TSR	Primer Parcial	Temas 1,2,3	2018-11-07	A

La parte test tiene un valor de 2 puntos, y consta de 20 cuestiones. Cada cuestión plantea 4 alternativas y tiene una única respuesta correcta. Cada respuesta correcta aporta 0.1 puntos, y cada error descuenta 0.033 puntos. Debe contestar en la hoja de respuestas. La pregunta de respuesta abierta vale 1 punto, y debe responderse en los folios adicionales que se le entregan

1 *En computación en la nube, los proveedores de plataforma (PaaS) NO tienen que:*

- **a** Gestionar el despliegue y actualización de los servicios.
- **b** Establecer puntos de elasticidad.
- **c** Monitorizar parámetros relativos a la carga.
- **d** Instalar máquinas virtuales.

2 *En el desarrollo de la Wikipedia, se añadieron proxies inversos para:*

- **a** Garantizar la fiabilidad y tolerancia a los fallos en los servidores web.
- **b** Acelerar las peticiones dinámicas a los servidores de bases de datos (BD).
- **c** Servir de intermediarios (broker) entre los servidores web y los servidores de bases de datos (BD), repartiendo la carga entre las réplicas de la BD.
- **d** Ofrecer puntos de entrada a los usuarios del servicio, y balancear la carga de los servidores web.

3 *Un servidor asincrónico puede escalar mejor que un servidor multi-hilo porque …*

- **a** El asincrónico garantiza consistencia fuerte, y el multi-hilo proporciona consistencia débil
- **b** El asincrónico utiliza comunicación persistente, y el multi-hilo debe utilizar comunicación no persistente
- **c** Algunas actividades de un servidor multi-hilo pueden bloquearse cuando varias de ellas acceden al mismo objeto compartido
- **d** El servidor multi-hilo no puede utilizar comunicación asincrónica

4 *En relación con el modelo de Sistema Distribuido*

- **a** Está formado por un conjunto de agentes que necesitan cooperar, y por lo tanto dichos agentes no poseen estado independiente
- **b** Un sistema distribuido no es necesariamente un sistema en red
- **c** Cada agente es un proceso secuencial
- **d** Obliga a que todos los agentes avancen al mismo ritmo

5 *Sabiendo que en el directorio donde ejecutamos el siguiente programa no hay ficheros con la extensión txt, elige cuál es la salida en consola al ejecutar el siguiente programa::*

```
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

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 *Sabiendo que en el directorio donde ejecutamos el siguiente programa no hay ficheros con la extensión txt, elige cuál es la salida en consola al ejecutar el siguiente programa::*

```
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 *Entre las siguientes opciones, elige la frase que mejor define el concepto de callback:*

- **a** Es el resultado de invocar a una función, cuando el resultado también es una función
- **b** Es el protocolo que necesitan el navegador de un cliente para interactuar con los servidores de Wikipedia
- **c** Es una característica de JavaScript, que permite a una función anidada acceder a las variables o argumentos de la función que la contiene
- **d** Es una función A que se pasa como argumento a otra función B, de forma que A se ejecuta cuando finaliza B a fin de procesar sus resultados

8 *ANULADA Sobre el ámbito de declaración de las variables en JavaScript*

- **a** Una variable declarada con let es local a la función donde se declara
- **b** Una variable declarada con var es local al bloque donde se declara
- **c** Una variable que no se declara dentro de una función es accesible desde todo el fichero
- **d** Se busca desde el ámbito más externo al más interno

9 *Sobre el sistema de tipos de JavaScript*

- **a** No es necesario definir las variables locales a una función antes de su uso
- **b** Cuando se define una variable es obligatorio proporcionarle un valor inicial
- **c** Cuando se define una variable es obligatorio indicar su tipo
- **d** Durante su existencia una variable puede mantener valores de varios tipos

10 *Sobre las funciones en JavaScript*

- **a** Una función anónima es un valor que puede asignarse, pasarse como argumento, o devolverse como resultado
- **b** Dentro de una función podemos declarar otra, pero únicamente si esa función anidada en anónima
- **c** Las funciones pueden tener múltiples valores de retorno
- **d** El número de argumentos en la declaración de una función debe coincidir con el número de valores en la invocación de dicha función

11 *Sobre clausuras*

- **a** Una clausura es una función que devuelve otra función como resultado
- **b** Una clausura es una función que recuerda el entorno en que se ha creado
- **c** No se puede definir una clausura con argumentos
- **d** Una clausura sólo puede acceder a sus argumentos, sus variables locales, y a las variables globales

12 *Sobre eventos*

- **a** Node dispone de un hilo de ejecución para el código principal, y lanza hilos auxiliares para cada evento que gestiona
- **b** Existe una cola de eventos que representa actividades finalizadas
- **c** Sólo podemos asociar una respuesta a un determinado evento
- **d** No se procesa un nuevo evento como mínimo hasta finalizar la gestión del actual

13 *Hemos desarrollado un servicio Chat usando NodeJS+ZMQ que dispone de dos canales de comunicación: PUB/SUB para difundir los mensajes desde el servidor a todos los clientes, y PUSH/PULL para que cada cliente haga llegar al servidor los mensajes tecleados.*
Si nos piden reescribir el chat para que tanto cliente como servidor utilicen un único socket cada uno, la mejor opción es:

a El cliente utiliza un socket REQ, y el servidor utiliza un socket REP

b El cliente utiliza un socket ROUTER, y el servidor utiliza un socket DEALER

c El cliente utiliza un socket DEALER, y el servidor utiliza un socket ROUTER

d No se puede implementar el chat con un solo socket tanto en cliente como en servidor

14 *Queremos implementar un servicio de pertenencia a grupo usando NodeJS+ZMQ. Cada proceso participante está en un nodo distinto: envía de forma periódica un mensaje* `latido` *al resto de los procesos, y escucha los mensajes* `latido` *del resto de participantes.*
Todos conocen la IP de los restantes nodos, y todos usan para este servicio el mismo número de puerto.
Elige la mejor opción para implementar este servicio:

a Cada proceso define un socket r de tipo ROUTER, ejecuta r.bind(..), y para cada IP de otro proceso ejecuta r.connect(..)

b Cada proceso define sockets p de tipo REP y q de tipo REQ, ejecuta p.bind(..), y para cada IP de otro proceso ejecuta q.connect(..)

c Cada proceso define un socket q de tipo REQ, ejecuta q.bind(..), y para cada IP de otro proceso ejecuta q.connect(..)

d Cada proceso define sockets p de tipo PUB y s de tipo SUB, ejecuta p.bind(..), y para cada IP de otro proceso ejecuta s.connect(..)

15 *Supongamos los ficheros siguientes. Tras ejecutar* `$ node client & node server 8888 &` *se cumple que:*

```
// 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 El servidor recibe y gestiona de forma correcta todas las peticiones remitidas por el cliente

b El cliente se bloquea de forma inmediata, porque debe interactuar con dos servidores, y únicamente hemos arrancado uno

c Como hemos arrancado el cliente antes que el servidor, las primeras solicitudes se pierden

d El servidor recibe y gestiona de forma correcta 50 de las peticiones remitidas por el cliente

16 *Supongamos los ficheros siguientes. Asumiendo que el puerto 8888 está libre, tras ejecutar* `$ node client & node server 8888 &` *se cumple que:*

```
// 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 El servidor recibe y gestiona de forma correcta todas las peticiones remitidas por el cliente

b No se pueden comunicar entre sí, porque el socket tipo DEALER únicamente puede interactuar con sockets tipo ROUTER, y un socket tipo REP únicamente puede interactuar con sockets tipo REQ

c No se pueden comunicar entre sí, porque un cliente no puede utilizar bind (debe utilizar siempre connect)

d El cliente no muestra correctamente las respuestas, porque el parámetro `result` en el callback de `rq.on('message')` siempre es `undefined`

17 *Considérese una conexión entre dos sockets de tipo dealer. ¿Cuál de los siguientes enunciados es válido?*

a Este patrón operativo es inválido ya que no está permitido en zmq.

b El dealer que hace el bind sólo puede responder a los mensajes que reciba del otro dealer.

c Ambos dealers puede enviar mensajes tal y como lo precisen, en cualquier momento y formato.

d Los mensajes enviados por ambos dealers deben necesariamente incluir, al menos, un segmento vacío y no están sujetos a ningún otro requisito.

18 *Sobre bind/connect*

a Si se intenta connect antes que bind, el agente que ha realizado connect aborta

b Excepto con los sockets PUB/SUB, en el resto de casos no podemos realizar varios connects sobre un único bind

c Con PUSH/PULL no podemos realizar varios connects sobre un único bind

d Un único socket tipo REQ puede hacer connect sobre más de un socket de tipo REP

19 *Sobre la estructura de los mensajes (mensajes multi-segmento)*

a Un mensaje enviado por un socket tipo ROUTER no modifica la estructura del mensaje enviado

b Un mensaje enviado por un socket tipo REQ no modifica la estructura del mensaje enviado

c Un socket tipo DEALER modifica la estructura del mensaje recibido

d Un socket tipo ROUTER modifica la estructura del mensaje recibido

20 *Sobre las colas de Entrada/Salida de los sockets*

a Un socket de tipo PUSH posee una cola de entrada y una cola de salida

b Un socket de tipo DEALER únicamente posee una cola de entrada

c Un socket de tipo REQ posee una cola de entrada y salida por agente conectado al mismo

d Un socket de tipo ROUTER posee una cola de entrada y salida por agente conectado al mismo

TSR

Considérense estos programas:

```
// 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)
```

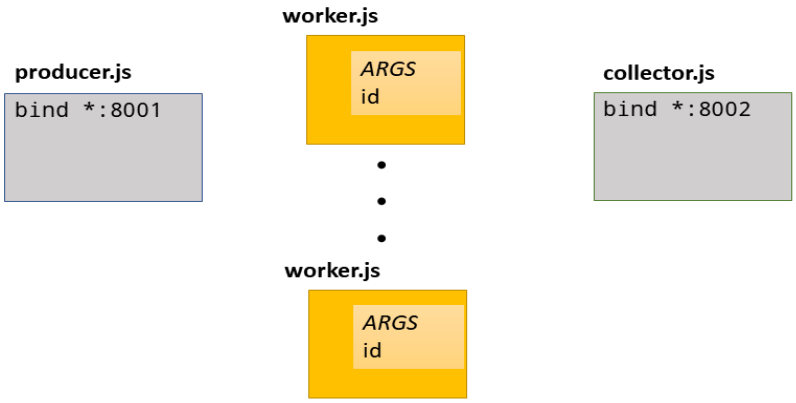
```
// collector.js
const zmq = require('zmq')
let pull = zmq.socket('pull')
pull.bind('tcp://*:8002')
pull.on('message',
  function(w, n, m) {
    console.log('[w' + w
      + '] fact(' + n +') = ' + m)
})
```

```
// myMath.js
function fact(n) {
  if (n < 2) return 1
  else return n * fact(n - 1)
}
exports.fact = fact
```

Estos programas forma parte de un *pipeline* (1:*n*:1), incompleto, mediante el cual se quiere distribuir el cómputo de los factoriales de los primeros 150 números enteros entre ***n*** componentes trabajadores (*workers*).

Se pide implementar el componente ***worker*** para proporcionar dicha funcionalidad, teniendo en cuenta que el ***worker***:

- Debe recibir (por línea de comandos) un valor, a usar como su identificador.
- Debe conectarse a los componentes ***producer*** y ***collector*** mediante sockets adecuados.
- En respuesta a cada petición, debe enviar un mensaje multisegmento que incluya su identificador, el número recibido (en la petición), y el factorial de dicho número.



Solución (worker.js)

```
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))])
})
```

Esta prueba tiene un valor de 1 punto, y consta de 10 cuestiones tipo test. Cada cuestión plantea 4 alternativas y tiene una única respuesta correcta. Cada respuesta correcta aporta 0.1 puntos, y cada error descuenta 0.033 puntos. Debe contestar en la hoja de respuestas.

Los códigos de esta primera página corresponden a:

1 El módulo auxiliar auth.js

2 El fichero ej.js, que utiliza dicho módulo (fichero con las líneas numeradas)

Ambos ficheros se referencian posteriormente desde las cuestiones. Se recomienda desgrapar esta página para mantenerla como referencia al contestar las cuestiones.

```
// 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 *En relación con el módulo* `auth.js` *¿Cuál de las siguientes funciones se invoca pasándole como argumento alguna función callback?*

- **a** check_login
- **b** doCheckPasswd
- **c** f_err
- **d** f_succ

2 *El código de* `ej.js`

- **a** Crea un servidor HTTP en el puerto 8000
- **b** Crea un servidor TCP en el puerto 8000
- **c** Crea un servidor TCP en el puerto pasado como parámetro en A1
- **d** Crea un servidor TCP en el puerto pasado como parámetro en A2

3 *En el código de* `ej.js`, *la variable* `sg` …

- **a** Se utiliza para contabilizar los clientes conectados con éxito
- **b** Se utiliza para contar el número de peticiones servidas
- **c** Se utiliza para determinar qué procesamiento se realizará con el siguiente mensaje
- **d** Determina si una conexión con un servidor TCP está abierta o no

4 *En el código de* `ej.js`, *la variable* `c` …

- **a** Se utiliza para contabilizar el número de mensajes que envía el cliente
- **b** Se utiliza para contabilizar el número de clientes que han proporcionado una información de inicio de *sesión* (usuario, clave) correcta
- **c** Se utiliza para conocer el número de clientes conectados
- **d** Se utiliza para conocer el número de clientes aceptados después de su mensaje inicial

5 *En el código de* `ej.js`, *un cliente envía un mensaje inicial en JSON…*

- **a** Que debe incluir el nombre del usuario y su clave
- **b** Con la siguiente información: el usuario y la clave, así como la dirección ip y el puerto de la máquina hacia donde desea redirigir su petición
- **c** `login` para a continuación enviar la información de inicio de sesión `usuario, clave`
- **d** `{"op":"login"}`

6 *En el código de* `ej.js`, *la función procesa puede acceder a la variable* `sg` …

- **a** Porque `sg` es una variable global
- **b** Porque `sg` es un atributo del objeto global
- **c** Porque `procesa` se define al ejecutar la función anónima de la línea 7, y dicha función declara a `sg` dentro de su ámbito
- **d** Porque `sg` se le pasa como parámetro

7 *Analizando el código de* `ej.js`, *selecciona la respuesta verdadera:*

- **a** Sólo un cliente puede acceder a la funcionalidad del proxy
- **b** Es posible que ningún cliente haya iniciado sesión con el proxy, y que ningún otro cliente pueda iniciar sesión con el proxy
- **c** Sólo los clientes pares pueden iniciar sesión
- **d** Sólo los clientes impares pueden iniciar sesión

8 *Considerando el siguiente cambio en la línea 8 del fichero* `ej.js`

sg = 0; var s, ssC = false

- **a** El código resultante es equivalente al original
- **b** Ningún cliente podrá iniciar sesión con el proxy
- **c** Como mucho un cliente podrá estar conectado al proxy
- **d** Tan sólo un cliente podrá iniciar sesión con el proxy

9 *Considerando los siguientes cambios en las líneas correspondientes del fichero* `ej.js`

```
(8)  var s, ssC = false
(13) var sg = 0;
```

a El código resultante es equivalente al original

b Sólo el cliente `mc` puede establecer una sesión con el proxy

c Ningún cliente puede establecer una sesión con el proxy

d Sólo el primer cliente puede establecer una sesión con el proxy

10 *Considerando los siguientes cambios en las líneas correspondientes del fichero* `ej.js`

```
(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 El código resultante es equivalente al original

b El contador de etapa *sg* se comparte entre todos los clientes, por lo que ningún cliente puede establecer una sesión

c Ningún cliente puede progresar más allá de la etapa 1

d Sólo el primer cliente puede progresar más allá de la etapa 1