Esta prueba tiene un valor de 4 puntos, y consta de 32 cuestiones tipo test. Cada cuestión plantea 4 alternativas y tiene una única respuesta correcta. Cada respuesta correcta aporta 1/8 puntos, y cada error descuenta 1/24 puntos. Debe contestar en la hoja de respuestas.

Los siguientes listados (client1.js y server1.js) se referencian por su nombre desde algunas cuestiones de prácticas

```
1  // File: server1.js
2     const net=require('net')
3     const fs=require('fs')
4
5     function getLoad() {
6     // Assume that it retuns the current workload
7     // level. Do not care about its implementation.
8       let data=fs.readFileSync('/proc/loadavg')
9       return parseFloat((data+'').split(' ')[0])*100
10    }
11    let counter=0
12    const server=net.createServer(
13      (c) => {
14        counter++
15        c.on('end', () => {counter--=1})
16        c.on('data', (m) => {
17          if (m=='reqLoad')
18            c.write(JSON.stringify(
19              {type :"load",
20               value:getLoad()}))
21          else if (m=='reqCount')
22            c.write(JSON.stringify(
23              {type:"count", value:counter}))
24          else c.write(JSON.stringify(
25            {type :"error",
26             value:"Unknown request"}))
27        })
28    })
29    server.listen(8000,
30      () => {console.log('server bound')})
```

```
1  // File: client1.js
2     const net=require('net')
3     let counter=0
4     function stage() {
5       counter++
6       client.write('reqCount')
7       setTimeout(stage,2000)
8     }
9     const client=net.connect(8000,
10      () => stage())
11
12    client.on('data', (msg) => {
13      let m = JSON.parse(msg)
14      console.log("Answer from server: ",m.value)
15    })
16    client.on('end', () => {
17      console.log("Client is ending now!")
18      process.exit(0)
19    })
```

1 *¿Cuál de las siguientes* **no** *es una característica de la programación asincrónica?*

a Las actividades en los procesos nunca o casi nunca se bloquean.

b Las condiciones de carrera rara vez podrán ocurrir.

c Todas las estructuras de datos compartidas deben protegerse con monitores.

d Se minimizan las sobrecargas (p.ej., sincronización entre actividades, cambios de contexto internos, gestión de los atributos de los procesos…) en el procesamiento.

2 *Los roles de desarrollador,proveedor de servicio y administrador de sistema pueden ser desempeñados por una misma empresa en este modelo de servicio en la nube:*

a SaaS.

b IaaS.

c PaaS.

d Ninguno.

3 *Los proxies inversos no pueden:*

a Proporcionar transparencia de ubicación.

b Ser equilibradores de carga.

c Automatizar el despliegue de los servicios.

d Actuar como cachés intermedias.

4 *Para gestionar de manera escalable el almacenamiento secundario en Wikipedia,se utilizan las siguientes tecnologías o mecanismos:*

a Blockchain.

b Reparto y distribución de sus bases de datos entre múltiples ordenadores.

c ZeroMQ.

d NodeJS.

5 *Dado el siguiente código, su ejecución da lugar a (cada valor en una línea distinta):*

```
function g(){console.log(i)}
i=0
g()
for(let i=0;i<=2;i++) setTimeout(g,i*1000)
i=0
console.log(i)
```

a 0 0 0 1 2

b 0 0 1 2 0

c 0 0 0 0 0

d un error

6 *En NodeJS:*

a Se permite el uso de módulos, callbacks y listeners.

b No se permite el uso de operaciones síncronas.

c Los programas pueden compilarse para mejorar el rendimiento.

d Se permite utilizar tanto JavaScript como java.

7 *Indique cuál de las siguientes afirmaciones sobre JavaScript es cierta:*

a Se trata de un lenguaje orientado a objetos que se debe compilar para poder ejecutarse.

b Se trata de un lenguaje propietario. En particular el lenguaje pertenece a la compañía Oracle.

c Se trata de un lenguaje interpretado con características funcionales.

d Se trata de un lenguaje más eficiente que el lenguaje C, pues a diferencia del lenguaje C, JavaScript puede ejecutarse en la mayoría de Navegadores Web.

8 *Dado el siguiente programa, indique qué afirmación es cierta.*

```
function test (x) {
  console.log (x)
}
setTimeout (test, 1000)
setTimeout (test (2), 2000)
setTimeout ( () => test (3), 3000)
```

a El programa imprimirá las líneas `2, undefined` y `3` en este orden.

b El programa imprimirá las líneas `undefined, 2` y `3` en este orden.

c El programa imprimirá las líneas `undefined` y `3`

d El programa escribe 2 y luego aborta con una excepción.

9 *Dado el siguiente programa, indique qué afirmación es cierta:*

```
function test (a,b,c) {
  console.log (a + b  + c)
}
test (1, "Hola")
test ((()=>{return 1})(), 2, 3)
```

a La primera llamada a test, no funcionará debido a que sólo se pasan 2 argumentos.

b La primera llamada a test, no funcionará debido a que se intenta sumar 1 con la cadena Hola.

c La segunda llamada a test imprimirá 6

d La segunda llamada a test imprimirá 123

10 *La salida del programa siguiente es:*

```
function g(x) {
  y = (a) => { return a*x }
}
g(3)
console.log(y(4))
```

a `undefined`.

b 12

c Un error.

d 4

11 *La salida del programa siguiente es:*

```
function g(x) {
  y = (a) => { return a*x }
  return y
}
console.log(g(3)(5))
```

a Un error.

b `undefined`.

c 15

d 3

12 *Necesitamos averiguar cuántas veces un carácter concreto está contenido en cierto fichero. Para ello, hemos escrito una función* `count(c,a)` *que muestra cuántas veces el carácter* `c` *está contenido en el vector o cadena* `a`. *Asumamos que una línea inicial en nuestro programa es:* `const fs=require('fs')`. *¿Cómo puede usarse* `count` *para averiguar cuántos caracteres* `'A'` *hay en el fichero* `'f.txt'`*?*

a
```
fs.readFile('f.txt', count)
```

b
```
fs.readFile('f.txt', (e,d) => {
  if (!e) count('A',d) })
```

c
```
fs.readFile('f.txt',
 count('A',data))
```

d
```
fs.readFile('f.txt', (data) => {
 count('A',data) })
```

13 *Necesitamos utilizar el módulo* `'net'` *para enviar a un servidor una petición estructurada que consta de cuatro atributos internos. ¿Cuál es la mejor opción para realizar el 'marshalling' y 'unmarshalling' de ese mensaje?*

a No debe hacerse nada. Los objetos se pueden transmitir directamente.

b Convertir cada atributo en un par (longitud, cadena), construir un vector de estos pares y enviar el vector al servidor. El servidor hará el 'unmarshalling' cuando lo reciba.

c Utilizar JSON.parse() en el cliente y JSON.stringify() en el servidor.

d Utilizar JSON.stringify() en el cliente y JSON.parse() en el servidor.

14 *Necesitamos escribir una función* `g` *con un número indeterminado de argumentos de tipo entero. Esta función debe devolver la suma de todos esos argumentos. Ya hemos escrito este cuerpo para la función. ¿Qué signatura debería utilizarse con ese código?*

```
{  let sum=0
   for (let j=0;j<args.length;j++)
      sum += args[j]
   return sum
}
```

a function g(…args)

b function g()

c function g(args,…)

d function g(args)

15 *Consideremos un programa P que mantiene en* `names` *un vector de nombres de ficheros de texto y en* `str` *una cadena. Sus valores se obtienen desde la línea de órdenes. P debe mostrar todas las líneas en esos ficheros que contengan el valor de* `str`*. Cada línea debe estar precedida por el nombre de su fichero. Una primera solución incorrecta se muestra seguidamente. Para corregirla se debería:*

```
const fs=require('fs')
const names=...
const str=...
for (var i=0; i<names.length; i++)
 fs.readFile(names[i], (e,x) => {
  if (e) return
  let lines=(x+'').split('\n')
  for (var j=0; j<lines.length; j++)
   if (lines[j].includes(str))
    console.log(names[i],':',lines[j])
 })
```

a No hay ninguna solución.

b Reemplazar `var i` con `let i`

c Reemplazar `var j` con `let j`

d En la línea 4 debería usarse: `i<=names.length`

16 *Suponiendo que en este directorio se encuentra un fichero* `a` *con contenido* `Nada,` *indica cuál es el resultado de ejecutar el siguiente código*

```
const fs = require('fs')
fs.readFile('a', 'utf8', (err,data) => {
   console.log(data)})
console.log('Fin')
```

a Se muestra en pantalla dos líneas con el texto:
Fin
Nada

b Se muestra en pantalla dos líneas con el texto:
Nada
Fin

c Se muestra en pantalla una línea con el texto:
Fin

d Se muestra en pantalla un mensaje de error (`syntaxError`)

17 *En ZeroMQ, en el patrón PUB-SUB:*

a Los SUBs no pueden conectarse con más de un PUB.

b Los SUBs no pueden subscribirse al prefijo vacío o nulo.

c Los SUBs pueden subscribirse a más de un prefijo.

d Los SUBs pueden conectarse con un PUB en cualquier momento sin perder ninguno del total de mensajes emitidos por el PUB.

18 *Para enviar un mensaje multisegmento en ZeroMQ se debe:*

a Enviar partes de cada segmento en una operación send() diferente.

b Utilizar cada segmento como un argumento diferente en una única operación send().

c Todas las opciones son válidas.

d Construir un vector con todos esos segmentos y utilizarlo como el único argumento de send().

19 *Para recibir un mensaje multisegmento en un socket ZeroMQ, se debe:*

a Especificar un único parámetro Array en el 'listener' del evento.

b Considerar que habrá tantos parámetros en el 'listener' del evento como segmentos tenga el mensaje.

c No utilizar parámetros en el 'listener', pues los segmentos estarán en el vector '`segments`' ya predefinido.

d Especificar un único parámetro Object en el 'listener' del evento.

20 *Los sockets REQ de ZeroMQ son, en cierto sentido, sincrónicos porque:*

a No transmiten una nueva petición hasta que reciban la respuesta para la petición actual.

b El proceso emisor permanece suspendido en socket.send() hasta que reciba una respuesta.

c No aceptarán la respuesta a la petición actual hasta que envíen la petición siguiente.

d Son gestionados por un único hilo.

21 *Este patrón de comunicación de ZeroMQ es bidireccional y asincrónico:*

a REQ/REP.

b PUSH/PULL.

c Ninguno de ellos.

d PUB/SUB.

22 *Este socket ZeroMQ tiene múltiples colas de entrada y de salida, una por cada conexión:*

a REQ.

b PUSH.

c REP.

d Ninguno de ellos.

23 *Con el fin de implementar un modelo de comunicación clente-servidor usando ZeroMQ, hemos decidido asociar un socket PUSH en el cliente y otro PULL en el servidor (par para envío y recepción de la solicitud) y otro socket PUSH en servidor y otro PULL en cliente (par para envío de la respuesta). ¿Esta solución se comporta correctamente?*

a Sí. Con dicha solución el servidor puede gestionar tantas solicitudes como se necesite de forma concurrente.

b No. Los canales PUSH-PULL pueden perder mensajes si la tasa de envío es elevada.

c No. Los canales PUSH-PULL no son asincrónicos.

d No. Cuando conectamos varios clientes al mismo servidor, el servidor puede remitir respuestas a clientes a los que no les corresponde dicha respuesta.

24 *Disponemos de un servidor que necesita al menos 100 ms para procesar cada solicitud que recibe. Asumimos que dicho servidor utiliza un socket de tipo REP para recibir las peticiones y remitir las respuestas. Suponemos un programa cliente que interactúa con el servidor y dispone del código que aparece a continuación. Una vez iniciada la ejecución del cliente, ¿Cuándo imprimirá su mensaje?*

```
const zmq = require('zeromq')
const s = zmq.socket('req')
s.connect(...)
s.on(...) // Irrelevant
for(let i=0; i<100; i++)
  s.send(['request',i+'',...])
console.log("%d requests have been sent!",i)
```

a No puede imprimir nada

b De forma prácticamente inmediata

c Tras, al menos, 100*100=10000ms (o sea, 10 segundos)

d Tras, al menos, 10 minutos

25 **[ver Server1.js].** *La variable* `counter` *se utiliza en* `Server1.js` *para registrar cuántos:*

- **a** Mensajes ha recibido el servidor.
- **b** Clientes han interactuado con el servidor.
- **c** Clientes están actualmente conectados con el servidor.
- **d** Conexiones ha cerrado el servidor.

26 **[ver Server1.js, Client1.js].** *Vamos a asumir que tenemos un* `Server1` *en ejecución. Si arrancamos la ejecución de un único* `Client1` *en la misma máquina, entonces dicho cliente:*

- **a** Recibe un único mensaje de respuesta, tras lo cual termina.
- **b** Recibe varios mensajes con el valor de `counter` (uno cada dos segundos), y muestra en pantalla los valores indicados.
- **c** Termina de forma inmediata, dado que no envía ni recibe ningún mensaje.
- **d** Recibe un mensaje `error` cada 2 segundos, y muestra en pantalla el error correspondiente.

27 **[ver Server1.js, Client1.js].** *Suponemos que tras la línea 26 de* `server1.js` *insertamos una nueva línea (26bis) con la sentencia* `c.end()`. *Como consecuencia de dicho cambio, el comportamiento de* `client1` *y* `server1` *cambia de la siguiente forma:*

- **a** Cada cliente termina en cuanto recibe su primera respuesta.
- **b** El servidor termina en cuanto remite su primera respuesta.
- **c** El servidor termina en cuanto un segundo cliente intenta conectar con él.
- **d** No hay cambios en el comportamiento. Ambos procesos se comportan de la misma forma que con el código original.

28 **[ver Server1.js, Client1.js].** *Si arrancamos un proceso* `Server1,` *y luego tres procesos* `Client1,` *entonces:*

- **a** Únicamente interactúa con `Server1` el primer `Client1.`
- **b** Todos los procesos `Client1` obtienen el valor 1 en sus respuestas.
- **c** Cada vez que se arranca un nuevo proceso `Client1,` si existe un `Client1` previo, dicho `Client1` previo cierra su conexión.
- **d** En algún momento todos los procesos `Client1` obtienen en sus respuestas el valor 3.

29 **[ver Server1.js, Client1.js].** *Arrancamos un proceso* `Server1` *y luego otro proceso* `Client1.` *Cada vez que el cliente envía un mensaje al servidor, y el servidor posteriormente remite una respuesta, el valor de las respectivas variables* `counter` *es:*

- **a** Idéntico en ambos procesos
- **b** Servidor: número de peticiones procesadas. Cliente: 1
- **c** Servidor: 1. Cliente: número de peticiones procesadas.
- **d** Servidor: número de peticiones procesadas. Cliente: número de peticiones procesadas

30 **[ver Server1.js, Client1.js].** *Necesitamos escribir un segundo programa cliente (*`Client2.js`*) que obtenga de forma periódica (cada 10 segundos) la carga de trabajo del servidor. Indica los cambios a introducir en el código de* `Client1.js` *para obtener el código necesario para* `Client2.js`:

- **a** línea 6: `client.write('reqLoad')`
línea 7: `setTimeout(stage,10000)`
- **b** línea 6: `client.write('reqLoad')`
línea 7: `setInterval(stage,10000)`
- **c** línea 6: `client.write('getLoad')`
línea 7: `setTimeout(stage,10)`
- **d** línea 6: `client.write('getLoad')`
línea 7: `setInterval(stage(),10)`

31 *En la parte final de la práctica 1 se proporciona el código* `Proxy.js`. *Anteriormente se ha trabajado la capacidad de comunicar un par de componentes (netClient.js y netServer.js), añadiendo en la respuesta del servidor un número que representa la carga de trabajo a la que se encuentra sometido.*
¿Es posible emplear este Proxy original (`Proxy.js`*) para que intermedie entre* `netClient.js` *y* `netServer.js`*?*

a No, porque Proxy está preparado para intermediar entre un navegador web y un servidor web, pero netClient y netServer no interactúan por HTTP.

b Directamente no, aunque algunos cambios en la lógica de `Proxy.js`, especialmente dentro de `net.createServer()`, pueden arreglar el nuevo problema.

c Directamente no, aunque algunos cambios en las constantes de `Proxy.js`, especialmente `REMOTE_PORT` y `REMOTE_IP`, pueden arreglar el nuevo problema.

d Sí, sin necesidad de ninguna modificación sobre el código y datos originales.

32 *En la práctica 1 aparece una modificación denominada* **Proxy programable**. *En el código del programador debe incluirse el envío de un mensaje similar a* `JSON.stringify ({remote_ip:'158.42.4.23', remote_port:80})`
Selecciona qué debería hacer este nuevo proxy al recibir ese mensaje.

a Debe extraer la información mediante `JSON.parse` y usar cada pieza.

b Debe modificar `LOCAL_PORT` y `LOCAL_IP`.

c Debe extraer la información mediante `JSON.stringify` y usar cada pieza.

d Debe distinguir, por el contenido, si se trata del mensaje de un cliente o del programador.