

*This test provides 2 points (20%) of the global grade in NIST and it consists of 20 questions. Each question has 4 choices and only one of them is correct. Each correct answer provides 0.1 points and each error -0.033 points. Answers must be given in a separate answer sheet. The exercise mark is 1 point and its answer must be given in the blank sheets delivered with this wording.*

- 1 *In cloud computing, PaaS provides DO NOT have to:*
- a Manage service deployment and upgrading.
  - b Set elasticity points.
  - c Monitor workload-related parameters.
  - d **Install virtual machines.**
- 2 *In the Wikipedia development, reverse proxies have been used in order to:*
- a Guarantee reliability and fault tolerance in its web servers.
  - b Accelerate dynamic requests to the database management servers.
  - c Become intermediaries (i.e., brokers) between web servers and database management servers, balancing the workload among the database server replicas.
  - d **Provide endpoints to service users, balancing the workload among web servers.**
- 3 *An asynchronous server (AS) may scale better than a multi-threaded server (MTS) because...*
- a AS guarantees strong consistency and MTS provides weak consistency.
  - b AS uses persistent communication and MTS must use transient (i.e., non-persistent) communication.
  - c **Some MTS activities may block when multiple threads access the same shared data element.**
  - d MTS cannot use asynchronous communication.
- 4 *Regarding the distributed system model:*
- a It is composed of a set of cooperating agents. Those agents do not have any independent state.
  - b A distributed system is not necessarily a networked system.
  - c **Each agent is a sequential process.**
  - d It compels all system agents to proceed at the same pace.

- 5 Let us assume that this program runs in a directory that does not have any file with a .txt extension. Choose the correct program output:

```
const fs = require('fs')
function afterRead(n) {
  return function(err, data) {
    if (err) console.error(n, 'not found')
    else console.log(data.toString())
  }
}
function startReading(n) {
  return function() { fs.readFile(n, afterRead(n)) }
}
for (var i = 1; i <= 2; i++) {
  fs.writeFileSync('data' + i + '.txt', 'Hello ' + i)
}
for (var i = 1; i <= 3; i++) {
  var filename = 'data' + i + '.txt'
  var time = 100 - 10 * i
  setTimeout(startReading(filename), time)
}
console.log("root(" + i + ") = ", Math.sqrt(i))
```

- a Hello 1  
Hello 2  
data3.txt not found  
root(4) = 2
- b root(4) = 2  
data3.txt not found  
data3.txt not found  
data3.txt not found  
data3.txt not found
- c root(4) = 2  
Hello 1  
Hello 2  
data3.txt not found
- d root(4) = 2**  
**data3.txt not found**  
**Hello 2**  
**Hello 1**

- 6 Let us assume that this program runs in a directory that does not have any file with a .txt extension. Choose the correct program output:

```
const fs = require('fs')
function afterReading(n) {
  return function(err, data) {
    if (err) console.error(n, 'not found')
    else console.log(data.toString())
  }
}
for (var i = 1; i <= 2; i++) {
  fs.writeFileSync('data' + i + '.txt', 'Hello ' + i)
}
for (var i = 1; i <= 3; i++) {
  var name = 'data' + i + '.txt'
  var time = 100 - 10 * i
  setTimeout(function() {
    fs.readFile(name, afterReading(name))
  }, time)
}
console.log("root(" + i + ") = ", Math.sqrt(i))
```

- a Hello 1  
Hello 2  
data3.txt not found  
root(4) = 2
- b root(4) = 2**  
**data3.txt not found**  
**data3.txt not found**  
**data3.txt not found**
- c root(4) = 2  
Hello 1  
Hello 2  
data3.txt not found
- d root(4) = 2  
data3.txt not found  
Hello 2  
Hello 1

**7** *The statement that defines in the best way (among those presented in this question) what is a callback is...*

- a** ...the result of a call to a function, when that result is also a function.
- b** ...the protocol needed by a client browser for interacting with the Wikipedia servers.
- c** ...a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables or arguments.

**d** ...is a function A that is passed as a parameter to another function B, and A is run when B ends in order to process B's results.

**8** *On the declaration scope of JavaScript variables:*

- a** A variable declared with 'let' is local to the function where it has been declared.
- b** A variable declared with 'var' is local to the block where it has been declared.
- c** A variable that is implicitly declared (without 'let' or 'var') in a function, is accessible from the entire file.
- d** Scopes are scanned from the most external to the most internal.

**9** *About type management in JavaScript:*

- a** We do not need to define a local variable (in a function) before using it.
- b** When a variable is defined, we must assign an initial value to it.
- c** When a variable is defined, we must specify its type.
- d** In its lifetime, a variable may hold values of different types.

**10** *About JavaScript functions:*

- a** An anonymous function is a value that can be assigned, passed as an argument or returned as a result.
- b** Inside functions, we may declare other functions, but only when the latter are anonymous functions.
- c** A function may have multiple return values.
- d** The number of parameters in a function declaration must be equal to the number of arguments used when it is called.

**11** *About JavaScript closures:*

- a** A closure is a function that returns another function as its result.
- b** A closure is a function that remembers the scope where it has been created.
- c** It is impossible to define a closure with arguments.
- d** A closure can only access its arguments, its local variables and the global variables.

**12** *About events:*

- a** NodeJS uses an execution thread for its main code and it starts an auxiliary thread per managed event.
- b** There is an event queue that represents ended activities.
- c** We may only associate an answer to each given event.
- d** A new event cannot be managed, at least, until the management of the current one ends.

- 13** Let us assume a chat service developed in NodeJS+ZeroMQ, with two communication channels: a PUB/SUB in order to broadcast messages from the server to all clients, and a PUSH/PULL in order to forward typed-in client messages to the server.

Now, we must rewrite this service in order to use a single socket (instead of two) in each program. The best option is:

- a The client uses a REQ socket, and the server uses a REP.
- b The client uses a ROUTER socket, and the server uses a DEALER.
- c The client uses a DEALER socket, and the server uses a ROUTER.**
- d None. No solution exists with a single socket per program.

- 14** We want to implement a group membership service using NodeJS+ZeroMQ. Each process is placed in a different computer: it periodically sends a heart-beat message to all other processes, and listens to the heart-beat messages from the other participants. Every process uses the same port number and knows the IP addresses of the other processes.

In order to implement this service, choose the best option:

- a Every process defines a socket r of type ROUTER, uses r.bind(...), and for each URL of other processes uses r.connect(...)
- b Every process defines sockets p of type REP and q of type REQ, uses p.bind(...) and for each URL of other processes uses q.connect(...)
- c Every process defines a socket q of type REQ, uses q.bind(...) and for each URL of other processes uses q.connect(...)
- d Every process defines sockets p of type PUB and s of type SUB, uses p.bind(...) and for each URL of other processes uses s.connect(...)**

- 15** Let us assume these programs. Once this command line is started, \$ node client & node server 8888 &, then the following sentence is true:

```
// client.js
const zmq=require('zmq')
const rq=zmq.socket('dealer')
rq.connect('tcp://127.0.0.1:8888')
rq.connect('tcp://127.0.0.1:8889')
for (let i=1; i<=100; i++) {
  rq.send([",", i])
  console.log("Sending %d",i)
}
rq.on('message',function(del,req,rep){
  console.log("%s: %s",req,rep)
})
```

```
// server.js
const zmq = require('zmq')
const rp = zmq.socket('rep')
let port = process.argv[2]
rp.bindSync('tcp://127.0.0.1:'+port)
rp.on('message',function(msg) {
  let j = parseInt(msg)
  rp.send([msg,(j*3).toString()])
})
```

- a That server receives and correctly manages all the requests sent by that client.
- b The client gets immediately blocked since it must interact with two servers, and there is only one in this example.
- c The first few requests are lost since the client has been started before the first server was started.
- d That server receives and correctly manages 50 of the requests sent by that client.**

- 16** Let us assume these programs. Let us consider that port 8888 is initially unbound. Once this command line is started, \$ node client & node server 8888 &, then the following sentence is true:

```
// client.js
const zmq=require('zmq')
const rq=zmq.socket('dealer')
rq.bindSync('tcp://127.0.0.1:8888')
for (let i=1; i<=100; i++) {
  rq.send(["", i])
  console.log("Sending %d", i)
}
rq.on('message',function(del,req,result){
  console.log("%s: %s",req,result)
})
```

```
// server.js
const zmq = require('zmq')
const rp = zmq.socket('rep')
let port = process.argv[2]
rp.connect('tcp://127.0.0.1:'+port)
rp.on('message',function(msg) {
  let j = parseInt(msg)
  rp.send([msg,(j*3).toString()])
})
```

- a** That server receives and correctly manages all the requests sent by that client.
- b** No communication is possible in this example, since DEALER sockets may only interact with ROUTER sockets and REP sockets may only interact with REQ sockets.
- c** No communication is possible in this example, since client processes cannot use the bind() operation. They must only use connect().
- d** Replies are not correctly shown by the client, since its result parameter in the 'message' listener is always undefined.

- 17** Let us consider a connection between two DEALER sockets. Which of these sentences is correct?

- a** That communication pattern is invalid since it is not allowed by ZeroMQ.
- b** The bound DEALER can only reply to the messages sent by the other DEALER.
- c** Both DEALER sockets may send messages at any time and in any format. There is no constraint.
- d** Messages sent by both DEALER sockets must include, at least, an empty segment. There is no other constraint.

- 18** About bind/connect:

- a** If connect() precedes bind(), then the agent that has tried connect() is aborted.
- b** Except for PUB/SUB sockets, all other sockets do not accept multiple connect() onto a single bind().
- c** With PUSH/PULL, multiple connect() cannot be done onto a single bind().
- d** A single REQ socket may connect onto multiple REP sockets.

- 19** About multi-segment message structure:

- a** A ROUTER socket does not alter the structure of the sent messages.
- b** A REQ socket does not alter the structure of the sent messages.
- c** A DEALER socket modifies the structure of the received messages.
- d** A ROUTER socket modifies the structure of the received messages.

- 20** *About the incoming and outgoing message queues in ZeroMQ sockets:*
- a** A PUSH socket has one incoming queue and one outgoing queue.
  - b** A DEALER socket only has an incoming message queue.
  - c** A REQ socket has a pair of incoming and outgoing queues per agent connected to it.
  - d** A ROUTER socket has a pair of incoming and outgoing queues per agent connected to it.

# NIST

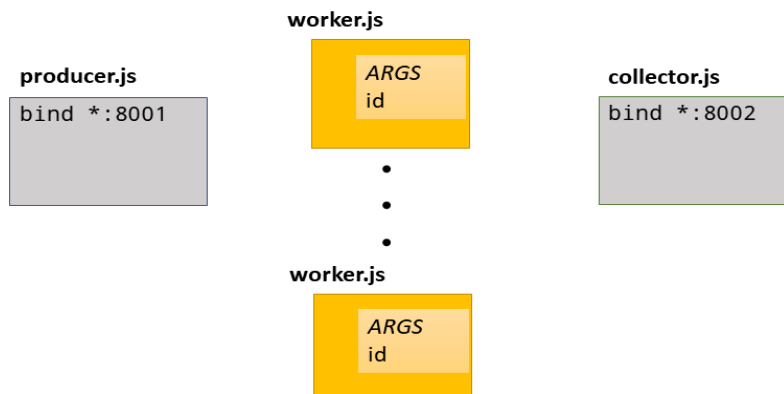
Let us consider these programs:

<pre>// producer.js const zmq = require('zmq') let psh = zmq.socket('push') psh.bind('tcp://*:8001') let i = 0 let handler =   setInterval(function() {     psh.send(''+ ++i)     if (i == 150) {       clearInterval(handler)       psh.close()       process.exit()     }   }, 100)</pre>	<pre>// collector.js const zmq = require('zmq') let pull = zmq.socket('pull') pull.bind('tcp://*:8002') pull.on('message',   function(w, n, m) {     console.log('[w' + w       + '] fact(' + n + ') = ' + m)   })  // myMath.js function fact(n) {   if (n &lt; 2) return 1   else return n * fact(n - 1) } exports.fact = fact</pre>
---	--

These programs belong to an incomplete *pipeline* (1:n:1) that should distribute the computation of the factorials of the first 150 integer numbers among *n* worker components.

**You must implement** that *worker* component in order to provide such functionality, considering that the *worker*:

- Has to receive (from the command line) a value, to be used as its identifier.
- Has to connect to the *producer* and the *collector* using the appropriate sockets.
- For each delivered message, it should forward through the pipeline a multi-segment message that includes its worker identifier, the received number and the factorial of that number.



```
const zmq = require('zmq')
const math= require('./myMath')
let sin = zmq.socket('pull')
let sou = zmq.socket('push')
const id = process.argv[2]
sin.connect('tcp://127.0.0.1:8001')
sou.connect('tcp://127.0.0.1:8002')
sin.on('message', (n) => {
  sou.send([id, n,  math.fact(ParseInt(n))])
})
```



*This part provides 1 point to your global NIST grade. It consists of 10 multiple choice questions. Each question has 4 choices and a single correct answer. Each correct answer provides 0.1 points and each error -0.033 points. Answers must be delivered in a separate specific sheet.*

The programs in this first page correspond to:

- 1 The auxiliary module auth.js
- 2 File ej.js, that uses such a module (with numbered lines)

Both files are mentioned in the questions. We recommend that you separate this first sheet in order to easily read it when you answer those questions.

```
// auth.js
const fs=require("fs")
const file_pw = './passwd.json'
const errlec='Error reading passwd file'
const errlog='Incorrect username/passwd'
function check_login(u,ok) {
    return u.user==ok.user && u.pass==ok.pass
}
function doCheckPasswd(u,f_err,f_succ) {
    fs.readFile(file_pw,"utf8",(err,data)=>{
        if(err)f_err(errlec)
        else
            if (!check_login(u,JSON.parse(data)))
                f_err(errlog)
            else f_succ()
    })
}
exports.doCheckPasswd = doCheckPasswd
```

```
1  const net = require("net")
2  const auth = require("./auth.js")
3  var c = 0
4  var arg=process.argv.slice(2)
5  var mc= arg[0]*1, A1= arg[1], A2= arg[2]
6
7  const proxy = net.createServer((soc)=>{
8      var sg = 0, s, ssC = false
9      function w(r,m) {
10         soc.write(JSON.stringify({res:r,mens:m}))
11     }
12     function procesa(msg) {
13
14         var obj = JSON.parse(msg)
15         switch (sg) {
16             case 0:
17                 if (obj.op == "login") {w("OK"); sg++;}
18                 else w("err")
19                 break
20             case 1:
21                 auth.doCheckPasswd(obj,
22                     m => {w("login err",m)},
23                     () => {sg++; w("login ok")})
24                 break
25             case 2:
26                 if (ssC) s.write(msg)
27                 else {
28                     s = new net.Socket()
29                     s.connect(A2,A1, () => {
30                         ssC = true; s.write(msg)
31                         s.on("data", (m)=>{soc.write(m)})
32                         s.on("end", () =>{soc.end()})
33                     })
34                 } // else
35             } // switch
36         } // procesa
37         if (c >= mc) soc.close() //demasiadas conex
38         c++ // cliente conectado
39         soc.on("data", procesa)
40         soc.on("end", (m)=>{c=c-1})
41     }).listen(8000)
```



- 1 Regarding the `auth.js` module, which of the following functions is invoked with some callback argument?
- a `check_login`
  - b `doCheckPasswd`**
  - c `f_err`
  - d `f_succ`
- 2 The code of `ej.js`...
- a Creates an HTTP server at port 8000
  - b Creates a TCP server at port 8000**
  - c Creates a TCP server at the port number stored in A1
  - d Creates a TCP server at the port number stored in A2
- 3 In `ej.js`, variable `sg` is used to ...
- a count the number of clients successfully connected
  - b count the number of served requests
  - c determine how the next message is to be processed**
  - d determine if the proxy has already opened a connection to the target TCP server
- 4 In `ej.js`, variable `c` is used to count the number of ...
- a messages sent by the client
  - b clients that have provided correct \*session\* start information (user, password)
  - c connected clients**
  - d clients that have been accepted after their initial message
- ~~5 In `ej.js`, a client sends an initial JSON formatted message ...~~
- ~~a that must include the user name and password~~
  - ~~b with the following information: username and password, plus ip address and port number where the request must be forwarded~~
  - ~~c login, so that it sends next the \*session\* start information: `usuario`, `clave`~~
  - ~~d `op:login`~~
- 6 In `ej.js`, function `procesa` can access variable `sg` because ...
- a `sg` is a global variable
  - b `sg` is an attribute of the global object
  - c `procesa` is defined when running the anonymous function defined on line 7, and that function declares `sg` within its scope**
  - d `sg` is passed as a parameter
- 7 Considering the code of `ej.js`, select the correct answer:
- a Kat most one client can access the functionality of the proxy
  - b The following situation is possible: no client has started a session with the proxy, and no client can start a sessions with the proxy**
  - c Only even clients can start a session
  - d Only odd clients can start a session
- 8 Consider changing line 8 in file `ej.js` like this
- `sg = 0; var s, ssC = false`
- a The resulting code is functionally equivalent to the originally provided
  - b Clients can no longer start a session with the proxy
  - c At most one client can connect to the proxy
  - d At most one client can start a session with the proxy**

- 9 Let us change the lines from `ej.js` specified within parenthesis as follows

(8) `var s, ssC = false`

(13) `var sg = 0;`

- a The resulting code is functionally equivalent to the originally provided
- b Only client `mc` can start a session with the proxy
- c No client can start a session with the proxy
- d Only the first arriving client can start a session with the proxy

- 10 Let us change the lines from `ej.js` specified within parenthesis as follows

(3) `var sg = { }, c = 0`

(8) `sg[soc] = 0; var s, ssC = false`

(15) `switch (sg[soc])`

(17) `if (obj.op=="login"){w("OK");sg[soc]++}`

(23) `() => {sg[soc]++;w("login ok");}`

- a The resulting code is functionally equivalent to the originally provided
- b The stage counter, `sg`, is shared among all clients. As a consequence, no client can start a session
- c No client goes beyond stage 1
- d Only the first client can go beyond stage 1



ID

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9

ETSINF - Tsr

First Parcial + Lab 1 - 11/07/2018

Surname .....

Name .....

Mark this way



DO NOT mark this way



DO NOT ERASE, do corrections with Typex

Theory

	a	b	c	d
15				

	a	b	c	d
1				

	a	b	c	d
2				

	a	b	c	d
3				

	a	b	c	d
4				

	a	b	c	d
5				

	a	b	c	d
6				

	a	b	c	d
7				

	a	b	c	d
8				

	a	b	c	d
9				

	a	b	c	d
10				

	a	b	c	d
11				

	a	b	c	d
12				

	a	b	c	d
13				

	a	b	c	d
14				

Lab

	a	b	c	d
1				

	a	b	c	d
2				

	a	b	c	d
3				

	a	b	c	d
4				

	a	b	c	d
5				

	a	b	c	d
6				

	a	b	c	d
7				

	a	b	c	d
8				

	a	b	c	d
9				

	a	b	c	d
10				