Este examen contiene 20 cuestiones de opción múltiple. En cada una de ellas solo una de sus respuestas es correcta. Las contestaciones deben presentarse en una hoja entregada aparte. Las respuestas correctas aportan 0.5 puntos a la nota del parcial mientras que las incorrectas restan una décima.

En la hoja de respuestas debes rellenar la casilla elegida cuidadosamente. Utiliza un lápiz o un bolígrafo (negro o azul oscuro). Se puede usar "Tipp-Ex" o algún corrector similar. En ese caso, NO INTENTES DIBUJAR DE NUEVO LA CASILLA QUE HAYAS BORRADO.

TEORÍA

1. Los sistemas distribuidos...

A	...están formados generalmente por múltiples agentes que se ejecutan concurrentemente. Esos agentes mantienen cierto estado independiente. Para tener un sistema distribuido se necesita tener múltiples procesos (o agentes) que colaboren y se ejecuten en ordenadores diferentes facilitando una imagen de sistema único. A pesar de esta colaboración, cada proceso puede mantener algún estado local desconocido por (e inaccesible para) los demás procesos, es decir, cierto estado independiente.
B	...no necesitan ningún mecanismo de comunicación entre ordenadores. Estos agentes deben ejecutarse en al menos dos ordenadores distintos para tener un sistema distribuido. Por ello, estos sistemas necesitan algún mecanismo de comunicación (normalmente una red) entre ordenadores.
C	...siempre utilizan interacciones cliente-servidor. Hemos visto en el Tema 1 que hay otros tipos de interacción entre agentes; por ejemplo, interacciones *peer-to-peer*. Por tanto, no todas las interacciones entre agentes deben establecerse entre clientes y servidores siguiendo un patrón petición/respuesta.
D	...nunca tendrán condiciones de carrera. No. Las condiciones de carrera pueden darse en cualquier sistema concurrente. Los sistemas distribuidos (por ser también concurrentes) no pueden garantizar, por sí mismos, que las condiciones de carrera nunca se den.
E	Todas las anteriores.
F	Ninguna de las anteriores.

2. La computación en la nube (*Cloud computing*)...

A	...es uno de los paradigmas actuales de prestación de servicios en la computación distribuida.

B	...tiene como objetivo la prestación de servicios de cómputo de una manera escalable y eficiente.
C	...sigue un modelo de “pago por uso”.
D	...usa generalmente infraestructuras virtualizadas.
E	Todas las anteriores. La computación en la nube está orientada a facilitar los mecanismos necesarios para desarrollar, desplegar y proveer servicios software de una manera eficiente y escalable. Normalmente estos servicios se facilitan siguiendo un modelo de “pago por uso” y los mecanismos de virtualización se utilizan para aprovechar tanto como sea posible los recursos de los equipos, es decir, de la infraestructura.
F	Ninguna de las anteriores.

3. En un sistema distribuido, la interacción entre sus agentes...

A	…nunca debe llevarse a cabo. Si interactuaran, el sistema sería concurrente en lugar de distribuido. No. Los agentes deben interactuar. De otra manera, el sistema resultante sería un conjunto de procesos aislados e independientes en lugar de un sistema capaz de combinar la funcionalidad de múltiples componentes básicos para resolver de una manera eficiente y robusta algunos problemas y tareas complejos.
B	…se realiza intercambiando mensajes o compartiendo memoria. Sí. Estos son los mecanismos de comunicación habituales en sistemas concurrentes. Los sistemas distribuidos usan ambos mecanismos.
C	…se realiza sin compartir memoria. La compartición de memoria está prohibida en los sistemas distribuidos. No. Cuando varios agentes de un sistema distribuido están ubicados en un mismo ordenador pueden compartir memoria. No hay ninguna regla que limite o prohíba el uso de memoria compartida en una aplicación distribuida cuando esa memoria pueda compartirse entre algunos de sus agentes.
D	…se consigue cuando todos los agentes residan en un mismo ordenador. No. No tiene sentido obligar a que todos los agentes de un sistema distribuido estén desplegados en un único ordenador. El fallo de ese ordenador conllevaría el fallo completo de esa aplicación o sistema distribuido.
E	Todas las anteriores.
F	Ninguna de las anteriores.

4. El modelo de programación guarda / acción ...

A	Se sigue en la programación multi-hilo, donde las secciones críticas equivalen a las guardas y los hilos ("threads") equivalen a las acciones. No. Una guarda es equivalente a una precondición y una sección crítica será siempre un fragmento de código. Por tanto, no pueden darse esas equivalencias. Además un hilo es una entidad dinámica (conlleva ejecución de instrucciones) y una acción podría facilitar la secuencia de instrucciones que debía ejecutar ese hilo pero, aun así, no son conceptos equivalentes (el hilo es dinámico y la acción es estática pues este último concepto se suele utilizar para dar estructura a los algoritmos).
B	Se sigue en la programación asincrónica (o dirigida por eventos), donde los eventos equivalen a las acciones y las funciones "callback" de los eventos equivalen a las guardas. Véase la explicación del apartado D. Los eventos son guardas y las funciones son acciones.
C	Se sigue en la programación multi-hilo, donde los hilos ("threads") equivalen a guardas que se activan y se suspenden, y las operaciones en las secciones críticas equivalen a las acciones. Véase la explicación del apartado A. Estos cuatro conceptos no son equivalentes (ni guardan una relación estrecha).
D	Se sigue en la programación asincrónica (o dirigida por eventos), donde los eventos equivalen a las guardas y las funciones "callback" de los eventos equivalen a las acciones. En un algoritmo una acción es un bloque de sentencias que tiene sentido por sí mismo ya que define alguna operación útil. En el modelo guarda / acción, esas acciones están asociadas a sus precondiciones (a las que llamamos "guardas" en este modelo). Una vez la precondición o guarda se cumpla, la acción estará habilitada y podrá ejecutarse. La programación asincrónica se corresponde bien con el modelo guarda / acción pues los "listeners" de los eventos definen bloques de sentencias equivalentes a acciones (que se ejecutan de manera atómica ya que no pueden ser interrumpidas por otras acciones habilitadas) y un evento puede considerarse equivalente a una guarda que se satisfará cada vez que ocurra el evento.
E	Todas las anteriores.
F	Ninguna de las anteriores.

5. Algunas características relevantes de los modelos de sistemas distribuidos:

A	Se centran en las principales propiedades del comportamiento del sistema.
B	Facilitan una buena herramienta para razonar sobre la corrección de los algoritmos y protocolos basados en ellos.
C	Su alto nivel de abstracción.
D	Facilitan una base para discutir sobre la imposibilidad de resolver problemas en ciertos sistemas distribuidos (p.ej., el consenso en sistemas asincrónicos).
E	Todas las anteriores. Un modelo debe centrarse en las propiedades esenciales del elemento o sistema que represente (A) y esto conlleva la utilización de un alto nivel de abstracción (C), descartando muchos detalles que se considerarían irrelevantes. El objetivo de los modelos es facilitar una imagen de un sistema que permita diseñar algoritmos capaces de resolver problemas en el sistema real así como discutir sobre su corrección antes de que sean implantados en un programa (B). Esas discusiones pueden también orientarse a evaluar la imposibilidad de resolver algunos problemas en los sistemas modelados (D). Estos resultados de imposibilidad suelen basarse en las condiciones que definen el modelo de sistema que se haya supuesto.
F	Ninguna de las anteriores.

6. Los elementos a considerar en un modelo de sistema pueden ser...

A	La arquitectura del equipo, el sistema operativo, el middleware y el lenguaje de programación. Estos elementos suelen ser irrelevantes al diseñar un algoritmo. Los modelos de sistema facilitarán una imagen abstracta que resulte útil para generar un algoritmo que resuelva algunos de los problemas de en un sistema real. Por tanto, esta lista no incluye los elementos a considerar en un modelo de sistema.
B	Procesos, eventos, aspectos de comunicación, fallos, gestión del tiempo y nivel de sincronía. Estos son los elementos que definen un modelo de sistema: qué tipo de **procesos** se están suponiendo, qué **eventos** debemos considerar en el algoritmo que estemos construyendo (internos, de entrada, de salida...) ya que con ellos definiremos la interfaz del componente que implante este algoritmo, qué mecanismos de **comunicación** serán utilizados por los agentes, qué tipos de **fallos** podrán ocurrir en el sistema y qué consecuencias podrá tener cada clase de fallo, cómo gestionan los agentes el transcurso del **tiempo** y la sincronía, qué nivel de **sincronía** se supone en la interacción entre agentes...
C	Nivel físico, nivel de enlace, nivel de red, nivel de transporte y nivel de aplicación. No. Estos son los niveles que definen la arquitectura estándar en un sistema de comunicaciones de red.
D	Sistema gestor de bases de datos, middleware e interfaz de usuario. No. Estos son ejemplos de los elementos que permiten implantar una arquitectura cliente/servidor multinivel.
E	Todas las anteriores.
F	Ninguna de las anteriores.

7. En la programación de sistemas distribuidos, el uso del middleware es aconsejable porque...

A	Introduce múltiples transparencias, ocultando detalles de bajo nivel y ofreciendo una interfaz uniforme.
B	Tiene una implantación sencilla, y poca complejidad en los elementos manejados.
C	Proporciona una operativa estandarizada, comprensible y bien definida.
D	Facilita la interoperabilidad, la interacción con productos de terceras partes.
E	Todas las anteriores. Esas son cuatro de las ventajas introducidas por un nivel middleware en una arquitectura de sistemas distribuidos.
F	Ninguna de las anteriores.

8. Los problemas que encontramos en los sistemas distribuidos orientados a objetos son:

A	Todos los objetos parecen ser locales y esto puede generar largos intervalos para completar su invocación en caso de que sean remotos. Es un problema puesto que el tiempo necesario para realizar una invocación a objeto será difícil de predecir en un entorno como este.
B	Los objetos mantienen estado y ese estado se compartirá entre los agentes que invoquen sus métodos. Esto puede provocar problemas de consistencia. El estado compartido puede provocar condiciones de carrera y las condiciones de carrera generarán inconsistencias de estado. Además, si el objeto se llegara a replicar sus métodos serían invocados por varios agentes y esos agentes podrían ser atendidos por réplicas diferentes. Las modificaciones generadas en esas llamadas podrían, de nuevo, generar inconsistencias entre las réplicas.
C	Su estado compartido necesita mecanismos de control de concurrencia. Esto puede ocasionar bloqueos, evitando que los sistemas sean escalables. El estado compartido define secciones críticas. Esas secciones críticas deben ser protegidas mediante protocolos de entrada y protocolos de salida. Esos protocolos suelen implantarse utilizando mecanismos de control de concurrencia (p.ej., locks, semáforos, monitores...) y esos mecanismos bloquean, cuando es necesario, a los agentes en ejecución. Por tanto, esto puede conllevar problemas graves cuando deseemos desarrollar servicios escalables.
D	Sus mecanismos de invocación facilitan una alta transparencia de ubicación. Esto exige protocolos de recuperación complejos para gestionar los fallos. Cuando un servidor falle será reemplazado por alguna de sus réplicas, manteniendo la transparencia de ubicación. Esto exige que el protocolo de recuperación deba reaccionar rápidamente. Los protocolos que requieran una reacción y recuperación rápidas no son sencillos y dependen del modelo de replicación utilizado (activo, pasivo o alguna variante intermedia).
E	Todas las anteriores.
F	Ninguna de las anteriores.

SEMINARIOS

9. Considérese el siguiente programa (incompleto) escrito en Node:

```
function logaritmo(x,b) { return Math.log(x)/Math.log(b) }
function logBase ... // a completar
log2 = logBase(2);
log8 = logBase(8);
console.log("Logarithm base 2 of 1024 = " + log2(1024));
console.log("Logarithm base 8 of 4096 = " + log8(4096));
```

¿Cuál implementación de la función `logBase` sería correcta?

A	`function logBase(b) { return logaritmo(x,b) }`

B	`function logBase(b) {`
	`  return function(x) { return logaritmo(x,b) }`
	`}`
	logBase() debe ser una función que devuelva otra función como su resultado (ya que log2() y log8() son funciones en este ejemplo). Además, su función retornada debe recordar el argumento utilizado en la llamada a logBase() para emplearlo como la base del logaritmo a calcular. Adicionalmente, su parámetro debe ser un número ("x") del que se pide ese logaritmo. Este apartado B cumple todos esos requisitos.
	
	El apartado A no retorna una función sino un valor.
	
	El apartado C retorna una función que no devuelve nada.
	
	El apartado D retorna una función que realiza algunos cálculos pero en la que se han interpretado incorrectamente los parámetros de logBase(), utilizándolos al revés de lo que se debía.

C	`function logBase(x) {`
	`  return function(b) { logaritmo(x,b) }`
	`}`

D	`function logBase(x) {`
	`  return function(b) { return logaritmo(x,b) }`
	`}`

E	Todas las anteriores.
F	Ninguna de las anteriores.

10. Considérese el siguiente programa escrito en Node:

```
var fruits = ["Banana", "Orange", "Lemon", "Apple"];
var numbers = [7, 3, "Cloud", 9];
var funcs = [function(x) {return 2*numbers[x]},
             function(x) {return fruits[x]}];
var s = "";
for (var i=0; i<2; i++)
  for (var j=0; j<5; j=j+2)
     s += funcs[i](j) + ", ";
console.log(s);
```

Al ejecutarlo, la salida que se mostrará en consola será:

A	14, 6, NaN, Banana, Orange, Lemon,
B	14, NaN, NaN, Banana, Lemon, undefined, Este programa contiene dos bucles anidados. El externo (con la variable "i") realiza dos iteraciones, con los valores 0 y 1 para "i". El interno (con la variable "j") realiza tres iteraciones, con los valores 0, 2 y 4 para "j". En cada iteración su única instrucción concatena a la cadena "s" (inicialmente vacía) el resultado de llamar a la función "funcs[i]" pasando como argumento el valor de "j". La función funcs[0] retorna como resultado el doble del valor contenido en numbers[j]. Ese valor debe ser un número (de otra manera, el resultado del operador "*" es NaN). La función funcs[1] retorna la componente "j" en el vector fruits[]. Por tanto, mientras i valga 0, los valores retornados en la llamada a funcs[0] serán: el doble de 7 (14), el doble de "Cloud" (NaN) y el doble de "undefined" (NaN). Cuando valga 1 los valores retornados serán "Banana" (es decir, fruits[0]), "Lemon" (fruits[2]) y undefined (fruits[4]). Con ello la secuencia obtenida en esta ejecución es: "14, NaN, NaN, Banana, Lemon, undefined,"
C	14, 2Cloud, undefined, Banana, Lemon, undefined,
D	Banana, 14, Lemon, NaN, undefined,
E	No se mostraría nada, salvo un mensaje de error indicando que el array `numbers` está mal definido, por contener valores de diferentes tipos.
F	Ninguna de las anteriores.

NOTA: En el tercer elemento de esa lista se estaba multiplicando 2 por *undefined* pues se está intentando acceder a una componente inexistente de un vector (numbers[4]). El valor resultante es NaN pero también se podría pensar que era *undefined*. Debido a ello, también se admitirá F (ninguna de las anteriores) como respuesta correcta.

11. Considérese la siguiente función escrita en Node:

```
function f(x,y) {
   x = x || 'naranja'; y = y || 98;
   console.log('x='+x+' y='+y);
}
```

Indique cuál sería la salida que se mostrará en consola si se ejecuta:

```
f(36);          f(undefined,'manzana');    f(45,0,67);
```

A	x=36 y=98	x=undefined y=manzana	x=45 y=0
B	x=36 y=98	x=naranja y=manzana	x=45 y=0
C	x=36 y=98	x=naranja y=manzana	x=45 y=98
	Esta cuestión versa sobre el operador lógico '||' (OR) y la utilización de argumentos en las llamadas a función. El operador '||' retorna su operando izquierdo cuando este no pueda considerarse falso (es decir, la constante Booleana **false**) y retornará su operando derecho en otro caso. Sin embargo, en JavaScript hay varios valores falsos. Cuando consideramos números, 0 es **false** y todos los demás valores son **true**. Por otra parte, *undefined* también es **false**. Por tanto, en la llamada f(36) estamos utilizando solo un argumento para f y f tiene dos parámetros. Esto implica que el parámetro "y" recibirá un valor *undefined*. Debido a ello, "y" tomará en este caso el valor 98. Así, en esta primera llamada tendremos **x=36** e **y=98**. En la llamada f(undefined, 'manzana'), la variable "x" obtendrá el valor **naranja** ya que este último es el operando derecho en la instrucción "x = x || 'naranja". Por tanto, tendremos **x=naranja** e **y=manzana**. Finalmente, en la llamada f(45,0,67), el argumento 67 (el tercero) será descartado y la "y" obtendrá el valor 98 pues 0 es equivalente a **false**. Con ello, **x=45** e **y=98**. El apartado C es el único que presenta valores correctos para estas tres llamadas.		
D	x=36 y=36	x=naranja y=manzana	x=45 y=67
E	No se mostraría nada, salvo mensajes de error pues hay invocaciones incorrectas (por su número de argumentos) de la función f.		
F	Ninguna de las anteriores.		

12. Considérese el siguiente programa escrito en Node:

```
var eve = new (require('events')).EventEmitter;
var s = "print";
var n = 0;
var handler = setInterval( function(){eve.emit(s);}, 1000 );
eve.on(s, function() {
   if ( n < 2 ) console.log("Event", s, ++n, "times.");
   else clearInterval(handler);
});
```

Si se ejecuta este programa indique, en relación a la salida que se mostrará en consola y al tiempo de ejecución, cuál de las siguientes opciones es la correcta:

A	Event print 1 times. Event print 2 times. Este programa genera un evento "print" cada segundo (línea 4). En el *listener* para "print" se comprueba si n es menor que 2 (inicialmente es cero) y, de ser así, se imprime un mensaje (empezando con n=1 ya que esta variable se preincrementa en los argumentos de la instrucción console.log). En otro caso se elimina el intervalo de generación del evento "print". Cuando eso ocurra, el proceso finalizará pues no hay más eventos que manejar ni otros turnos pendientes. Con ello, cuando se inicia este proceso, su línea 4 programa la generación del evento 4 cada segundo. Un segundo después se imprime el mensaje "Event print 1 times". De nuevo, un segundo después se imprime "Event print 2 times". En el tercer segundo, el *listener* comprueba si n es menor que 2 pero ahora ya es 2. Por tanto, se ejecuta el else y se cancelan los intervalos. Como resultado de ello, el proceso finaliza tres segundos después de su inicio habiendo impreso dos mensajes.	Y concluiría después de 3 segundos.
B	Event print 1 times. Event print 2 times. Event print 3 times.	Y concluiría después de 4 segundos.
C	Event print 1 times. Event print 2 times. …	Y no concluiría. Cada segundo, mostraría una nueva línea con el número incrementado en una unidad.
D	Event print 0 times. Event print 1 times. …	Y no concluiría. Cada 10 segundos, mostraría una nueva línea con el número incrementado en una unidad.
E	No se mostraría nada, porque no está bien definido el objeto listener.	Y no concluiría, pues se emite cíclicamente el evento "print".
F	Ninguna de las anteriores.	

13. Considerando el programa siguiente...

```
var http = require('http');
var fs = require('fs');
http.createServer(function(request,response) {
  fs.readdir(__dirname, function(err,data) {
    if (err) {
      response.writeHead(404, {'Content-Type':'text/plain'});
      response.end('Unable to read directory '+__dirname);
    } else {
      response.writeHead(200, {'Content-Type':'text/plain'});
      response.write('Directory: ' + __dirname + '\n');
      response.end(data.toString());
    }
  })
}).listen('1337');
```

Seleccione las opciones correctas:

A	Este programa genera una excepción y aborta en caso de no poder leer el contenido del directorio actual. No. Si encontrara un error tratando de leer ese directorio, el *callback* para la llamada a readdir() recibiría un objeto en su primer argumento y el servidor web retornaría una respuesta a la solicitud del cliente, sin generar ninguna excepción ni abortar.
B	Este programa es un servidor web que responde con el nombre y lista de ficheros en el directorio actual. Esta es la descripción correcta de las tareas desarrolladas en este programa. La operación http.createServer() facilita la base para escribir un servidor HTTP. Independientemente de qué peticiones se hayan recibido, este proceso siempre contesta con una respuesta que contiene el nombre y la lista de ficheros contenidos en el directorio donde haya sido iniciado. El nombre de ese directorio se mantiene en la variable __dirname.
C	Este programa no funciona porque no ha declarado la variable "__dirname" y no ha importado el módulo 'process' donde está definida. No. "__dirname" es una variable que ya está declarada por omisión. No se necesita importar ningún módulo ni declararla para poderla usar.
D	Este programa no funciona porque 'data' es un vector de nombres de fichero y los vectores no pueden ser transformados en cadenas. Falso. Un vector de cadenas (pues los nombres de fichero son cadenas) puede ser convertido en cadena sin mayor problema.
E	Todas las anteriores.
F	Ninguna de las anteriores.

14. Algunos problemas del algoritmo de exclusión mutua con servidor central:

A	No cumple su condición de vivacidad. Todos los algoritmos distribuidos correctos deben cumplir sus condiciones de vivacidad y seguridad. Este algoritmo respeta su condición de vivacidad, esto es, asegura que todos los solicitantes lograrán acceder a la sección crítica en algún momento.
B	No cumple su condición de seguridad. Todos los algoritmos distribuidos correctos deben cumplir sus condiciones de vivacidad y seguridad. Este algoritmo respeta su condición de seguridad, esto es, asegura que nunca habrá más de un proceso simultáneamente en la sección crítica.
C	Necesita más mensajes que los demás algoritmos vistos en el Seminario 2 para resolver el problema de exclusión mutua. No. De hecho es uno de los algoritmos que necesita menos mensajes para gestionar una sección crítica.
D	Es frágil en situaciones de fallo. El servidor central es un punto único de fallo. Cierto. Si el servidor central fallara ninguno de los participantes podría superar su protocolo de entrada a la sección crítica, pues quedarían esperando el mensaje de autorización de entrada que debería enviarles el servidor central.
E	Todas las anteriores.
F	Ninguna de las anteriores.

15. Los algoritmos de elección de líder...

A	...son un subconjunto de los algoritmos de consenso. Sí. Para elegir un líder los procesos deben alcanzar un consenso sobre cuál es el mejor candidato.
B	...necesitan que todos los procesos tengan un identificador distinto. Sí. De otra manera sería imposible elegir a alguno de ellos pues esa decisión está basada en la comparación de sus identificadores.
C	...usan un criterio determinista para elegir al líder. Sí, y ese criterio debe ser conocido por todos los procesos participantes.
D	...exigen que se elija solo a un proceso. Sí. El líder debe ser único.
E	Todas las anteriores.
F	Ninguna de las anteriores.

16. Supongamos que se necesita implantar un servicio de chat utilizando node.js y ØMQ. El servidor difunde los mensajes de los usuarios y nunca debe suspenderse tratando de enviar un mensaje (de cualquier tipo). Los programas clientes envían los mensajes de los usuarios al servidor, esperan los mensajes reenviados por el servidor e informan al servidor cuando un usuario se incorpora o abandona el sistema. Para implantar este servicio de chat...

A	El servidor debe usar un socket PULL y otro REP para interactuar con los clientes. No. Los mensajes de los usuarios deben ser difundidos. Solo los sockets PUB pueden difundir un mensaje utilizando una única llamada a send().

B	El servidor debe usar un socket SUB y otro REQ para interactuar con los clientes. No. Los mensajes de los usuarios deben ser difundidos. Solo los sockets PUB pueden difundir un mensaje utilizando una única llamada a send().
C	El servidor debe usar un socket PUB y otro PULL para interactuar con los clientes. Sí. Los mensajes de los usuarios deben ser difundidos por el servidor y solo un socket PUB puede realizar esa difusión con una única llamada a send(). Además, se necesitará otro socket para aceptar y procesar los mensajes enviados por los procesos clientes para decirle al servidor que un usuario se ha incorporado o ha abandonado la aplicación. Ese segundo socket debe admitir recepciones de mensajes y un socket PULL puede gestionar esto de una manera asincrónica, sin bloquear nunca al servidor en su procesamiento de mensajes.
D	El servidor debe usar un socket REP y otro SUB para interactuar con los clientes. No. Los mensajes de los usuarios deben ser difundidos. Solo los sockets PUB pueden difundir un mensaje utilizando una única llamada a send().
E	Todas las anteriores.
F	Ninguna de las anteriores.

17. Supongamos que hemos implantado un servicio soportado por múltiples (p.ej., 10) procesos servidores ubicados en ordenadores diferentes. Esos servidores utilizan sockets REP y sus clientes usan sockets REQ. Si construimos un broker con un socket ROUTER como *front-end* y un socket DEALER como *back-end* (y para ambos se realiza un bind()), entonces...

A	Los clientes no necesitan conocer cuántos procesos servidores hay. Sí. El broker es el único proceso que interactúa directamente con los servidores. Por tanto, los procesos clientes no necesitan ninguna información sobre los procesos servidores.
B	Los clientes no necesitan conocer las direcciones y puertos de cada proceso servidor. Sí. El broker es el único proceso que interactúa directamente con los servidores. Por tanto, los procesos clientes no necesitan ninguna información sobre los procesos servidores.
C	La cantidad de procesos servidores puede variar dinámicamente. Ellos deben conectarse al socket *back-end* para que el broker pueda utilizarlos. Sí. Podemos modificar la cantidad de servidores de una manera transparente. Solo necesitan conectarse al socket DEALER.
D	El broker no debe modificar ningún segmento de los mensajes para propagarlos del *front-end* al *back-end* y del *back-end* al *front-end*. Sí. Ambos sockets (ROUTER y DEALER) no necesitan preocuparse por el contenido de los mensajes. Ninguno de los segmentos en los mensajes necesita ser modificado (ni añadido ni eliminado). Con esta estrategia los sockets DEALER distribuyen de manera circular los mensajes de petición entre todos los servidores conectados.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Para contestar a las siguientes 2 cuestiones (nº 18 y 19), considérense los siguientes programas Node con ØMQ. Un servidor (*server.js*):

```
var zmq = require('zmq')
var rep = zmq.socket('rep')
rep.bindSync('tcp://127.0.0.1:'+process.argv[2])
var n = 0
rep.on('message', function(msg) {
  console.log('Request: ' + msg)
  rep.send('World ' + ++n)
})
```

Y un cliente (*client.js*):

```
var zmq = require('zmq')
var req = zmq.socket('req')
req.connect('tcp://127.0.0.1:'+process.argv[2])
req.connect('tcp://127.0.0.1:'+process.argv[3])
var n = 0
setInterval( function() { req.send('Hello ' + ++n) }, 100 )
req.on('message', function(msg) {
  console.log('Response: ' + msg)
})
```

---

18. Considérense los anteriores programas Node con ØMQ (*server.js* y *client.js*). Si, en 3 terminales, se ejecutaran 2 servidores y 1 cliente mediante:

```
node server 8001
node server 8002
node client 8001 8002
```

Las primeras líneas que se mostrarán en las terminales de los servidores serán:

<table>
<tr><td>A</td><td>En una terminal:
<code>Request: Hello 1</code>
<code>Request: Hello 3</code>
Y en la otra terminal:
<code>Request: Hello 2</code>
<code>Request: Hello 4</code>
Ya que el socket REQ del cliente está conectado a los REP de los servidores, ese REQ distribuye de manera circular sus mensajes entre ellos. Esto implica que su primer mensaje irá al primer servidor, la segunda petición al segundo servidor, la tercera petición al primer servidor y así sucesivamente.
Por ello, los mensajes se estarán imprimiendo tal como se muestra en este apartado pues cada petición incrementa el mismo contador local (n) y la primera petición incluyó el valor 1 en su mensaje.</td></tr>
<tr><td>B</td><td>En ambas terminales:
<code>Request: Hello 1</code>
<code>Request: Hello 2</code>
No. Ver la explicación en el apartado A.</td></tr>
<tr><td>C</td><td>En ambas terminales (siendo x, y, z… números tales que x < y < z < …):
<code>Request: Hello x</code>
<code>Request: Hello y</code>
<code>Request: Hello z</code>
<code>...</code>
No. Ver la explicación en el apartado A.</td></tr>
<tr><td>D</td><td>En una terminal:
<code>Request: Hello 1</code>
<code>Request: Hello 2</code>
Y en la otra terminal:
<code>Request: Hello 3</code>
<code>Request: Hello 4</code>
No. Ver la explicación en el apartado A.</td></tr>
<tr><td>E</td><td>No se mostraría nada, dado que el cliente no sabría a cuál de los servidores enviar sus peticiones. (Para un funcionamiento correcto, el cliente debería conectarse a un socket ROUTER).
No. Ver la explicación en el apartado A.</td></tr>
<tr><td>F</td><td>Ninguna de las anteriores.</td></tr>
</table>

19. Considérense los mismos programas, y el mismo escenario de ejecución, de la cuestión anterior. Las primeras líneas que se mostrarán en la terminal del cliente serán:

A	`Response: World 1`
	`Response: World 2`
	`Response: World 3`
	`Response: World 4`
	No. Ver la explicación en el apartado B.

B	`Response: World 1`
	`Response: World 1`
	`Response: World 2`
	`Response: World 2`
	Sí. Tal como se ha explicado en la cuestión 18, el cliente envía cada petición a un servidor diferente, equilibrando la carga de ambos. Cada servidor utiliza un contador local para etiquetar su respuesta. Por tanto, la primera respuesta recibida por el cliente es la primera enviada por el primer servidor, la segunda es la primera respuesta del segundo servidor, la tercera es la segunda respuesta del primer servidor y así sucesivamente.

C	`Response: World 1`
	`Response: World 3`
	`Response: World 5`
	`Response: World 7`
	No. Ver la explicación en el apartado B.

D	`Response: World 1`
	`Response: World 3`
	`Response: World 2`
	`Response: World 4`
	No. Ver la explicación en el apartado B.

E	No se mostraría nada, dado que, como el cliente no sabría a cuál de los servidores enviar sus peticiones, ninguno de los servidores podría enviar respuestas. No. Ver la explicación en el apartado B.
F	Ninguna de las anteriores.

20. Considérense los siguientes programas Node con ØMQ. Un publicador:

```
var zmq = require('zmq')
var pub = zmq.socket('pub').bindSync('tcp://*:5555')
var count = 0
setInterval(function() {
  pub.send('PRG ' + count++)
  pub.send('TSR ' + count++)
}, 1000)
```

Y un suscriptor:

```
var zmq = require('zmq')
var sub = zmq.socket('sub')
sub.connect('tcp://localhost:5555')
sub.subscribe('TSR')
sub.on('message', function(msg) {
  console.log('Received: ' + msg)
})
```

Si se ejecutara, en primer lugar, el publicador y, tres segundos después, el

suscriptor. Las primeras líneas de la salida que se mostrarán en la terminal del suscriptor serán:

A	`Received: TSR 1`
	`Received: TSR 2`
	`Received: TSR 3`
	…
	No. Ver la explicación en el apartado D.

B	`Received: TSR 1`
	`Received: TSR 3`
	`Received: TSR 5`
	…
	No. Ver la explicación en el apartado D.

C	`Received: PRG 4`
	`Received: TSR 5`
	`Received: PRG 6`
	…
	No. Ver la explicación en el apartado D.

D	`Received: TSR 5`
	`Received: TSR 7`
	`Received: TSR 9`
	…
	Sí. Debe considerarse que el publicador envía dos mensajes por segundo, cada uno con un prefijo distinto, pero utilizando el mismo contador que incrementamos en cada envío. Por tanto, los mensajes enviados por el publicador son...
	En el segundo 1: PRG 0, TSR 1
	En el segundo 2: PRG 2, TSR 3
	En el segundo 3: PRG 4, TSR 5
	En el segundo 4: PRG 6, TSR 7
	
	El suscriptor solo recibirá los mensajes con el prefijo TSR. Es iniciado en el segundo 3. Como los canales PUB-SUB no tienen una persistencia fuerte, aquellos mensajes difundidos antes de que el suscriptor se conecte se habrán perdido. Por ello, el suscriptor recibirá todos los mensajes TSR a partir del 5; es decir, TSR 5, TSR 7, TSR 9...

E	`Received: PRG 0`
	`Received: TSR 1`
	`Received: PRG 2`
	…
	No. Ver la explicación en el apartado D.

F	Ninguna de las anteriores.