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. Si una aplicación se diseña como distribuida, los datos a procesar deben repartirse entre sus procesos. De esta manera, su eficiencia normalmente mejorará. Por tanto, este es uno de los aspectos relevantes de los sistemas distribuidos.
b	Proporcionar transparencia de fallos. Si una aplicación es ejecutada por un solo agente, cuando ese agente falle la aplicación entera fallará. Por el contrario, en una aplicación distribuida se necesita múltiples agentes que colaboren entre sí. Así, cuando alguno de ellos falle, los demás podrán todavía completar sus respectivas ejecuciones y proporcionar un resultado válido. Para ello, los agentes se replican. Por tanto, las aplicaciones distribuidas pueden proporcionar transparencia de fallos y esta es una de sus principales ventajas.
c	Permitir la compartición de recursos, especialmente de aquellos dispositivos que resulten caros y puedan accederse de manera remota. Ese tipo de compartición no puede ocurrir en sistemas formados por un solo ordenador. Por ello, esta es una de las razones principales que conducen al uso de sistemas distribuidos.
d	Facilitar el despliegue de las aplicaciones. Las aplicaciones distribuidas constan de varios componentes, funcionando en múltiples ordenadores. Esto complica su configuración y la resolución de dependencias entre los componentes. Además, una vez ya estén en ejecución, sus usuarios esperan continuidad en el servicio. Esa continuidad resulta difícil en caso de que se actualicen los programas que componen esa aplicación. Esas tareas (instalación, configuración, resolución de dependencias, actualizaciones) forman parte del despliegue y resultan mucho más complejas en las aplicaciones distribuidas que en las formadas por un solo programa y una sola instancia.

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. Estas iniciales corresponden a los "acuerdos de nivel de servicio". Los SLA no son un ejemplo de modelo de servicio. Son un acuerdo entre el proveedor de servicios y el cliente sobre el nivel de calidad de servicio que deba garantizarse. Los modelos de servicio indican cuál es el servicio principal que proporciona un proveedor de sistemas de computación en la nube.

b	SaaS. SaaS sí es un modelo de servicio para los sistemas de computación en la nube. Sin embargo, su principal objetivo es facilitar servicios "software" a sus clientes. Por tanto, se centra en el diseño y la arquitectura de esas aplicaciones distribuidas y su principal preocupación no es la virtualización de la infraestructura.
©	IaaS. La virtualización de la infraestructura es el mecanismo básico por el que deben preocuparse los proveedores de infraestructura. Como el modelo de servicio IaaS está enfocado a la provisión de infraestructura, la virtualización es la tecnología clave en ese modelo.
d	*Middleware* de comunicaciones (basado en mensajes). Los niveles "middleware" no son ejemplos de modelos de servicios para la computación en la nube.

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. El modelo SaaS no gestiona accesos a servicios "locales". Los modelos de servicio de computación en la nube se han definido para la provisión de servicios a usuarios remotos en todos los casos. Por tanto, la primera parte de la afirmación no tiene sentido. Además, la virtualización de la infraestructura no es el principal objetivo de ese modelo.
b	Sus interacciones cliente-servidor deben basarse en un *middleware* de comunicaciones asincrónicas basado en mensajes. En ese modelo no se obliga a que las comunicaciones estén orientadas a mensajes ni a que sean asincrónicas.
©	Proporciona servicios *software* distribuidos a sus clientes, generalmente bajo un modelo de pago por uso. Esta afirmación puede considerarse una definición breve de ese modelo.
d	Los usuarios de los servicios deciden de qué manera se desplegarán los programas. El despliegue de los servicios jamás debería ser una responsabilidad de los usuarios. En el modelo SaaS se respeta esa restricción. Por ello, los usuarios de un servicio no participan en esas tareas del ciclo de vida de los servicios.

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

a	…logra que el despliegue de aplicaciones sea trivial. Los paradigmas de programación no tienen un efecto importante sobre las tareas de despliegue. El despliegue incluye aquellas etapas del ciclo de vida de los programas que sigan a su desarrollo (instalación, configuración, activación, actualización, escalado, desactivación, eliminación...) y muchas de esas etapas no dependen del modelo de programación utilizado para desarrollar la aplicación.

b	…está basado en eventos y asegura la ejecución atómica de cada acción. Ambas características se respetan en ese paradigma de programación. Además, esas características son interesantes, pues la ejecución atómica evita las condiciones de carrera y la orientación a eventos proporciona una correspondencia directa entre lo que se especifique en un algoritmo y el comportamiento real de las acciones de la aplicación.
c	…proporciona transparencia de fallos. La transparencia de fallos solo puede proporcionarse cuando los procesos de la aplicación estén replicados. El uso de replicación debe preverse durante la etapa de diseño de la aplicación y el número de réplicas variará dinámicamente en función de algunos factores (por ejemplo, el nivel actual de carga). El paradigma de programación no influye en ninguno de estos aspectos.
d	…utiliza *proxies* inversos y, debido a ello, es altamente escalable. Los *proxies* inversos son un ejemplo de mecanismo de propagación de mensajes y uso de una memoria "caché" (en cuanto a su proximidad y rapidez de acceso) en la gestión de resultados. Esos aspectos no están considerados en la definición de la programación asincrónica.

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

a	Los relojes lógicos generan procesos sincrónicos. Para afirmar que los procesos utilizados en una aplicación o algoritmo son sincrónicos, estos deben avanzar sus acciones de manera sincronizada: en cada paso o etapa, todo proceso del algoritmo debe completar una acción. Los relojes lógicos no logran ese tipo de sincronización fuerte. Con ellos se puede etiquetar cada acción realizada para determinar posteriormente si ha habido dependencias de orden entre algunas de las acciones realizadas por los procesos. En ningún momento condicionan cuándo se podrá iniciar una nueva acción o cuánto debe tardarse en finalizarla.

b	El orden (sincrónico) de los mensajes está basado en acotar el tiempo de propagación de los mensajes. Cuando pueda acotarse el tiempo de transmisión o propagación de los mensajes se podrá hablar de "canales sincrónicos". El "orden sincrónico de los mensajes" se refiere a otra cosa. Concretamente, a que los mensajes transmitidos en esa aplicación o algoritmo respetarán cierto orden de entrega; por ejemplo, FIFO, causal, total... Para ello, los nodos utilizarán "buffers" de recepción y algún protocolo de comunicaciones que permita respetar ese orden. Esos protocolos no necesitan utilizar canales sincrónicos. Por ejemplo, para garantizar orden FIFO basta con que cada proceso emisor numere los mensajes que él envía. Los receptores tendrán que respetar ese orden secuencial de numeración a la hora de entregarlos. Si se utilizaran canales sincrónicos y no se numeraran los mensajes, el orden FIFO podría no respetarse.
©	Los procesos sincrónicos avanzan en pasos. En cada paso, todo proceso completa una acción. Esta es una definición concisa y válida del término "proceso sincrónico".
d	La comunicación sincrónica requiere que los canales mantengan los mensajes enviados hasta que los receptores puedan aceptarlos. Esa es la definición de "comunicación persistente". La comunicación sincrónica se da cuando el emisor se bloquea hasta que reciba algún reconocimiento sobre la correcta entrega del mensaje que acaba de enviar. Esos dos tipos de comunicación consideran aspectos diferentes. Por tanto, la afirmación es falsa.

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. Esta afirmación es cierta. Uno de los objetivos de los estándares es facilitar la interoperabilidad.

b	El uso de estándares proporciona una interfaz de alto nivel en los *middleware*. Así, las tareas de programación resultan más sencillas. Cierto. Las interfaces proporcionadas por un *middleware* suelen ser de alto nivel y estar basadas en las soluciones definidas en algunos estándares para ciertos problemas.
c	Las APIs proporcionadas por sistemas *middleware* no siempre son estándar. ZeroMQ es un ejemplo. Cierto. ZeroMQ es un ejemplo de *middleware* para comunicación asincrónica. Hay un estándar en ese campo: AMQP, pero ZeroMQ no lo respeta.
d	Los sistemas *middleware* no deben respetar ningún estándar, pues los estándares solo se definen para los elementos internos de los sistemas operativos. Falso. Los estándares no están limitados a los elementos internos de los sistemas operativos. Los sistemas *middleware* se ubican sobre el sistema operativo y suelen respetar algunos estándares, principalmente cuando la interoperabilidad sea uno de sus objetivos.

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

a	El protocolo IP. Todo *middleware* debe ubicarse sobre el sistema operativo. El protocolo IP está en el nivel de red (nivel 3) en una arquitectura de comunicaciones basada en niveles. Los niveles entre 2 y 4, inclusive, son gestionados por el sistema operativo. Por tanto, este protocolo está por debajo del *middleware* y no puede ser un ejemplo de él.
b	Un servicio de nombres distribuido. Este tipo de servicio está por encima del sistema operativo y facilita una base para proporcionar transparencia de ubicación en un sistema distribuido. Sus servicios son utilizados por aplicaciones que están conceptualmente sobre él en una arquitectura estructurada en niveles. Por ello, puede considerarse un ejemplo de *middleware*.
c	TCP. TCP es un protocolo de transporte. Por tanto, pertenece al nivel 4 y está gestionado por el sistema operativo. Al igual que en el apartado "a", es un protocolo ubicado en niveles inferiores a los ocupados por el *middleware*.
d	El servidor APACHE. Este es un ejemplo de "servidor web". Desde el punto de vista de las arquitecturas de comunicaciones, pertenece al nivel de aplicación. Por ello, está ubicado sobre el *middleware*.

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. Cierto. Los sistemas de objetos facilitan transparencia de ubicación. Toda invocación ofrece la imagen de ser local, pero puede acceder a objetos remotos. Esos objetos remotos necesitarán mecanismos de control de concurrencia para gestionar múltiples invocaciones simultáneas, desde diferentes clientes. Esos mecanismos podrán bloquear la actividad de los agentes durante intervalos prolongados.

b	No proporcionan transparencia de ubicación. Falso. Las interacciones entre *proxies* y esqueletos facilitan transparencia de ubicación, pues el *proxy* ofrece la misma interfaz que el objeto remoto y oculta el intercambio de mensajes.
c	Facilitan un bajo nivel de abstracción, complicando los programas resultantes. Al contrario: ofrecen un nivel de abstracción alto. Las aplicaciones se pueden desarrollar fácilmente bajo este modelo.
d	Su comportamiento es excesivamente asíncrono, y por ello no pueden depurarse fácilmente. Las aplicaciones orientadas a objetos usan interacciones cliente-servidor, por lo que el cliente (emisor inicial) permanece bloqueado hasta recibir la respuesta. Esas interacciones, como acabamos de explicar, serían un ejemplo de comunicación sincrónica. Por tanto, su comportamiento no puede calificarse como asíncrono.

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-1);
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:

@	Debido a la asincronía del *callback* empleado en readFile(), este programa no muestra en cada iteración el nombre y longitud correctos para cada fichero. El programa muestra la longitud correcta en cada iteración, pero no el nombre. La longitud se calcula utilizando uno de los argumentos del *callback* ("data"). En ese argumento se recibe el contenido del fichero correspondiente, por lo que ese cálculo siempre será válido. Sin embargo, el nombre se toma directamente del vector "files", utilizando para ello el valor actual de la variable "i". Como readFile() es una operación asincrónica, los *callbacks* irán siendo invocados una vez el bucle do..while haya finalizado todas sus iteraciones. En ese momento, el valor de "i" será igual a "file.length". Desafortunadamente el elemento "files[file.length]" tiene un valor "undefined", pues en esa componente no se ha llegado a guardar ningún nombre. Por ejemplo, si se pasaron tres nombres de fichero "a", "b" y "c" como argumentos, cada uno de esos nombres estará en las componentes "files[0]", "files[1]" y "files[2]", respectivamente, pero "files.length" será 3 y "files[3]" no guardará ningún valor.

b	Muestra el nombre y tamaño de cada fichero recibido como argumento. Será incapaz de mostrar el nombre, como ya se ha explicado en el apartado "a".
c	Muestra "We have processed 0 files" como su primer mensaje en pantalla. Falso. Sí que llega a mostrarse un mensaje "We have processed..." al empezar la ejecución, pero no visualiza "0 files" en su parte final. En su lugar, se muestra el número correcto de nombres de fichero recibidos como argumentos. Obsérvese que "files.length" ya está definido en ese momento y su valor debe ser superior a 2, pues la segunda instrucción del programa requiere que process.argv.length sea superior a 4 y posteriormente, en la sexta línea, se eliminan las dos primeras componentes de ese vector ("node" y el nombre del programa) para asignar el vector resultante a "files".
d	Descarta algunos de los nombres de fichero proporcionados como argumentos tras los elementos "node nombre-programa". No. Como ya hemos explicado en el apartado anterior, se utiliza el método "slice()" para eliminar las dos primeras componentes de "process.argv[]" cuando su contenido es copiado en la variable "files". Esas dos componentes no contienen ninguno de los nombres de fichero utilizados como argumentos.

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*. Las lecturas son asíncronas. Por ello, cuando cada una finaliza, llama a su *callback* asociado. Esa ejecución del *callback* se realiza en el contexto de un hilo independiente, y cada uno de esos hilos utilizará un turno de ejecución diferente en el *runtime* de "node".
b	El incremento de la "i" (instrucción "i++") está ubicado incorrectamente. Debería estar dentro del *callback*. No. Si se hiciera de esa manera, el bucle nunca finalizaría y se iniciarían infinitas lecturas del primer fichero cuyo nombre se ha pasado como argumento en la línea de órdenes.
c	Este programa muestra un error y finaliza si se han pasado menos de cinco nombres de fichero como argumentos. No. Eso solo ocurre cuando se pasen menos de tres nombres de fichero. Si se pasan tres o cuatro nombres, la ejecución procede con normalidad y no muestra ningún error asociado al número de argumentos facilitados.
d	Muestra el mismo tamaño en todas las iteraciones. Se necesita una clausura para evitar este comportamiento incorrecto. No. Como ya se explicó en la cuestión nueve, la longitud de cada fichero se muestra correctamente en cada iteración.

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. Falso. Si falla el servidor central, el algoritmo no podrá continuar. Los algoritmos centralizados suelen tener ese problema.

b	El algoritmo de anillo virtual unidireccional no pierde el *token* si el proceso actualmente en la sección crítica falla. Falso. El proceso que actualmente ejecute la sección crítica podrá hacer eso porque mantiene el *token*. Si fallara, ese *token* dejaría de existir.
c	El algoritmo de difusión con relojes lógicos usa menos mensajes que el algoritmo de difusión basado en cuórums. No. Los cuórums fueron introducidos en los algoritmos de difusión para reducir la cantidad de mensajes a utilizar. En lugar de enviar un mensaje a cada participante, en ellos basta con enviar un mensaje a cada miembro del cuórum. Cada quórum no incluye a todos los procesos que utilicen ese algoritmo.
d	El algoritmo de difusión con relojes lógicos cumple las tres condiciones de corrección del problema de exclusión mutua. Cierto. Satisface seguridad, vivacidad y orden causal. Esas son las tres condiciones de corrección para ese problema.

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. Sí. La última línea programa la generación de ese evento dos segundos después. El *listener* para “e1” incrementa “num1” y muestra “Event e1 has happened 1 times.” en la pantalla. Entonces, se programa la generación de “e2” tres segundos después. Pero en el *listener* ya no se realiza ningún “setTimeout()” adicional si el evento gestionado es “e2”. Por ello, “e1” ya no volverá a generarse.
b	El evento “e2” nunca ocurre. Falso. Como hemos explicado en el apartado “a”, el evento “e2” ocurre una vez.
c	El evento “e2” se da periódicamente, cada tres segundos. Falso. Como hemos explicado en el apartado “a”, el evento “e2” ocurre solo una vez.
d	El evento “e1” se da periódicamente, cada dos segundos. Falso. Como hemos explicado en el apartado “a”, el evento “e1” ocurre solo una vez.

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. No. Sucede cinco segundos después de iniciarse el proceso.

b	Como ambos eventos utilizan el mismo *listener*, ambos muestran mensajes con exactamente el mismo contenido cuando ocurren. No. Ambos eventos emplean el mismo *listener*, pero reciben un argumento. Ese argumento permite que el *listener* se comporte de manera diferente para cada evento.
c	El primer evento "e2" ocurre dos segundos después del primer evento "e1". No. Ese plazo dura tres segundos.
ⓓ	Ninguno de los eventos ocurre dos o más veces. Cierto. Ambos ocurren una sola vez.

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. Falso. Por ejemplo, un patrón de comunicación PUSH-PULL también exige que ambos *sockets* utilicen un mismo URL, pero el canal resultante es asíncrono. Por tanto, ese aspecto no es una característica de los canales sincrónicos.
b	Ambos *sockets* son bidireccionales. Falso. Los *sockets* ROUTER y DEALER son también bidireccionales y permiten definir un patrón de comunicaciones. Sin embargo, ROUTER y DEALER son asincrónicos. Por tanto, la bidireccionalidad no está ligada a la sincronía.
c	El *socket* REP utiliza una operación sincrónica para manejar los mensajes recibidos. Falso. Todos los *sockets* ZeroMQ usan un *listener* (es decir, un tipo de *callback*) para gestionar la recepción de mensajes. Los *listeners* tienen un comportamiento asíncrono.
ⓓ	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). Cierto. Este comportamiento obliga a que tanto REP como REQ se utilicen de manera sincrónica, siguiendo un orden secuencial y bloqueante en la transmisión de los mensajes. Así, un servidor no puede transmitir una nueva respuesta hasta que reciba una nueva petición. De la misma manera, un cliente no puede transmitir una nueva petición mientras no reciba una respuesta para la petición actual.

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. Falso. No hay ninguna instrucción que provoque la finalización del servidor en su programa. Por tanto, el servidor no finaliza de manera voluntaria.

b	El cliente nunca termina. Falso. Cuando el cliente reciba su primera respuesta, incrementará el valor de "i". Entonces, comprobará si "1" vale 1 (y así es). En ese caso, cierra su conexión. Cuando la conexión se cierra el cliente gestiona el evento "end" y a partir de ese momento la conexión desaparece. Con ello ya no puede haber ningún evento pendiente para el que el proceso cliente tenga un *listener* asociado y por ello el proceso finaliza en ese momento. Esto no causa la finalización del servidor, pues el servidor todavía puede gestionar otras conexiones con otros clientes (tanto en ese mismo momento como más tarde).
c	El servidor puede gestionar múltiples conexiones. Cierto. La operación createServer() utiliza un gestor de conexión como su *callback*. De esa manera, puede gestionar varias conexiones. Para cada conexión habrá *listeners* para sus eventos 'data' (recepción de mensajes) y 'end' (cierre de la conexión).
d	El cliente no puede conectar con el servidor. Falso. Si el cliente y el servidor son lanzados en un mismo ordenador, como ambos utilizan el mismo puerto local (9000) podrán comunicarse sin excesivos problemas. El servidor atiende nuevas conexiones en ese puerto, pues ha utilizado para ello la operación 'listen', mientras que el cliente se conecta al puerto con la operación 'connect'.

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. Falso. Cada proceso guarda la identidad del líder en una variable local. Por ser local, ninguno de ellos puede observar ni acceder directamente al valor mantenido por los demás procesos. Por tanto, no se necesita ningún algoritmo de exclusión mutua para gestionar esa identidad.
b	…necesitan consenso entre todos los procesos participantes: todos deben elegir un mismo líder. Cierto. Ese es el objetivo de estos algoritmos, que el valor elegido sea aceptado por todos.
c	…no necesitan identidades únicas para cada proceso. Falso. Se necesita utilizar identidades únicas para que cada proceso pueda distinguirse de los demás.
d	...deben respetar orden causal. Falso. La comunicación entre procesos no necesita orden causal en los algoritmos de elección de líder presentados en el Seminario 2. Eso fue un requisito para las soluciones al problema de exclusión mutua si se pretendía asegurar equidad entre los participantes.

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. Falso. Este algoritmo utiliza comunicación unidireccional. Eso significa que cada

	proceso recibe mensajes de su predecesor y los envía (cuando se necesite) a su sucesor. Los sockets REQ y REP no pueden utilizarse en ese tipo de comunicación. Son bidireccionales y no permiten el envío de un segundo mensaje mientras no se reciba otro mensaje entre tanto. Por ello, los sockets REQ se bloquearían al tratar de enviar un segundo mensaje y serían incapaces de transmitir nada más.
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. Falso. Aunque ambos son asíncronos, el socket ROUTER no puede enviar un mensaje a un proceso A mientras no haya recibido algún mensaje desde A. Cuando reciba ese primer mensaje, el socket ROUTER pasará como primer segmento al proceso receptor la identidad de la conexión establecida con A. Una vez conocida esa identidad, podrá utilizarla para enviar mensajes a A por esa conexión. Este algoritmo utiliza comunicación estrictamente unidireccional. Eso evita que cada socket ROUTER llegue a conocer la identidad de su proceso sucesor. Sin ella, no se podrá enviar ningún mensaje.
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. Falso. Aunque los sockets PUB y SUB realicen comunicación unidireccional, el socket SUB no puede ser utilizado para enviar mensajes y el socket PUB no puede utilizarse para recibirlos.
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. Cierto. Esta combinación es la más sencilla para implantar comunicación unidireccional.

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". Falso. Los mensajes "elección" y "respuesta" se pueden considerar mensajes de petición y respuesta, respectivamente. A primera vista, ese patrón podría ser soportado por sockets REQ y REP. Sin embargo, en este apartado se indica que el REQ de un proceso tendría que estar conectado a todos los REP de todos los demás procesos. De ser así, el emisor no tendría ningún control a la hora de especificar a qué proceso le está enviando el mensaje "elección" actual. El algoritmo indica que esos mensajes deben enviarse a los posibles candidatos (es decir, todos aquellos con un identificador mayor que el del emisor) y esa selección no puede llevarse a cabo con un único socket REQ. Si conectamos un socket REQ a múltiples sockets REP, sus mensajes de distribuyen siguiendo un orden circular. Únicamente el socket ROUTER permite elegir cuál de las conexiones establecidas por el socket deberá utilizarse en cada envío. Para ello se requiere además que previamente se haya recibido algún mensaje por esa misma conexión.

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. A la hora de enviar, los sockets DEALER ofrecen el mismo problema que los sockets REQ: ambos utilizan un turno circular entre sus conexiones. Por ello, no hay forma de seleccionar a qué otros procesos enviaremos el mensaje "elección". Eso hace que esta opción no sea viable.
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. Los sockets SUB no deben ser utilizados para recibir los mensajes enviados mediante sockets PUSH. Debería ser un socket PULL en lugar de SUB.
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. Este esquema sí que llegaría a funcionar. Teniendo N-1 sockets PUSH se podrá seleccionar a qué otro proceso le enviamos el mensaje "elección". Para ello el programa correspondiente debería saber qué socket corresponde a cada otro proceso (una forma sencilla de conseguirlo es mantener un vector de sockets, indexado con el identificador de proceso y forzar a que todos los mensajes "elección" incluyan el identificador de su emisor, para que así el receptor sepa qué socket utilizar para enviar la "respuesta" asociada). Mediante un solo socket PULL en cada participante se pueden recibir los mensajes enviados por los demás procesos y dirigidos a ese receptor.

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

a	El tipo PUB, para gestionar sus difusiones. No. Solo hay una cola de envío en este tipo de socket. Los mensajes difundidos previamente no llegan a ser recibidos por aquellos suscriptores que se conecten más tarde.

b	El tipo PUSH, para gestionar múltiples operaciones send() asincrónicas. Falso. Solo hay una sola cola de envío en este tipo de sockets. Cada mensaje solo se propaga a un receptor. En caso de múltiples conexiones, se sigue un turno circular para los envíos.
c	El tipo REQ, en caso de estar conectado a múltiples *sockets* REP. Falso. Solo hay una sola cola de envío en este tipo de sockets. Cada mensaje solo se propaga a un receptor. En caso de múltiples conexiones, se sigue un turno circular para los envíos.
d	El tipo ROUTER, utilizando una cola de envío para cada conexión. Cierto. Cada cola de envío se asocia a un identificador de conexión diferente. Debido a ello, en las operaciones send(), los sockets ROUTER necesitan (como el primer segmento del mensaje a enviar) la identidad de la conexión o cola de envío por la que debe transmitirse el mensaje.

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". Cierto. Cuando un socket REQ (o DEALER o PUSH) se conecte a múltiples URL, utilizará una gestión circular para distribuir los mensajes enviados a través de ellas. En este caso, el socket REQ se ha conectado a dos servidores diferentes. Como los mensajes se numeran consecutivamente, uno de los servidores recibe los mensajes con números impares y el otro aquellos con números pares.

b	Cada servidor recibe, gestiona y contesta las 100 solicitudes. Así, el cliente recibe y muestra 200 respuestas. No. Los mensajes se reparten entre ambos. Eso implicará que un servidor reciba 50 mensajes y el otro los 50 restantes.
c	Algunas solicitudes iniciales se pierden, pues el cliente ha sido iniciado antes de que empezara el primer servidor. No. Los mensajes no pueden enviarse mientras las conexiones correspondientes no se hayan establecido correctamente. Por ello, ningún mensaje llega a perderse con este tipo de sockets.
d	Si uno de los servidores falla durante la ejecución, el cliente y el otro servidor gestionarán sin interrumpirse las demás solicitudes y respuestas. No. Tras el fallo de uno de los servidores, el cliente permanecerá esperando su respuesta de manera indefinida. Eso implica que ya no podrá transmitir ningún mensaje más a través de la red. Por tanto, las demás solicitudes no llegarán a transmitirse. Eso provocará que sus respuestas tampoco sean generadas. El cliente llegará a terminar su bucle de envíos, pero esos mensajes permanecerán indefinidamente en la cola de envío. Su bloqueo se percibe al no mostrar ningún otro mensaje por pantalla acerca de la recepción de respuestas.