TSR

Este examen incluye 20 cuestiones de opción múltiple. Cada una de ellas solo tiene una respuesta correcta. Debes contestar en otra hoja. Las respuestas correctas aportan 0.5 puntos a tu calificación. Las erróneas descuentan 0.167 puntos.

TEORÍA

1. Este NO ES uno de los "aspectos relevantes" de los sistemas distribuidos:

a	Mejorar la eficiencia de las aplicaciones, dividiendo el problema a resolver en tareas y ejecutando cada tarea en un agente / ordenador diferente.
b	Proporcionar transparencia de fallos.
c	Permitir la compartición de recursos, especialmente de aquellos dispositivos que resulten caros y puedan accederse de manera remota.
d	Facilitar el despliegue de las aplicaciones.

2. En los sistemas de computación en la nube, la tecnología de virtualización de equipos es un mecanismo básico para este modelo de servicio:

a	SLA.
b	SaaS.
c	IaaS.
d	*Middleware* de comunicaciones (basado en mensajes).

3. En el modelo SaaS (para sistemas de computación en la nube) esta afirmación es cierta:

a	Sus servicios pueden accederse localmente, sin necesidad de utilizar la red, empleando virtualización.
b	Sus interacciones cliente-servidor deben basarse en un *middleware* de comunicaciones asincrónicas basado en mensajes.
c	Proporciona servicios *software* distribuidos a sus clientes, generalmente bajo un modelo de pago por uso.
d	Los usuarios de los servicios deciden de qué manera se desplegarán los programas.

4. El Tema 2 recomienda el paradigma de programación asincrónica porque ese paradigma...

a	…logra que el despliegue de aplicaciones sea trivial.
b	…está basado en eventos y asegura la ejecución atómica de cada acción.
c	…proporciona transparencia de fallos.
d	…utiliza *proxies* inversos y, debido a ello, es altamente escalable.

5. Si consideramos los aspectos de la sincronía vistos en el Tema 2, es cierto que...:

a	Los relojes lógicos generan procesos sincrónicos.
b	El orden (sincrónico) de los mensajes está basado en acotar el tiempo de propagación de los mensajes.
c	Los procesos sincrónicos avanzan en pasos. En cada paso, todo proceso completa una acción.
d	La comunicación sincrónica requiere que los canales mantengan los mensajes enviados hasta que los receptores puedan aceptarlos.

6. Estas oraciones relacionan el *middleware* con los estándares. ¿Cuál es falsa?

a	El uso de estándares permite que los *middleware* y las implementaciones de agentes realizadas por diferentes empresas sean interoperables.
b	El uso de estándares proporciona una interfaz de alto nivel en los *middleware*. Así, las tareas de programación resultan más sencillas.
c	Las APIs proporcionadas por sistemas *middleware* no siempre son estándar. ZeroMQ es un ejemplo.
d	Los sistemas *middleware* no deben respetar ningún estándar, pues los estándares solo se definen para los elementos internos de los sistemas operativos.

7. ¿Cuál de los siguientes elementos de comunicación puede considerarse un ejemplo de *middleware*?

a	El protocolo IP.
b	Un servicio de nombres distribuido.
c	TCP.
d	El servidor APACHE.

8. En el ámbito de los sistemas *middleware*, ¿cuáles son los problemas de los sistemas de objetos distribuidos cuando son comparados con los sistemas de mensajería?

a	Su acoplamiento (potencialmente alto) puede conducir a bloqueos prolongados cuando algún recurso compartido es utilizado concurrentemente por muchos agentes.
b	No proporcionan transparencia de ubicación.
c	Facilitan un bajo nivel de abstracción, complicando los programas resultantes.
d	Su comportamiento es excesivamente asíncrono, y por ello no pueden depurarse fácilmente.

TSR

SEMINARIOS

9. Considérese este programa:

```
var fs=require('fs');
if (process.argv.length<5) {
  console.error('More file names are needed!!');
  process.exit();
}
var files = process.argv.slice(2);
var i=-1;
do {
  i++;
  fs.readFile(files[i], 'utf-8', function(err,data) {
      if (err) console.log(err);
      else console.log('File '+files[i]+': '+data.length+' bytes.');
  })
} while (i<files.length);
console.log('We have processed '+files.length+' files.');
```

Esta afirmación es cierta si asumimos que ningún error aborta su ejecución y se pasan suficientes nombres de fichero como argumentos desde la línea de órdenes:

a	Debido a la asincronía del *callback* empleado en readFile(), este programa no muestra en cada iteración el nombre y longitud correctos para cada fichero.
b	Muestra el nombre y tamaño de cada fichero recibido como argumento.
c	Muestra "We have processed 0 files" como su primer mensaje en pantalla.
d	Descarta algunos de los nombres de fichero proporcionados como argumentos tras los elementos "node nombre-programa".

10. La siguiente afirmación sobre el programa de la cuestión anterior es cierta:

a	Necesita varios turnos para completar su ejecución, pues cada fichero a leer necesita un turno para su *callback*.
b	El incremento de la "i" (instrucción "i++") está ubicado incorrectamente. Debería estar dentro del *callback*.
c	Este programa muestra un error y finaliza si se han pasado menos de cinco nombres de fichero como argumentos.
d	Muestra el mismo tamaño en todas las iteraciones. Se necesita una clausura para evitar este comportamiento incorrecto.

11. Respecto a los algoritmos de exclusión mutua del Seminario 2, esta afirmación es cierta:

a	El algoritmo de servidor central gestiona correctamente aquellas situaciones en las que ese servidor central falla.
b	El algoritmo de anillo virtual unidireccional no pierde el *token* si el proceso actualmente en la sección crítica falla.
c	El algoritmo de difusión con relojes lógicos usa menos mensajes que el algoritmo de difusión basado en cuórums.
d	El algoritmo de difusión con relojes lógicos cumple las tres condiciones de corrección del problema de exclusión mutua.

12. Considerando este programa y sabiendo que no genera ningún error...

```
var ev = require('events');
var emitter = new ev.EventEmitter;
var num1 = 0;
var num2 = 0;
function myEmit(arg) { emitter.emit(arg,arg) }
function listener(arg) {
   var num=(arg=="e1"?++num1:++num2);
   console.log("Event "+arg+" has happened " + num + " times.");
   if (arg=="e1") setTimeout( function() {myEmit("e2")}, 3000 );
}

emitter.on("e1", listener);
emitter.on("e2", listener);
setTimeout( function() {myEmit("e1")}, 2000 );
```

La siguiente afirmación es cierta:

a	El evento “e1” ocurre una sola vez, dos segundos después de iniciarse el proceso.
b	El evento “e2” nunca ocurre.
c	El evento “e2” se da periódicamente, cada tres segundos.
d	El evento “e1” se da periódicamente, cada dos segundos.

13. Considerando el programa de la cuestión anterior, la siguiente afirmación es cierta:

a	El primer evento “e2” ocurre tres segundos después de iniciarse el proceso.
b	Como ambos eventos utilizan el mismo *listener*, ambos muestran mensajes con exactamente el mismo contenido cuando ocurren.
c	El primer evento “e2” ocurre dos segundos después del primer evento “e1”.
d	Ninguno de los eventos ocurre dos o más veces.

14. En ØMQ, el patrón de comunicaciones REQ-REP se considera sincrónico porque:

a	Ambos *sockets* están conectados o han realizado un “bind()” sobre el mismo URL.
b	Ambos *sockets* son bidireccionales.
c	El *socket* REP utiliza una operación sincrónica para manejar los mensajes recibidos.
d	Tras enviar un mensaje M, ambos *sockets* no pueden transmitir otro mensaje hasta que se haya recibido una respuesta a M (REQ) o una nueva petición (REP).

15. Considerando estos dos programas NodeJS...

```
// server.js
var net = require('net');
var server = net.createServer(
 function(c) {//'connection' listener
  console.log('server connected');
  c.on('end', function() {
   console.log('server disconnected');
  });
  c.on('data', function(data) {
   console.log('Request: ' +data);
   c.write(data+ 'World!');
  });
});
server.listen(9000);
```

```
// client.js
var net = require('net'); var i=0;
var client = net.connect({port:
  9000}, function() {
    client.write('Hello ');
  });
client.on('data', function(data) {
  console.log('Reply: '+data);
  i++; if (i==1) client.end();
});
client.on('end', function() {
  console.log('client ' +
    'disconnected');
});
```

Esta afirmación es cierta:

a	El servidor termina tras enviar su primera respuesta al primer cliente.
b	El cliente nunca termina.
c	El servidor puede gestionar múltiples conexiones.
d	El cliente no puede conectar con el servidor.

16. Los algoritmos de elección de líder (del Seminario 2)...

a	...necesitan la ejecución previa de un algoritmo de exclusión mutua, pues la identidad del líder se guarda en un recurso compartido y solo puede modificarla un proceso.
b	…necesitan consenso entre todos los procesos participantes: todos deben elegir un mismo líder.
c	…no necesitan identidades únicas para cada proceso.
d	...deben respetar orden causal.

17. Se quiere desarrollar un programa de elección de líder en NodeJS y ØMQ, utilizando el primer algoritmo del Seminario 2: el de anillo virtual. Para ello, selecciona la mejor opción de entre las siguientes:

a	Cada proceso usa un *socket* REQ para enviar mensajes a su sucesor en el anillo y un *socket* REP para recibir mensajes de su predecesor.
b	Cada proceso usa un *socket* ROUTER para enviar mensajes a su sucesor en el anillo y un *socket* DEALER para recibir mensajes de su predecesor.
c	Cada proceso usa un *socket* SUB para enviar mensajes a su sucesor en el anillo y un *socket* PUB para recibir mensajes de su predecesor.
d	Cada proceso usa un *socket* PUSH para enviar mensajes a su sucesor en el anillo y un *socket* PULL para recibir mensajes de su predecesor.

18. Se quiere desarrollar un programa de elección de líder en NodeJS y ØMQ, utilizando el segundo algoritmo (intimidador o "bully") del Seminario 2. Para soportar los mensajes "elección" (para preguntar a los mejores candidatos acerca de su vivacidad) y "respuesta" (a un "elección" previo, confirmando la vivacidad), una alternativa viable podría ser, asumiendo N procesos:

a	Un *socket* REQ conectado para enviar "elección" a los demás N-1 procesos y recibir sus "respuestas" y un *socket* REP ligado a un puerto local, para recibir "elecciones" y enviar "respuesta".
b	Un único *socket* DEALER para enviar "elección" y "respuesta" a los demás N-1 procesos. El mismo socket se utilizará para recibir los mensajes de los demás.
c	N-1 *sockets* PUSH para enviar "elección" y "respuesta" a los demás N-1 procesos. Un único *socket* SUB para recibir los mensajes de los demás.
d	Un único *socket* PULL para recibir mensajes. N-1 *sockets* PUSH conectados a los PULL de los demás procesos, para enviar "elección" y "respuesta" cuando se necesite.

19. ¿Cuál es el tipo de *socket* ØMQ que utiliza múltiples colas de envío?

a	El tipo PUB, para gestionar sus difusiones.
b	El tipo PUSH, para gestionar múltiples operaciones send() asincrónicas.
c	El tipo REQ, en caso de estar conectado a múltiples *sockets* REP.
d	El tipo ROUTER, utilizando una cola de envío para cada conexión.

20. Si consideramos estos programas...

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

```
// server.js
var zmq = require('zmq');
var rp = zmq.socket('rep');
var port = process.argv[2] || 8888;
rp.bindSync('tcp://127.0.0.1:'+port);
rp.on('message', function(msg) {
  var j = parseInt(msg);
  rp.send([msg,(j*3).toString()]);
});
```

…y suponemos que hemos iniciado un cliente y dos servidores con esta orden:
$ node client & node server 8888 & node server 8889 &
La siguiente afirmación es cierta:

a	Un servidor recibe todas las solicitudes con valor par para "i" y el otro recibe todas las solicitudes con valor impar para "i".
b	Cada servidor recibe, gestiona y contesta las 100 solicitudes. Así, el cliente recibe y muestra 200 respuestas.
c	Algunas solicitudes iniciales se pierden, pues el cliente ha sido iniciado antes de que empezara el primer servidor.
d	Si uno de los servidores falla durante la ejecución, el cliente y el otro servidor gestionarán sin interrumpirse las demás solicitudes y respuestas.