| TSR | Recuperación (Primer parcial) | Test de teoría | 2019-01-25 | A |
|---|---|---|---|---|

*Esta parte contiene 12 preguntas de teoría. Cada cuestión tiene cuatro opciones y solo una de ellas es correcta. Cada respuesta correcta aporta 0.167 puntos y cada error descuenta 0.067 puntos. Las respuestas deben darse en la hoja final.*

**1** *En el paradigma de programación asincrónica, esta afirmación es cierta:*

- **a** Cada proceso ejecuta explícitamente múltiples hilos de ejecución.
- **b** En cada proceso, varios eventos son gestionados sin necesitar mecanismos de sincronización.
- **c** Los procesos no pueden compartir ningún recurso.
- **d** Los procesos nunca fallan.

**2** *En ECMAScript 6, la diferencia principal entre* `let` *y* `var` *es:*

- **a** Ninguna. Ambas instrucciones tienen la misma funcionalidad.
- **b** Su ámbito de declaración de variables (`let`=bloque, `var`=función).
- **c** Su ámbito de declaración de variables (`let`=global, `var`=función).
- **d** `let` define constantes y `var` define variables.

**3** *¿Cuál es la salida proporcionada por este código?*

```
function something(x) {
return function(y) {return x*y}
}
let another = something(3)
console.log(another(20))
```

- **a** Un error, pues `another` no admite argumentos.
- **b** Un error, pues `another` necesita dos argumentos.
- **c** 60.
- **d** 20.

**4** *¿Cuál de estas afirmaciones ES FALSA?*

```
function something(x) {
return function(y) {return x*y}
}
let another = something(3)
console.log(another(20,10))
```

- **a** La variable `another` es una función.
- **b** En la última línea, se está usando `another` como un callback.
- **c** La segunda línea devuelve una función anónima.
- **d** La función `something` proporciona una clausura a su función interna.

**5** *Si consideramos este programa, ¿cuál de estas afirmaciones sobre gestión de tipos y parámetros ES FALSA?*

```
function something(x) {
return function(y) {return x*y}
}
let another = something(3)
console.log(another(0))
```

**a** Para obtener resultados válidos, el parámetro y debe recibir valores de tipo numérico.

**b** En la penúltima línea, no hay diferencias si sustituimos `something(3)` con `something(3,5,7)`.

**c** La instrucción `let` en la penúltima línea no es obligatoria; podemos eliminarla.

**d** La última línea genera un error.

**6** *¿Qué afirmación sobre este programa ES FALSA?*

```
const fs=require('fs')
const filename=process.argv[2]
fs.readFile(filename, (err,data) => {
  if (err) {
    console.log('Error in read operation')
  } else {
    console.log('Contents of %s:', filename)
    console.log(data+'')
  }
})
console.log('File %s has been read!',filename)
```

**a** Se usa un callback.

**b** Si se ejecuta, la línea `Contents of …` no muestra el nombre de fichero correcto.

**c** `File … has been read!` no es la última línea mostrada.

**d** El proceso aborta si no recibe ningún argumento.

**7** *¿Cuál es la salida de este programa?*

```
function writing(x) {
  console.log("Writing after " + x + " seconds")
  return ()=>0
}
setTimeout(writing(6), 6000)
setTimeout(writing, 3000)
console.log("Program completed!")
```

**a** Writing after 6 seconds
Program completed!
Writing after undefined seconds

**b** Program completed!
Writing after 6 seconds
Writing after 3 seconds

**c** Program completed!
Writing after 6 seconds
Writing after undefined seconds

**d** Program completed!
Writing after 3 seconds
Writing after 6 seconds

**8** *Sea un mensaje ZeroMQ multi-segmento con estos tres segmentos:* `'the'`,`'3'`,`'segments'`. *Entonces:*

**a** Será entregado utilizando tres eventos `'message';` uno por segmento.

**b** Será entregado utilizando un único evento `'message'`.

**c** Puede enviarse usando un socket `so` con esta instrucción: `so.send('the','3','segments')`

**d** No puede gestionarse en NodeJS.

**9** *¿Qué ocurre si utilizamos* `so.connect('tcp://*:8000')` *en un socket ZeroMQ* `so` *?*

**a** El proceso genera una excepción en esa línea.

**b** El socket `so` se conecta al puerto 8000 en todas las direcciones locales.

**c** El socket `so` se conecta al puerto 8000 en la dirección local de loopback (es decir, en la 127.0.0.1).

**d** El socket `so` hace `bind` en el puerto 8000 de todas las direcciones locales.

**10** *¿Cuál es la salida de esta línea de órdenes?: node Client & node Server 8888 & node Server 8889 &*

```
// Client.js
const zmq = require('zmq')
let rq = zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.connect('tcp://127.0.0.1:8889')
let counter=0
function sendMsg() {
  rq.send(['Hello',++counter])
}
setInterval(sendMsg,1000)
rq.on('message', function(msg) {
  console.log('Response: ' + msg)
})
```

```
// Server
const zmq = require('zmq')
let rp = zmq.socket('rep')
rp.bindSync('tcp://127.0.0.1:'+process.argv[2])
rp.on('message', function(msg,count) {
  console.log('Request: %s %d', msg, count)
  if (count==3) process.exit(1)
  rp.send(count)
})
```

**a** Ninguna. Los tres procesos no llegan a intercambiar ningún mensaje.

**b** Una salida infinita donde todas las peticiones enviadas por el cliente obtienen respuesta.

**c** Todos los mensajes pares obtienen respuesta, pero solo el primer mensaje impar será contestado.

**d** Solo se muestran las tres primeras peticiones con las dos primeras respuestas.

**11** *¿Cuál es la salida de esta línea de órdenes?: node Client & node Server 8888 & node Server 8889 &*

```
// Client.js
const zmq = require('zmq')
// Now, 'rq' is a DEALER instead of a REQ
let rq = zmq.socket('dealer')
rq.connect('tcp://127.0.0.1:8888')
rq.connect('tcp://127.0.0.1:8889')
let counter=0
function sendMsg() {
  rq.send(['Hello',++counter])
}
setInterval(sendMsg,1000)
rq.on('message', function(msg) {
  console.log('Response: ' + msg)
})
```

```
// Server
const zmq = require('zmq')
let rp = zmq.socket('rep')
rp.bindSync('tcp://127.0.0.1:'+process.argv[2])
rp.on('message', function(msg,count) {
  console.log('Request: %s %d', msg, count)
  if (count==3) process.exit(1)
  rp.send(count)
})
```

**a** Ninguna. Los tres procesos no llegan a intercambiar ningún mensaje.

**b** Una salida infinita donde todas las peticiones enviadas por el cliente obtienen respuesta.

**c** Todos los mensajes pares obtienen respuesta, pero solo el primer mensaje impar será contestado.

**d** Solo se muestran las tres primeras peticiones con las dos primeras respuestas.

**12** *Queremos implantar un servicio en el que un componente A debe enviar asíncronamente mensajes a otro componente B utilizando un canal unidireccional. Esto significa que no se necesitan enviar mensajes de B hacia A. Para implantar este canal utilizando ZeroMQ, ¿cuál de estas alternativas NO PUEDE UTILIZARSE?*

**a** Un socket `PUSH` en A y otro `PULL` en B.

**b** Un socket `PUB` en A y otro `SUB` en B.

**c** Un socket `REQ` en A y otro `REP` en B.

**d** Un socket `DEALER` en A y otro `ROUTER` en B.