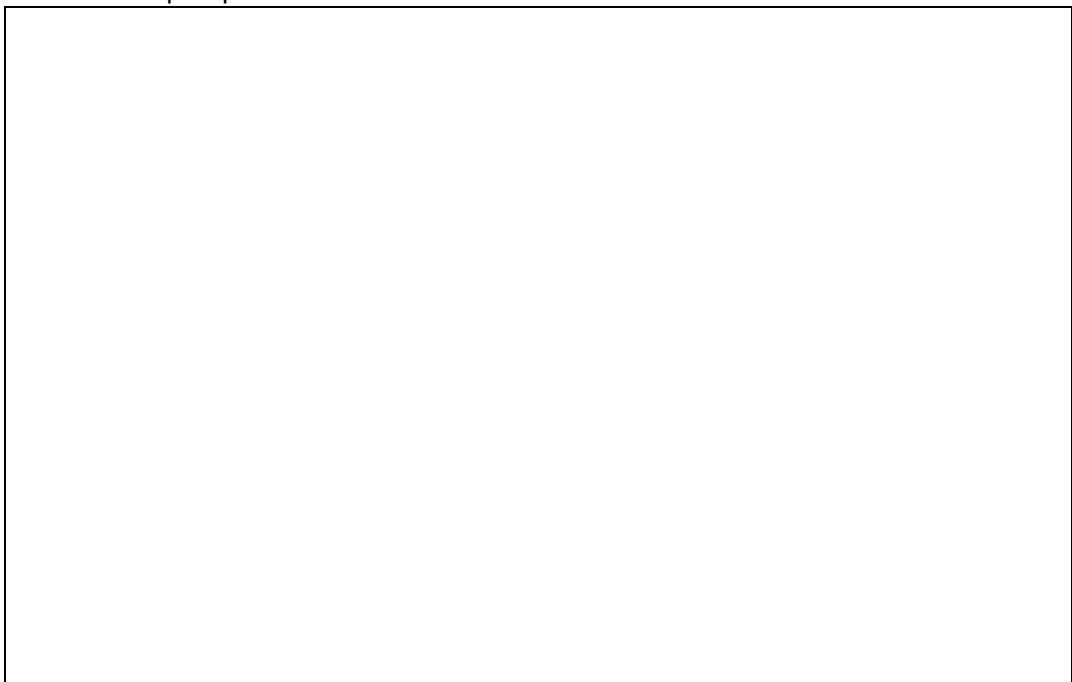
- Implante el codi necessari perquè l'operació REMOVE realitzi el seu treball correctament. Indique en la seua resposta entre quines línies del programa original han d'afegir-se les línies que componguen la seua solució.



- b) Realitze una ampliació equivalent en el programa basat en promeses (Act7serverPromise.js).



- c) Explique en quin programa ha hagut de fer més treball per a elaborar la solució sol·licitada i per quin motiu.



## ACTIVITAT 8

OBJECTIU: Aprofundir en la conversió de “callbacks” en promeses.

ENUNCIAT: Prenent com a base la solució completa basada en promeses per a l'activitat 7, transforme el programa perquè les promeses siguin construïdes mitjançant el constructor de promeses en lloc de ser generades mitjançant el mètode `promisify()`.

## ACTIVITAT 9

OBJECTIU: Executar codi de manera interactiva en un servidor remot usant els mòduls “net” i “repl”.

ENUNCIAT: El mòdul REPL (Read-Eval-Print Loop) representa el shell de Node. El shell es pot activar directament en la línia d'ordres d'una terminal escrivint:

```
> node
```

A més, el mòdul “repl”, si s'usa en el codi d'un programa Node, permet invocar el shell i executar sentències interactivament. Considere el següent programa:

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

Un exemple d'execució d'aquest programa:

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

Com s'aprecia, en el context del shell invocat amb “repl” existeix una funció referenciada com a “show” que és equivalent a “console.log”. En aquesta execució també s'adverteix l'aparició del valor “undefined”, això es pot evitar establint una propietat del mòdul (consultar <https://nodejs.org/api/repl.html>).

El shell del mòdul “repl” pot ser invocat remotament. Considere el codi dels programes, repl\_client.js i repl\_server.js, que es mostren a continuació:

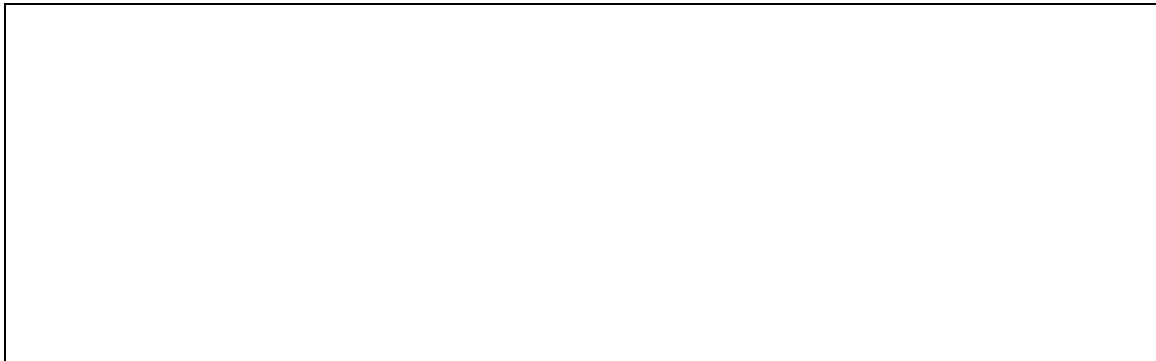
```
1 /* repl_client.js */
2
3 var net = require('net')
4 var sock = net.connect(8001)
5
6 process.stdin.pipe(sock)
7 sock.pipe(process.stdout)
```

```
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)
```

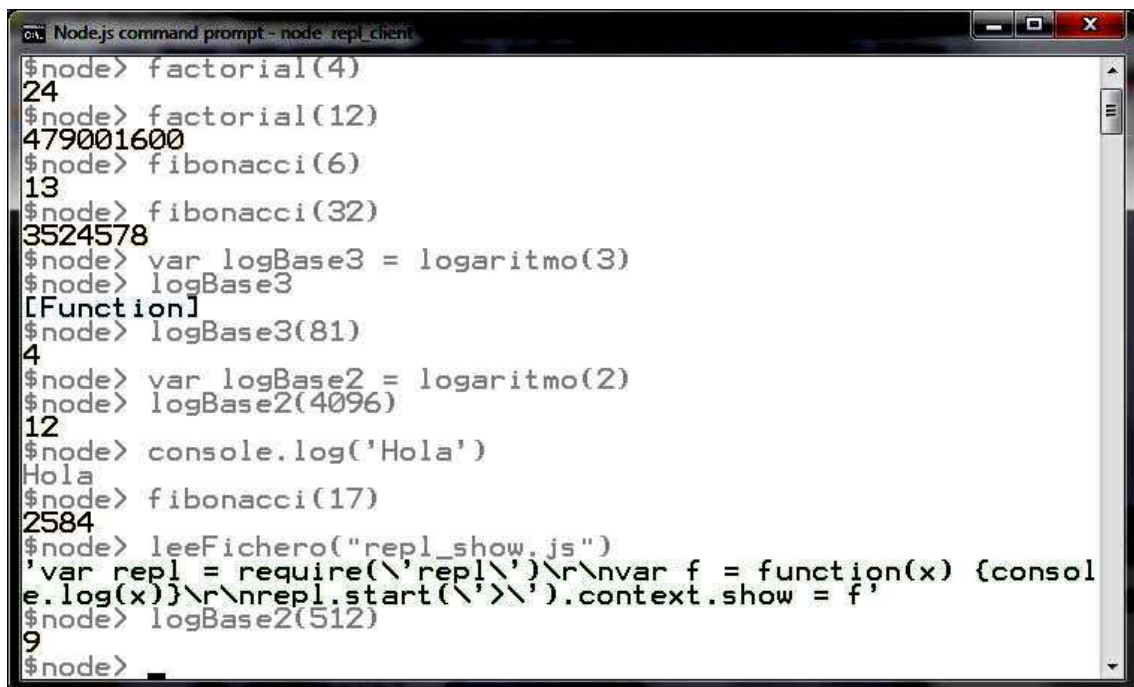
Es demana estudiar el funcionament d'aquests programes consultant, si cal, l'API dels mòduls utilitzats i executant els programes, interactuant amb el shell remot accessible des del programa client.

- a) Explique el funcionament de client i servidor, precisant com flueix la informació i on es realitza el processament.

- b) Si en la terminal on s'executa el client s'escriu **"console.log(process.argv)"**, què s'obté?, per què?



- c) Modifique el programa "repl\_server.js" perquè, des d'una terminal on s'execute el programa repl\_client es pugui desenvolupar una sessió interactiva com la mostrada en la següent figura:

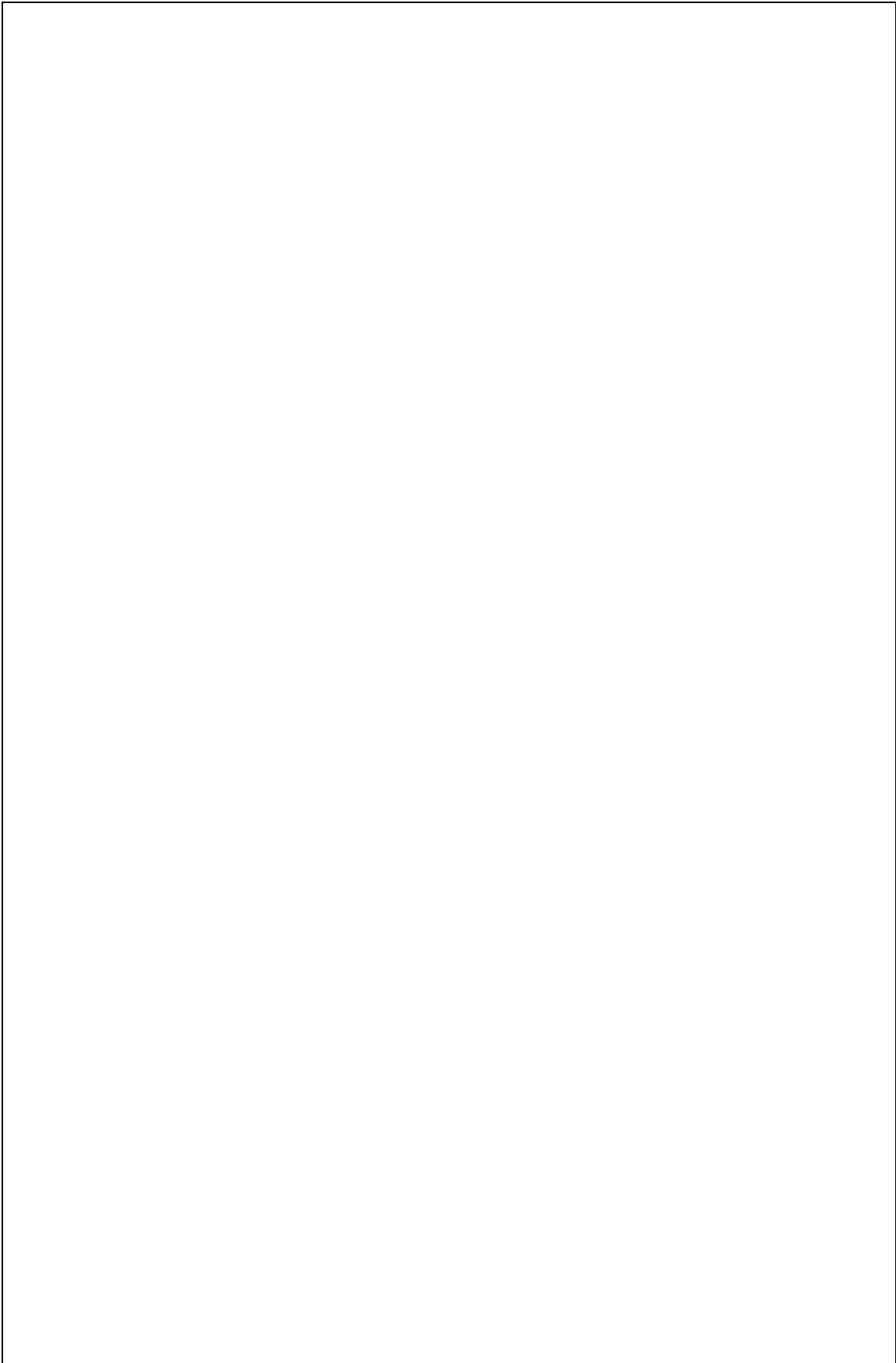


```
Node.js command prompt - node repl_client
$node> factorial(4)
24
$node> factorial(12)
479001600
$node> fibonacci(6)
13
$node> fibonacci(32)
3524578
$node> var logBase3 = logaritmo(3)
$node> logBase3
[Function]
$node> logBase3(81)
4
$node> var logBase2 = logaritmo(2)
$node> logBase2(4096)
12
$node> 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 s'ha de modificar el programa client. En el servidor s'hauran de modificar els paràmetres necessaris (canvi del prompt, supressió de "undefined", activació de colors) així com afegir al seu context les funcions necessàries:

- **factorial**: funció tal que donat  $n$  retorne  $n!$
- **fibonacci**: funció tal que donat  $n$  retorne el terme enèsim de la successió de Fibonacci.
- **logaritme**: funció tal que donat  $n$  retorne una funció per a calcular el logaritme en base  $n$  (consultar l'apartat 2, Clausures, de la guia d'estudi del seminari).
- **leeFichero**: funció tal que donat un nom de fitxer, si existeix el fitxer, retorne el seu contingut com String (ajuda: usar la funció "readFileSync" del mòdul "fs"). Si el fitxer no existira, es generarà l'error ENOENT.





## ACTIVITAT 10

OBJECTIU: Comunicació en una aplicació multiusuari en xarxa i ús del mòdul "Socket.IO". En particular, es pretén comunicar mitjançant sockets totes les instàncies d'una senzilla aplicació executable en xarxa (cada instància correspon a un usuari actiu) de manera que tots els usuaris de l'aplicació puguin col·laborar en un objectiu comú.

ENUNCIAT: Considere's la següent aplicació de dibuix en una pàgina web, constituïda per un fitxer HTML, "index.html", on es carrega un fitxer js, "script.js". Els fitxers són:

```
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>
```

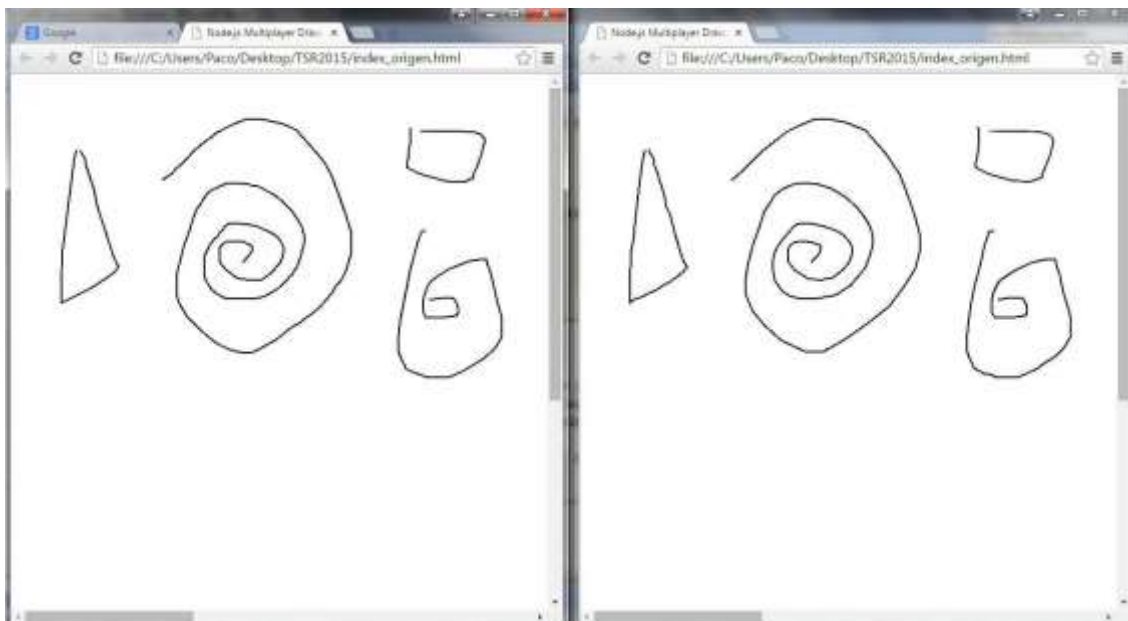
```
1 $(function(){
2   // This demo depends on the canvas element
3   if(!('getContext' in document.createElement('canvas'))){
4     alert('Sorry, it looks like your browser does not support canvas!');
5     return false;
6   }
7   var doc = $(document),
8       win = $(window),
9       canvas = $('#paper'),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

```

17 canvas.on('mousedown',function(i){
18     i.preventDefault();
19     drawing = true;
20     prev.x = i.pageX;
21     prev.y = i.pageY;
22 });
23 doc.bind('mouseup mouseleave',function(){
24     drawing = false;
25 });
26 doc.on('mousemove',function(i){
27     // Draw a line for the current user's movement
28     if(drawing){
29         drawLine(prev.x, prev.y, i.pageX, i.pageY);
30         prev.x = i.pageX;
31         prev.y = i.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 });

```

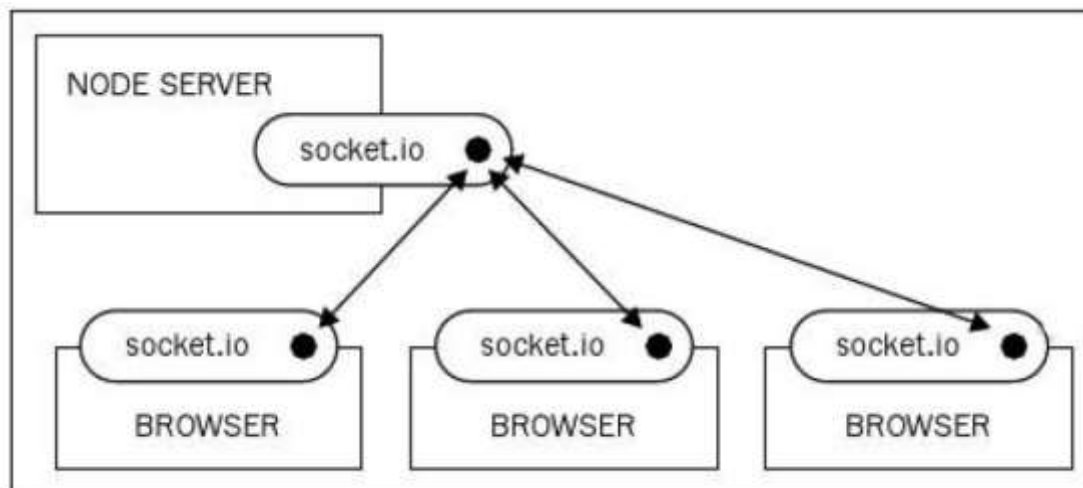
Aquesta aplicació, usant un objecte canvas i donant resposta als esdeveniments de ratolí, permet a l'usuari dibuixar amb traç lliure (vegeu, com a exemple, la següent figura), però és una aplicació monousuari. L'objectiu d'aquesta activitat és modificar-la de manera que múltiples usuaris puguin carregar-la en els seus navegadors i el dibuix siga compartit (és a dir, els traços efectuats per qualsevol usuari en el seu llenç de dibuix es mostren immediatament en els llenços de la resta d'usuaris connectats). La figura mostra un exemple amb dos usuaris, però la solució al problema no estarà limitada a un nombre màxim d'usuaris.



En la solució considerada s'utilitzarà el mòdul “SOCKET.IO”. Com no és un mòdul estàndard, preinstal·lat, s'haurà d'instal·lar, mitjançant l'ordre:

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

“SOCKET.IO” proporciona sockets bidireccionals, i s'integra adequadament amb els diferents navegadors d'Internet. Un disseny adequat per a l'aplicació multiusuari considerada es mostra en la següent figura:



És un disseny adequat atès que es requereix que cada “browser” (navegador de client) comuniqui amb un “node server” (servidor) al que envia les seues accions (traços de dibuix) i del que rep les accions dels altres clients. Com s'aprecia, la feina del servidor serà retransmetre els missatges rebuts de cada client a la resta de clients. Aquest tipus de comunicació es coneix com “broadcasting”.

Usant el mòdul “Socket.IO”, per a retransmetre missatges cal afegir el flag “broadcast” en les crides als mètodes “emit” i “send”. Per exemple, el següent fragment de codi correspondria a un servidor que retransmetria missatges a tots, excepte al socket que els va enviar:

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

Per a la solució de l'activitat, és necessari implantar un nou mòdul, el servidor, i modificar el fitxer client, “script.js”, i la pàgina web que ho carrega. En aquesta última, “index.html” solament es necessita afegir la següent línia en la secció d'includes:

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

Suposant que el fitxer “socket.io.js” estiga en el mateix directori que “index.html”.

El servidor estarà a l'escolta en un determinat port, al que es connectarà el socket (tipus “socket.io”) de cada script client. Les modificacions a efectuar en “script.js” seran les següents:

- Declarar el socket i connectar-lo al servidor.
- Modificar la funció de callback del document (variable “doc”) quan es produeix l'esdeveniment “mousemove” perquè, a més de seguir dibuixant el traç de l'usuari local, envii la informació d'aquest traç a través del socket. La informació a transmetre seria:

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

I l'esdeveniment associat a aquest enviament podria ser “mousemove” (si es tria també aquest nom d'esdeveniment per a ser escoltat en el servidor).

- D'altra banda, el socket del client haurà d'estar a l'escolta dels enviaments del servidor. Aquests enviaments correspondran a notificacions de traços de dibuix fets per altres usuaris. Hauran de tenir un nom d'esdeveniment, per exemple, “moving”. La funció de callback associada a aquest esdeveniment “moving” en el socket del client haurà de processar adequadament les dades rebudes (el corresponent objecte “data” tindrà les propietats “x”, “y”, “drawing” i “id”, donat el format indicat abans).

En aquest callback, en primer lloc, haurà de comprovar-se si les dades procedeixen d'un usuari nou per a, en tal cas, registrar-lo:

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

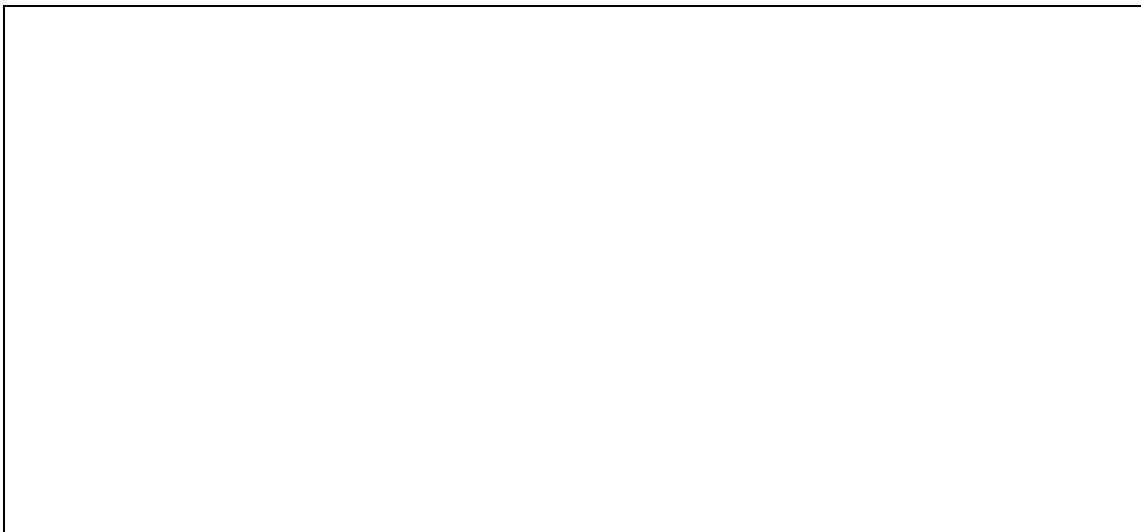
A continuació, es dibuixarà el traç corresponent a les dades rebudes (des de l'última posició de l'usuari, “clients[data.id]”, fins a la seua nova posició, “data”):

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

Finalment, el callback actualitzarà l'estat de l'usuari:

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

Es demana implantar en el script del client, “script.js”, les modificacions descrites:

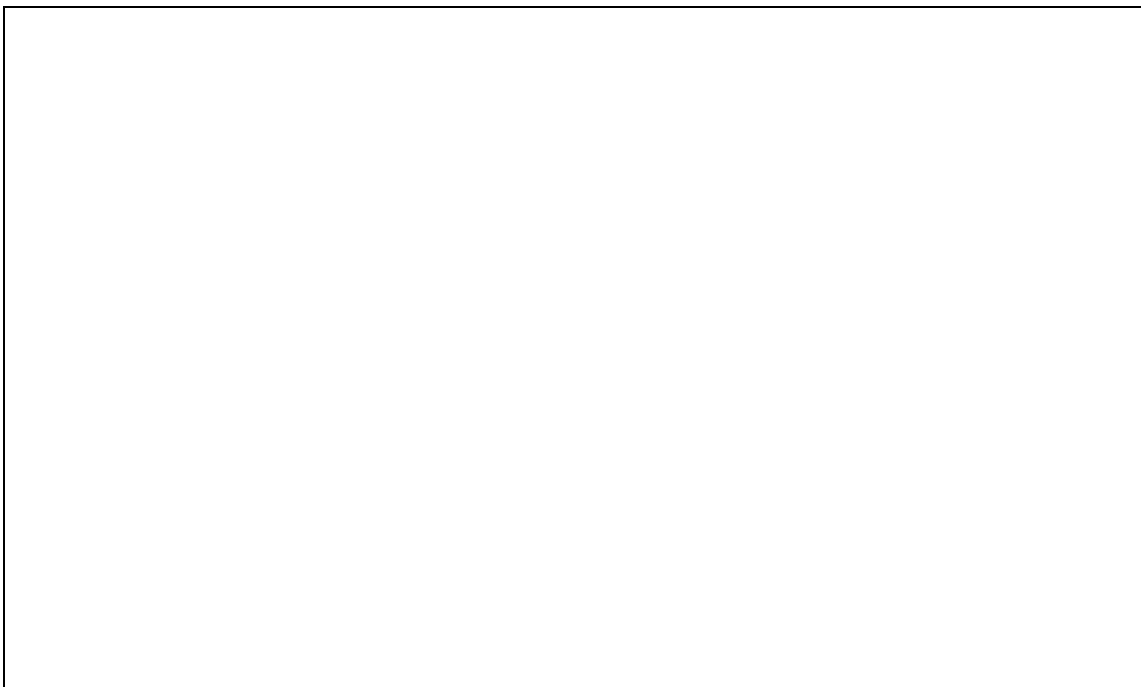




En el servidor s'ha d'escriure el codi necessari per a:

- Establir un socket (tipus “socket.io”) que escolte en el mateix port indicat en el script client.
- Per a l'objecte anterior (“io.sockets”), implementar un callback que responga a l'esdeveniment “connection” per part d'algun client.
- En aquest callback, per al socket identificat del client, implantar un altre callback que responga a l'esdeveniment “mousemove”.
- En resposta a aquest esdeveniment “mousemove”, es transmetrà (amb *broadcasting*) un missatge amb l'esdeveniment “moving” i les mateixes dades que acompanyen a l'esdeveniment “mousemove”.

Es demana implantar el servidor:



## ACTIVITAT 11

OBJECTIU: Comprovar si hi ha concurrència en els servidors asincrònics.

ENUNCIAT: El codi següent mostra un client i un servidor que modifiquen l'exemple mostrat en la fulla 44 de la presentació:

```
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 });
```

En aquest cas, en lloc d'enviar les paraules “Hello” i “world” entre els processos client i servidor, el client envia un text variable (especificat com a argument en invocar-ho) al servidor i el servidor concatena el text rebut en una variable global. És a dir, la “invocació” (ja que hi ha una petició i una resposta) realitzada per cada client aconsegueix que s'afiji una línia a un text mantingut pel servidor (i “compartit” per tots els clients) i aquest retorna com a resultat a cada client el valor actual d'aquest text.

Es demana el següent:

1. Una vegada iniciat el procés servidor mitjançant “`node Act11server`”, en un mateix ordinador, llance 4 processos client (utilitzant “`node Act11client '...' &`” per a cadascun d'ells en algun sistema UNIX i indicant, en lloc dels punts suspensius, la línia de text que el client enviarà al servidor).

Comprove si hi ha hagut “condicions de carrera” en l'execució (Quin hauria de ser el valor final de la variable “text”? Ha sigut aquest el valor obtingut? S'obté el mateix valor si es repeteix l'execució dels 4 processos client?). Explique per què.

2. Què ocurrerà si se substitueix la línia “`text += data;`” del servidor per aquesta altra: “`text += parseInt(data);`”? Per què ocorre això?

(En aquesta variant ha de suposar-se que l'argument rebut per cada procés client i enviat al servidor haurà de ser una representació vàlida d'un nombre enter, i no qualsevol línia de text).



3. Els programes mostrats en l'enunciat i utilitzats en els apartats 1 i 2 permeten comprovar fàcilment si hi ha “condicions de carrera” o no.

Modifiqui els programes client i servidor perquè siguin capaços de fer el següent:

- El client haurà de rebre dos arguments des de la línia d'ordres. El primer argument serà un nombre enter (que podem interpretar com un identificador del client) i el segon argument serà una línia de text (interpretable com la informació que el client ha de proporcionar al servidor).
- El client construirà un objecte amb els valors dels 2 arguments i el serialitzarà, usant la funció **stringify()** del mòdul JSON.
- El client mostrarà en consola l'objecte format, i l'enviarà al servidor.
- El servidor utilitzarà un vector com a variable global on emmagatzemar les dades que li envien els clients.
- Cada vegada que el servidor reba dades d'un client, reconstruirà l'objecte rebut, usant la funció **parse()** del mòdul JSON.
- A continuació, el servidor emmagatzemarà en el vector les dades rebudes de la següent forma: la primera dada de l'objecte (que hem suposat que representa un identificador del client) s'usarà com a índex del vector, i la segona dada de l'objecte s'emmagatzemarà en el vector en la posició fixada per aquest índex.
- Finalment, el servidor enviarà al client el contingut complet del vector.
- El client, en rebre el vector, el mostrarà en consola.

Una vegada realitzada aquesta extensió, llance de nou 4 clients amb diferents valors per als seus arguments, i comprova de nou si hi ha condicions de carrera i si l'ordre d'execució de les peticions coincideix amb el que s'esperava o no. Justifique els resultats obtinguts.

