TSR: Primer Parcial

El examen consta de 20 preguntas de opción múltiple. En cada caso solo una de las respuestas es correcta. Se debe responder en una hoja aparte. En caso de respuesta correcta, ésta aportará 0.5 puntos. En caso de error, la contribución es negativa: -0.167.

TEORÍA

1. Considerando el Tema 1, estas afirmaciones describen correctamente algunos aspectos de los sistemas distribuidos:

a	Todo sistema concurrente es un sistema distribuido. Falso. Todos los sistemas distribuidos son ejemplos de sistemas concurrentes, pero no todos los sistemas concurrentes son distribuidos. Por ejemplo, un programa Java que use dos hilos (uno para interactuar con el usuario y otro para acceder a ficheros) generará un proceso concurrente, pero ese proceso no es una aplicación distribuida.
b	El servicio de correo electrónico es un ejemplo de sistema distribuido. Verdadero. En un servicio de correo electrónico hay diferentes tipos de agentes (servidores, clientes,...) y esos agentes se despliegan en múltiples ordenadores y necesitan colaborar entre sí para gestionar la redacción, envío, encaminamiento, almacenamiento y entrega de los correos.
c	Los agentes en un sistema distribuido no pueden compartir ningún recurso pues están ubicados en ordenadores diferentes. Falso. Algunos recursos pueden compartirse entre esos agentes. En caso contrario no se habría necesitado ningún algoritmo distribuido de exclusión mutua (de los que hemos visto varios ejemplos en el Seminario 2).
d	El programador de una aplicación distribuida no necesita preocuparse sobre la tolerancia a fallos, pues ya está garantizada implícitamente por el sistema distribuido. Falso. Para proporcionar tolerancia a fallos se necesita replicar los componentes. La replicación no se facilita automáticamente en todos los "sistema distribuidos". La gestión de la replicación suele ser responsabilidad de las aplicaciones distribuidas (algunos tipos particulares de "middleware" pueden cooperar en esa gestión).

2. Una de las razones para decir que la Wikipedia es una aplicación distribuida escalable es...

a	Desde su primera edición, se ha implantado en la nube siguiendo el modelo de servicio SaaS. Falso. Las primeras ediciones de la Wikipedia fueron generadas en 2001, antes de que el modelo de servicio SaaS empezara a popularizarse. Además, en esas ediciones la Wikipedia podía utilizar unos pocos ordenadores para proporcionar sus servicios.
b	Es un sistema LAMP, y todos los sistemas de este tipo son escalables. Falso. Un sistema LAMP no tiene por qué ser escalable. Debería complementarse con una estrategia adecuada de gestión de cachés, interacción "*stateless*" con los clientes y replicación de componentes para mejorar su escalabilidad.
c	Utiliza interacción P2P y esto mejora su escalabilidad. Falso. Usa interacción cliente-servidor.
d	Utiliza *proxies* inversos (como cachés) y replicación de componentes. Verdadero. Ambas técnicas mejoran la escalabilidad.

3. El objetivo principal del modelo de servicio PaaS es…

a	Automatizar la configuración, despliegue y actualización de servicios distribuidos y su

	adaptación a cargas variables. Verdadero. Esos son sus objetivos principales.
b	Automatizar la provisión de infraestructura. Falso. Ese es el objetivo principal del modelo de servicio IaaS.
c	Proporcionar servicios distribuidos bajo un modelo de pago a medida ("pay-as-you-go"). Falso. Ese es el objetivo principal del modelo de servicio SaaS.
d	Gestionar datos persistentes bajo un modelo de pago a medida ("pay-as-you-go"). Falso. Ese es el objetivo principal del modelo de servicio DBaaS (subconjunto del IaaS).

4. El tema 2 propone un modelo de sistema distribuido sencillo porque ese modelo…

a	…garantiza persistencia de datos. Falso. Los modelos de sistema no pueden asegurar propiedades funcionales. Para asegurar persistencia de datos (es decir, su durabilidad y tolerancia a fallos) los datos deberían replicarse.
b	…es necesario para comparar dos tipos de programación: asíncrona y multi-hilo. Falso. Ese no es el objetivo del modelo de sistema explicado en el Tema 2.
c	…proporciona una buena base para diseñar algoritmos distribuidos y para razonar acerca de su corrección antes de iniciar su desarrollo. Verdadero. Este es el objetivo principal de los modelos de sistema.
d	…demuestra que las situaciones de bloqueo de actividades impiden que los servicios escalen. Falso. El modelo propuesto no puede demostrar directamente esas condiciones. Proporciona la base para analizar algoritmos, pero los comportamientos bloqueantes dependen en muchos casos de algunos mecanismos utilizados para implantar los algoritmos, no de los algoritmos en sí.

5. Esta es la mejor solución para proporcionar persistencia de datos:

a	Usar servidores *stateful* (es decir, servidores con gestión de estado en sus interacciones). Falso. La gestión del estado de una sesión en una interacción cliente-servidor no garantiza que el estado mantenido en el dominio servidor supere situaciones de fallos. De hecho, cuando un servidor *stateful* falla, al cliente le resulta difícil reanudar su sesión con otro servidor que reemplace al que ha fallado.
b	Replicar los datos. Verdadero. Los datos deben replicarse para superar situaciones de fallos y garantizar su durabilidad y capacidad de acceso.
c	Usar los discos duros más fiables. Falso. Por muy fiables que sean, siempre podrán estropearse y perder su información.
d	Evitar los accesos concurrentes a los datos. Falso. Los accesos concurrentes pueden poner en peligro la consistencia de los datos, pero no su persistencia.

6. El modelo de sistema sencillo descrito en el Tema 2 está soportado directamente por la programación asíncrona porque…

a	…la programación asíncrona está basada en comunicación causal.

	Falso. La programación asincrónica no depende de ningún orden en la comunicación.
b	…los procesos en el modelo de sistema sencillo son multi-hilo. Falso. La programación asincrónica no implica procesos multi-hilo. De hecho, los procesos multi-hilo suelen ser sincrónicos.
c	…hay una traducción directa entre guardas + acciones en el modelo y eventos + *callbacks* en la programación asíncrona. Verdadero. En el modelo, las acciones a ejecutar por los procesos están protegidas con precondiciones o guardas. En la programación asincrónica, un proceso ejecuta un fragmento de código (*callback*) cuando se da un evento. Por tanto, el evento se comporta como la precondición y, una vez se da, inicia la ejecución del *callback* (que se corresponde con la acción del modelo).
d	…los procesos siguen implícitamente el modelo de fallos de parada en la programación asíncrona. Falso. Este tipo de programación no depende de ningún modelo de fallos.

7. Para desarrollar aplicaciones escalables resulta más recomendable un sistema (middleware) de mensajería (MoM, por sus siglas en inglés) que la invocación de objetos remotos (RMI, en inglés) porque…

a	MoM proporciona transparencia de ubicación, pero RMI no puede proporcionarla. Falso. RMI puede proporcionar transparencia de ubicación (p.ej., con proxies) pero MoM no siempre facilita ese tipo de transparencia. ZeroMQ es un ejemplo de MoM y los URL que utiliza no proporcionan transparencia de ubicación.
b	MoM es inherentemente asincrónico, mientras RMI es sincrónico. Verdadero. ZeroMQ es un ejemplo de MoM y proporciona comunicación asincrónica: un proceso puede continuar con su ejecución tras llamar a socket.send(), incluso en el patrón de comunicaciones REQ-REP. Por otra parte, RMI suspende al invocador hasta que la respuesta sea recibida. Por tanto, RMI es sincrónico y MoM asincrónico.
c	Los procesos que utilizan MoM asumen un espacio compartido de recursos. En RMI los procesos no pueden compartir recursos. Falso. MoM proporciona una imagen sin compartición, debido a su falta de transparencia de ubicación, mientras RMI proporciona normalmente una imagen donde todos los recursos se pueden utilizar (compartir) como entidades locales (debido a su transparencia de ubicación).
d	Los procesos que utilizan MoM están replicados automáticamente. En RMI, la replicación no está permitida. Falso. Tanto MoM como RMI no están directamente relacionados con la gestión de replicación. Ni uno ni otro replican automáticamente los agentes o recursos (pero tampoco lo impiden).

8. Los sistemas (middleware) de mensajería persistente…

a	…proporcionan transparencia de ubicación. Falso. Ya se ha comentado en la pregunta anterior que los MoM no proporcionan transparencia de ubicación.

b	…implantan la persistencia automáticamente al utilizar un servicio de nombres. Falso. La persistencia en la comunicación basada en mensajes no depende de ningún servicio externo de nombres. La persistencia depende de la capacidad que tengan los gestores de comunicación para mantener los mensajes en tránsito cuando el proceso destinatario todavía no esté listo para recibirlos.
c	…pueden ser desarrollados de manera sencilla utilizando una implantación basada en gestores ("broker-based"). Verdadero. Esos gestores son los que deben mantener los mensajes para proporcionar comunicación persistente.
d	…no pueden ser utilizados en comunicaciones asincrónicas. Falso. ZeroMQ es un ejemplo de MoM en el que se combina comunicación asincrónica con persistencia parcial.

SEMINARIOS

9. Considerando 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.');
```

Estas afirmaciones son ciertas si se asume que ningún error aborta su ejecución:

a	Este programa muestra en todas las iteraciones, entre otra información, el nombre del último fichero facilitado en la línea de órdenes. Verdadero. Este programa facilita un ejemplo de bucle en el que cada iteración utiliza un *callback* asincrónico. En esos casos, el *callback* no debe utilizar el contador de iteraciones de ese bucle. Al ser asincrónico, el *callback* iniciará su ejecución cuando el bucle ya haya terminado de iterar. Por ello, normalmente, estos *callbacks* tomarán el valor final del contador de iteraciones. En este ejemplo, mostrará el nombre del último fichero en todas las iteraciones.

b	Muestra el nombre y tamaño de cada uno de los ficheros recibidos en la línea de órdenes. Falso. Para proporcionar esa salida debería utilizarse una clausura. Sin ella, los tamaños mostrados serán correctos, pero los nombres no, como ya se ha descrito en el apartado anterior.
c	Muestra "We have processed N files" al final de su ejecución, siendo N el número de nombres recibidos como argumentos. Falso. Aunque este mensaje se escribe en la última línea del programa, esa línea formará parte del primer turno de ejecución. Ese turno finaliza antes de que se inicie la ejecución del primer *callback*. Por ello, será el primer mensaje mostrado en pantalla.
d	Descarta los dos primeros nombres de fichero pasados como argumentos en la línea de órdenes. Falso. No se descarta ningún nombre de fichero pasado como argumento. La línea 6 descarta la palabra "node" y el nombre del programa, pero no los argumentos pasados al programa.

10. Considerando el programa de la pregunta anterior…

a	Necesita múltiples turnos (de la cola de planificación del intérprete) para completar su ejecución, pues cada fichero utiliza su propio turno. Verdadero. Cada fichero es leído utilizando un *callback* asincrónico. Cada *callback* se ejecutará en un turno independiente.
b	Genera una excepción y aborta si algún error ocurre cuando intenta leer un fichero. Falso. Si ocurriera algún error, se comunicará en el primer argumento del *callback*. Su código gestiona ese primer parámetro escribiendo información sobre el error, pero sin generar ninguna excepción ni abortar el proceso.
c	El programa es incorrecto. Debería utilizar "var i=0" al inicializar la variable "i" para ser correcto. Falso. El valor inicial de esa variable es correcto. Los vectores se indexan a partir de cero por omisión. Como el bucle do-while incremente el valor de "i" en su primera sentencia, debe inicializar esa variable a -1 para acceder así a todas las componentes del vector "files".
d	Siempre muestra el mismo tamaño en todas las iteraciones. Necesitaríamos una clausura para evitar este comportamiento defectuoso. Falso. El tamaño del fichero se muestra correctamente en todas las iteraciones, pues solo depende del parámetro "data" y su valor se pasa en cada invocación. La clausura se necesitaría para pasar adecuadamente el nombre del fichero.

11. En los algoritmos de exclusión mutua vistos en el Seminario 2 se puede afirmar que…

a	El algoritmo con servidor central minimiza el número de mensajes necesarios. Verdadero. Es el que necesita menos mensajes de entre los presentados en el Seminario 2. Para ello utiliza interacciones punto a punto y gestiona los accesos con un coordinador. Así, basta con un mensaje para solicitar la entrada o comunicar la salida de la sección crítica y el permiso para entrar también se proporciona con un solo mensaje (la contestación a la solicitud). Otros algoritmos emplean difusiones (N o N-1 mensajes por difusión) para hacer lo mismo o propagan continuamente un token

	cuando no hay ningún solicitante activo.
b	El algoritmo de anillo unidireccional tiene un retraso de sincronización de 1 mensaje. Falso. Su retraso de sincronización es de 1 mensaje en el mejor caso, pero puede necesitar N-1 mensajes en el peor caso.
c	El retraso de sincronización del algoritmo con multidifusión y relojes lógicos es de 2N-2 mensajes. Falso. Su retraso de sincronización es de 1 mensaje.
d	La versión del algoritmo de multidifusión y *quórums* descrita en la presentación satisface las 3 condiciones de corrección para estos algoritmos. Falso. Es propenso a interbloqueos y, debido a esto, no puede cumplir la condición de vivacidad exigida a estos algoritmos.

12. Considerando este programa...

```
var ev = require('events');
var emitter = new ev.EventEmitter;
var num1 = 0;
var num2 = 0;
function emit_e1() { emitter.emit("e1") }
function emit_e2() { emitter.emit("e2") }
emitter.on("e1", function() {
   console.log( "Event e1 has happened " + ++num1 + " times.")});
emitter.on("e2", function() {
   console.log( "Event e2 has happened " + ++num2 + " times.")});
emitter.on("e1", function() {
   setTimeout( emit_e2, 3000 )});
emitter.on("e2", function() {
   setTimeout( emit_e2, 2000 )});
setTimeout( emit_e1, 2000 );
```

Estas afirmaciones son ciertas:

a	El evento “e1” ocurre solo una vez, 2 segundos después del inicio del proceso. Verdadero. La última línea del programa establece que el evento **e1** debe emitirse en 2”. Ese evento tiene dos *listeners*: uno escribe cuántas veces se ha dado **e1** y el otro programa el evento **e2** para dentro de 3”. Ninguno de los *listeners* de **e2** generará el evento **e1**. Por tanto, **e1** solo sucederá una vez, 2” tras iniciarse el proceso.
b	El evento “e2” nunca ocurre. Falso. El evento **e2** sucede múltiples veces. La primera, 5” después del inicio. Las restantes de manera periódica, con intervalos de 2”.
c	El periodo de “e2” es cinco segundos. Falso. Su periodo es 2”.
d	El periodo de “e1” es tres segundos. Falso. Solo sucede una vez.

13. En el programa de la pregunta anterior…

a	El primer evento “e2” ocurre cinco segundos después de haber iniciado el proceso. Verdadero. Se ha explicado en la pregunta anterior.

b	No se genera ningún evento en su ejecución, pues las llamadas a emit() son incorrectas. Falso. Los eventos se generan correctamente.
c	No se puede tener más de un *listener* para cada evento. Por tanto, el proceso aborta generando una excepción en la tercera llamada a emitter.on(). Falso. Los eventos pueden tener múltiples *listeners*.
d	Ninguno de sus eventos ocurre periódicamente. Falso. El evento **e2** sucede cada dos segundos.

14. El patrón de comunicación REQ-REP de ØMQ se considera sincrónico porque…

a	Sigue el patrón de interacción cliente/servidor y en ese patrón el cliente permanece bloqueado hasta que se reciba una respuesta. Falso. El emisor de la petición puede continuar con su ejecución. El socket REQ no lo bloquea. Las respuestas a esa solicitud son gestionadas mediante un *listener* para el evento **message**. Esa gestión es asincrónica.
b	Tanto REQ como REP son *sockets* bidireccionales, es decir, ambos pueden enviar y recibir mensajes. Falso. Son bidireccionales, pero la bidireccionalidad no implica sincronía (bloqueos).
c	La cola de envío del *socket* REQ tiene capacidad limitada. Solo puede mantener un mensaje. Falso. Las colas de envío tienen mayor capacidad. No están limitadas a un solo mensaje.
d	Los *sockets* REQ no pueden transmitir una solicitud mientras la respuesta anterior no se haya recibido. Los *sockets* REP no pueden enviar una respuesta antes de la solicitud. Verdadero. Esta es la característica que introduce cierto grado de sincronía en el patrón de comunicación REQ-REP de ZeroMQ.

15. Considerando estos dos programas…

```
// 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==2) client.end();
});
client.on('end', function() {
  console.log('client ' +
    'disconnected');
});
```

Las siguientes afirmaciones son ciertas:

a	El servidor termina tras enviar su primera respuesta al primer cliente. Falso. El código del servidor no incluye ninguna instrucción que fuerce su finalización (p.ej., un process.exit()). Cuando una conexión se cierre, se mostrará un mensaje comunicándolo, pero no se hace nada más. Por tanto, el servidor gestionará múltiples conexiones, respondiendo a cada mensaje recibido en ellas con otro mensaje que concatenará la palabra "World!" al contenido del mensaje recibido.

b	El cliente nunca termina. Verdadero. Nunca cierra su conexión abierta, ya que eso lo haría tras procesar el segundo mensaje recibido en ella, pero el servidor solo contesta una vez. Por tanto, ese segundo mensaje nunca llegará y el cliente nunca terminará.
c	El servidor sólo puede manejar una conexión. Falso. Los servidores que utilicen el módulo "net" pueden manejar múltiples conexiones.
d	Este cliente no puede conectarse al servidor. Falso. Tanto cliente como servidor utilizan el mismo número de puerto. Pueden comunicarse sin problemas si son ejecutados en un mismo ordenador.

16. Los algoritmos de elección de líder (vistos en el Seminario 2)…

a	...no tienen condiciones de seguridad. Falso. Para ser correctos, todos los algoritmos distribuidos deben respetar condiciones de seguridad y vivacidad (al menos una condición de cada tipo). La condición de seguridad para los algoritmos de elección de líder es que no debe haber más de un líder simultáneamente en el sistema.
b	...pueden estar indefinidamente seleccionando un proceso líder. Falso. La condición de vivacidad para estos algoritmos exige que la elección se efectúe en algún momento. Por tanto, no puede retrasarse indefinidamente.
c	...deben asegurar que un solo líder ha sido elegido. Verdadero. Esta es la condición de seguridad.
d	...deben respetar el orden causal. Falso. Esta condición no se exige a estos algoritmos. Fue presentada como una condición de corrección para los algoritmos de exclusión mutua, pero no para los de elección de líder.

17. Asuma que se va a desarrollar un servicio de exclusión mutua utilizando NodeJS y ØMQ, empleando el primer algoritmo explicado en el Seminario 2: el basado en un servidor central. La mejor opción (de entre las mostradas) para ello sería…

a	El servidor utilizará un *socket* DEALER y un *socket* ROUTER para equilibrar la carga entre sus clientes. Falso. En ese algoritmo los clientes utilizan un patrón petición-respuesta para implantar el protocolo de entrada a la sección crítica y un envío unidireccional para implantar el protocolo de salida. El patrón ROUTER-DEALER no está diseñado para gestionar esos dos tipos de interacción simultáneamente. ROUTER-DEALER se utiliza principalmente para que un *broker* propague y reparta mensajes entre un conjunto de servidores o trabajadores. Ese tipo de interacción no se utiliza en el algoritmo.

b	Cada cliente utilizará un *socket* DEALER para interactuar con el servidor. Verdadero. Los sockets DEALER pueden implantar tanto el patrón petición-respuesta del protocolo de entrada como el de envío unidireccional del protocolo de salida.
c	Cada cliente utilizará un *socket* REP para interactuar con el servidor. Falso. Los sockets REP no pueden utilizarse para iniciar un protocolo de entrada a la sección crítica. Recuérdese que el socket REP no puede iniciar un envío hasta que un mensaje de petición previo haya sido recibido por ese socket.
d	Cada cliente utilizará un *socket* SUB para interactuar con el servidor. Falso. Los sockets SUB solo pueden recibir mensajes. No pueden enviarlos. Por tanto, no pueden implantar ninguno de los dos protocolos que hemos explicado en la descripción del primer apartado.

18. Asuma que se va a desarrollar un servicio de exclusión mutua utilizando NodeJS y ØMQ, empleando el 2º algoritmo explicado en el Seminario 2: el de anillo (virtual) unidireccional. La mejor opción (de entre las mostradas) para ello sería…:

a	Utilizar algún algoritmo de elección de líder para seleccionar un proceso coordinador. Falso. El algoritmo de anillo es simétrico. Todos los procesos se comportan igual. Por tanto, no se necesita ningún coordinador en este algoritmo.
b	Todos los procesos tienen el mismo rol y necesitan un *socket* REP para enviar mensajes y un *socket* REQ para recibirlos. Falso. Tanto REP como REQ impiden dos envíos consecutivos mientras no haya alguna recepción intermedia. Con ello, no pueden implantar un patrón de comunicación unidireccional como el que se necesita en este algoritmo.
c	Todos los procesos tienen el mismo rol y necesitan un *socket* PUSH para enviar el *token* y un *socket* PULL para recibirlo. Verdadero. Estos sockets son unidireccionales y cumplen los requisitos de comunicación de este algoritmo.
d	Todos los procesos tienen el mismo rol y necesitan un *socket* PUB para enviar el *token* y un *socket* DEALER para recibirlo. Falso. Aunque un socket PUB pueda emplearse para enviar mensajes y un socket DEALER pueda recibirlos, el socket PUB difunde aquello que emite. Ese tipo de envío no resulta apropiado para implantar este algoritmo.

19. Considerando estos programas...

```
//client.js
var zmq=require('zmq');
var rq=zmq.socket('dealer');
rq.connect('tcp://127.0.0.1:8888');
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('dealer');
rp.bindSync('tcp://127.0.0.1:8888');
rp.on('message', function(msg) {
  var j = parseInt(msg);
  rp.send([msg,(j*3).toString()]);
});
```

Las siguientes afirmaciones son ciertas:

a	El cliente y el servidor intercambian mensajes sincrónicamente, pues ambos siguen un patrón petición/respuesta. Falso. Ambos utilizan sockets DEALER y ninguno de ellos necesita bloquearse en

	ninguna acción relacionada con mensajes. De hecho, el cliente puede llegar a enviar sus 100 solicitudes antes de haber recibido la primera respuesta.
b	El servidor retorna un mensaje con 2 segmentos al cliente. El segundo segmento contiene un valor 3 veces superior al del primer segmento. Verdadero. Esa es la funcionalidad del *listener* para el evento 'message' en el servidor.
c	El cliente y el servidor pueden ejecutarse en diferentes ordenadores. Interactúan sin problemas en ese caso. Falso. Ambos utilizan 127.0.0.1 como dirección IP en sus llamadas a bindSync() o connect(). Eso implica que ambos deben ejecutarse en el mismo ordenador para poder comunicarse.
d	El cliente no envía ningún mensaje pues la sentencia `''+i` genera una excepción y el proceso aborta en ese punto. Falso. Esa sentencia no genera ningún error o excepción. Su objetivo es tratar el valor de **i** como una cadena.

20. Considere cuál de las siguientes variaciones generaría nuevos programas con el mismo comportamiento observado en la pregunta 19 (A--> B significa que la sentencia A es reemplazada por la sentencia B).

a	El *socket* 'rq' será de tipo PULL y el *socket* 'rp' de tipo PUSH. Falso. El programa necesita comunicación bidireccional. PUSH y PULL son unidireccionales.
b	El *socket* 'rq' será de tipo PUSH y el *socket* 'rp' de tipo PULL. Falso. El programa necesita comunicación bidireccional. PUSH y PULL son unidireccionales.
c	Cliente: rq.connect('tcp://127.0.0.1:8888'); --> rq.bindSync('tcp://*:8888'); Servidor: rp.bindSync('tcp://127.0.0.1:8888'); --> rp.connect('tcp://127.0.0.1:8888'); Verdadero. Esta es la mejor opción entre todas las presentadas. El comportamiento será idéntico al de la versión original si se asume que solo habrá un cliente y un servidor.
d	El *socket* 'rq' será de tipo REP y el *socket* 'rp' de tipo REQ. Falso. Un socket REP no puede ser utilizado para iniciar la interacción con otro proceso. Por tanto, el cliente no puede usar un REP para interactuar con el servidor.