Este examen consta de 26 cuestiones, con una puntuación total de 10 puntos. Cada cuestión posee 4 alternativas, de las cuales únicamente una es cierta. La nota se calcula de la siguiente forma: tras descartar las dos peores cuestiones, cada acierto suma 10/24 puntos, y cada error descuenta 10/72 puntos. Debes contestar en la hoja de respuestas.

1 *Consideremos el siguiente par de programas Node.js que usan ØMQ:*

```
// Program: client.js
const zmq = require('zeromq')
const rq = zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.send('Hello')
rq.on('message', function(msg) {
  console.log('Response: ' + msg)
})
```

```
// Program: server.js
const zmq = require('zeromq')
const rp = zmq.socket('rep')
rp.bind('tcp://127.0.0.1:8888',
    function(err) {
      if (err) throw err
    })
rp.on('message', function(msg) {
    console.log('Request: ' + msg)
    rp.send('World')
})
```

Todas las instancias de esos programas se ejecutan en el mismo ordenador. Elija la opción verdadera:

a Todas las demás opciones son verdaderas.

b No habrá ningún error si iniciamos y ejecutamos simultáneamente dos procesos servidores.

c Si iniciamos un proceso servidor, será posible iniciar varios procesos clientes. Esos clientes eventualmente recibirán una respuesta.

d Un servidor puede recibir y entregar un mensaje enviado por un cliente antes de enviar la respuesta a una solicitud enviada y entregada previamente desde otro cliente.

2 *El patrón de comunicación ØMQ REQ-REP se considera sincrónico porque*

a Un servidor no puede manejar conexiones simultáneas con más de un cliente.

b La recepción de mensajes de solicitud y respuesta no se puede manejar utilizando listeners en los procesos servidor y cliente, respectivamente.

c Un socket REQ no puede enviar ni transmitir dos mensajes de solicitud consecutivos si no recibe un mensaje de respuesta entre esos envíos.

d Ese patrón no puede proporcionar persistencia de comunicación.

3 *¿Cuál de estas oraciones es cierta sobre los sistemas LAMP?*

a Los sistemas LAMP están diseñados para una alta escalabilidad.

b Todo servicio SaaS debe ser un sistema LAMP.

c Algunos sistemas LAMP utilizan Windows 11 como sistema operativo.

d Los sistemas LAMP utilizan, entre otros elementos, servidores web Apache.

4 *Para mejorar la escalabilidad de su servicio de base de datos interno, Wikipedia:*

a Usa una política de administración que impide a los usuarios actualizar el contenido de la base de datos.

b Usa un sistema de gestión de bases de datos NoSQL implementado en JavaScript.

c Utiliza un modelo de replicación pasiva en el que las réplicas secundarias pueden atender solicitudes de solo lectura.

d No utiliza replicación, así evita condiciones de carrera e inconsistencias.

5 *Consideremos el siguiente par de programas Node.js que usan ØMQ:*

```
// Program: publisher.js
const zmq = require("zeromq")
const pub = zmq.socket('pub')
let count = 0
pub.bindSync("tcp://*:5555")
setInterval(function() {
  pub.send("TEST " + count++)
}, 1000)
```

```
// Program: subscriber.js
const zmq = require("zeromq")
const sub = zmq.socket('sub')
sub.connect("tcp://localhost:5555")
sub.subscribe("TEST")
sub.on("message", function(msg) {
  console.log("Received: " + msg)
})
```

Todas las instancias de esos programas se ejecutan en el mismo ordenador. Elija la opción verdadera:

a Podemos iniciar, en cualquier orden, varios suscriptores y un solo emisor. El conjunto de procesos resultante no generará ningún error y todos los mensajes enviados se entregarán a todos los suscriptores eventualmente.

b Solo podemos iniciar un único emisor en cada ejecución de este conjunto de programas.

c Todas las demás opciones son verdaderas.

d Si eliminamos en el programa suscriptor la llamada sub.subscribe, no observaremos ningún cambio en el comportamiento de los procesos resultantes.

6 *Supongamos que se utilizará una URL A (por ejemplo, A = tcp://158.42.1.118:30000) para interconectar múltiples procesos en ØMQ. La siguiente oración es verdadera:*

a Distintos procesos pueden hacer varias llamadas simultáneas y correctas a bindSync(A).

b Todas las llamadas a connect(A) deben preceder a la primera llamada a bind(A).

c La llamada a bind(A) debe preceder a todas las llamadas a connect(A).

d No hay problema si un proceso llama a connect(A) y luego otro proceso hace la primera llamada a bind(A).

7 *Consideremos el siguiente par de programas Node.js que usan ØMQ:*

```
// Program: sender.js
const zmq = require("zeromq")
const producer = zmq.socket("push")
let count = 0
producer.bind("tcp://*:8888", (err) => {
   if (err) throw err
   setInterval( () => {
        producer.send("msg# " + count++)
    }, 1000)
})
```

```
// Program: receiver.js
const zmq = require("zeromq")
const consumer = zmq.socket("pull")
consumer.connect("tcp://127.0.0.1:8888")
consumer.on("message", function(msg) {
   console.log("received: " + msg)
})
```

Todas las instancias de esos programas se ejecutan en el mismo ordenador. Elija la opción verdadera:

a Podemos iniciar, en cualquier orden, varios receptores y un solo emisor. El conjunto de procesos resultante no generará ningún error y los mensajes eventualmente se entregarán.

b En el programa receiver.js podemos agregar, como última línea de su listener para message, una instrucción consumer.send(msg). Responderá al emisor.

c Todas las demás opciones son verdaderas.

d Si se inicia un único receptor en cada ejecución de este par de programas, el argumento para su llamada a consumer.connect() puede ser tcp://*:8888.

8 *En el área de aplicación de la informática cooperativa:*

a Los fallos de los clientes se pueden superar fácilmente: ya sea redistribuyendo sus tareas o reenviando cada tarea a varios clientes.

b Los nodos clientes realizan las tareas de cómputo.

c Los nodos servidores realizan tareas de distribución de datos.

d Todas las demás opciones son verdaderas.

Consideremos estos programas JavaScript:

```
// Program: ex1.js
function f(x) {
   return (y) => { x++; return x+y }
}
```

```
// Program: ex2.js
function f(x) {
   return (y) => { x++; return x+y }
}
g=f(0)
console.log(g(2))
console.log(g(3))
```

```
// Program: ex3.js
function f(x) {
   return (y) => { x++; return x+y }
}
g=f(0)
h=f(0)
console.log(g(2))
console.log(h(2))
```

```
// Program: ex4.js
const ev = require('events')
const emitter = new ev.EventEmitter()
const e1 = "print"
emitter.on(e1, (y) => {console.log( y+"!!") })
setTimeout(() => {emitter.emit(e1, "First")}, 2000)
emitter.emit(e1,"Second")
console.log("End.")
```

9 *¿Cuál es el resultado de una ejecución de ex4.js?*

- **a** End.
  Second!!
  First!!
- **b** First!!
  Second!!
  End.
- **c** End.
  First!!
  Second!!
- **d** Second!!
  End.
  First!!

10 *Añadimos en ex1.js una llamada f(5) tras su última línea. ¿Cuál es el resultado de esa llamada?*

- **a** Un error.
- **b** Una función.
- **c** Valor 6.
- **d** Esta cadena: 5+y.

11 *¿Qué vemos en pantalla al ejecutar ex2.js?*

- **a** 3
  5
- **b** Un error.
- **c** 1y
  2y
- **d** 2
  3

12 *¿Qué vemos en pantalla al ejecutar ex3.js?*

- **a** 3
  3
- **b** Un error
- **c** 1y
  1y
- **d** 3
  4

13 *¿Cuál es el ámbito de x en el programa ex2.js?*

- **a** Ninguno: su primer acceso aborta el proceso.
- **b** Global
- **c** Local para f e inaccesible para la función devuelta por f.
- **d** Local para f y accesible en la clausura de g.

14 *¿Cuántas veces se pasa una función como argumento en ex4.js?*

- **a** Ninguna.
- **b** Una.
- **c** Dos.
- **d** Tres.

15 *Esta es una variación breve del programa emitter2.js utilizado en la práctica 1:*

```
const ev = require ( 'events' )
const emitter = new ev . EventEmitter ()
function handler ( event , n ) {
   return (incr)=>{
      n+=incr
      console.log(event + ': ' + n)
    }
}
emitter.on('e1', handler('e1','prefix'))
for (let i=1; i<3; i++) emitter.emit('e1',i)
```

¿Cuál es el resultado en pantalla de una ejecución de ese programa?

a e1: prefixi
e1: prefixii

b e1: prefix1
e1: prefix12

c prefix: 1
prefix: 3

d event: prefixincr
event: prefixincrincr

16 *Consideremos este programa JavaScript:*

```
const fs=require("fs")
console.log("Call to first readFile")
fs.readFile("/proc/loadavg",(e,d)=> {
   if (e) console.error(e.message)
   else console.log(d+'')
   console.log("End of first readFile\n")
})
console.log("Call to second readFile")
console.log( fs.readFileSync("/proc/loadavg") + "" )
console.log("End of second readFile\n")
```

¿Cuál es la última línea de texto que muestra ese programa?

a Puede cambiar de una ejecución a otra.

b La última línea en /proc/loadavg

c End of first readFile

d End of second readFile

17 *El término comunicación no persistente, en el área de los sistemas de mensajería, significa:*

a Los agentes receptores bloquean su operación de recepción cuando no hay mensajes en el canal de comunicación.

b Comunicación no fiable, es decir, se pueden perder mensajes.

c El remitente y el receptor no se bloquean en la gestión de comunicaciones.

d Los canales de comunicación no tienen capacidad y esto obliga a ambos agentes, emisor y receptor, a estar listos y conectados antes de iniciar su intercambio de mensajes.

18 *¿Qué se muestra cuando se ejecuta este programa?*

```
function f1 (a,b,c) {
  console.log (arguments.length + " arguments")
  return a+b+c;
}
console.log( "result: " + f1 ('3'))
```

a 1 arguments
result: 3undefinedundefined

b 3 arguments
result: 3undefinedundefined

c 3 arguments
result: 3

d 1 arguments
result: 3

19 *Consideremos este programa JavaScript:*

```
"use strict"
for (var i=0; i<5; i++) {
  console.log ("i: " + i)
}
console.log ("end --> i=" + i)
```

Si eliminamos la palabra clave var en su segunda línea, ¿cuáles son los cambios, si los hay, en la ejecución del programa resultante?

a Ninguna de las demás opciones es correcta.

b El programa se ejecuta de la misma manera.

c El programa aborta en su segunda línea.

d El programa aborta en su última línea.

20 *En la documentación de ØMQ, el término mensaje segmentado (o mensaje multiparte) se refiere a un tipo concreto de estructura de mensaje que exige una gestión de envío y recepción diferente a la de los mensajes no segmentados. Supongamos que 'so' es un socket. Seleccione la opción que proporciona un ejemplo de envío de un mensaje segmentado en ØMQ:*

a
```
so.on("message", (s1,s2)=>console.log(s1+s2))
```
b
```
so.send(["seg1","seg2"])
```
c
```
so.send("seg1","seg2","seg3")
```
d
```
so.send(JSON.stringify({seg1:valor1,seg2:valor2}))
```

21 *La última tarea de la sesión 3 de la práctica 1 consiste en extender el proxy configurable para construir un proxy programable. Ese proxy programable recibe el puerto y la dirección IP iniciales del servidor remoto desde la línea de órdenes, pero también puede aceptar nuevos valores de un proceso programador.*
Para construir un proxy programable, necesitamos aplicar estos cambios (entre otros) al programa del proxy configurable:

a Crear un socket adicional con net.createServer() que escuche las conexiones del proceso programador y reenvíe todos los mensajes recibidos al servidor remoto.

b Acceder y procesar de forma adecuada dos argumentos adicionales desde la línea de órdenes.

c Crear un socket cliente TCP adicional, conectarlo al servidor remoto y enviar a través de este nuevo socket toda la información recibida de los procesos clientes.

d Crear otro socket servidor que escuche las conexiones del cliente programador, actualizando los valores REMOTE_PORT y REMOTE_IP con la información recibida.

22 *ØMQ es un ejemplo de este tipo de sistema de mensajería:*

a Sin broker y con comunicación no persistente.

b Débilmente persistente y sin broker.

c Basado en broker y fuertemente persistente.

d Basado en broker con comunicación no persistente.

23 *Consideremos el programa proxy básico utilizado en la sesión 3 del Laboratorio 1:*

```
1  const net = require('net')
2  const PORT_A = 8000
3  const IP_A = '127.0.0.1'
4  const PORT_B = 80
5  const IP_B = '158.42.4.23' //www.upv.es
6  const server = net.createServer(socket=> {
7      const ss = new net.Socket()
8      ss.connect(parseInt(PORT_B),
9          IP_B, () => {
10             socket.on('data', msg => {
11                 ss.write(msg)
12             })
13             ss.on('data', data => {
14                 socket.write(data)
15             })
16     })
17 })
18 server.listen( PORT_A , IP_A )
19 console.log("accept conn on: "+PORT_A)
```

Deberíamos extenderlo, recibiendo los valores de la dirección remota y del puerto remoto como argumentos de la línea de órdenes, para construir un proxy configurable. ¿Cuáles son los cambios de programa necesarios para este fin?

a Las líneas 4 y 5 deben cambiarse a:
```
const PORT_B=parseInt(process.argv[3]+"")
const IP_B = process.argv[2]+""
```

b No es posible realizar una modificación breve. Necesitaríamos añadir nuevas líneas de código.

c Las líneas 2 y 3 deben cambiarse a:
```
const PORT_A=parseInt(process.argv[3]+"")
const IP_A = process.argv[2]+""
```

d La línea 18 debe cambiarse a:
```
server.listen( PORT_B, IP_B)
```

24 *En la computación en la nube, el objetivo principal del modelo de servicio IaaS es:*

a Proporcionar una infraestructura adecuada para desplegar aplicaciones distribuidas.

b Automatizar el escalado del servicio.

c Automatizar la actualización del servicio.

d Proporcionar servicios software específicos a sus clientes.

25 *La programación asincrónica (PA) presenta esta ventaja en comparación con la programación concurrente (es decir, multihilo):*

a Todas las demás opciones son falsas.

b Los recursos se utilizan de forma secuencial. Eso evita condiciones de carrera.

c Cuando se considera un solo proceso, PA siempre ejecuta en paralelo todas sus tareas.

d PA no está orientada a eventos; por tanto, los procesos PA no admiten eventos externos.

26 *Queremos mostrar el contenido del archivo Example.txt (está en el directorio actual, con permiso de lectura) en pantalla usando promesas. Una posible implementación en Node.js es:*

a

```
const fs=require('fs').promises
fs.readFile("Example.txt",'utf8').catch(console.log)
```

b Ninguna, ya que esa funcionalidad no se puede implementar mediante promesas.

c

```
const fs=require('fs')
console.log(fs.readFileSync("Example.txt",'utf8'))
```

d

```
const fs=require('fs').promises
fs.readFile("Example.txt",'utf8').then(console.log)
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

D

DNI NIE PASAPORTE

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

Primer Parcial - 02/11/2023

Apellidos .....

Nombre .....

Marque así Así NO marque

NO BORRAR, corregir con corrector

Primer Parcial

	a	b	c	d
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				