TSR: Segundo Parcial

Este examen consta de 20 cuestiones de opción múltiple. En cada una, solo una respuesta es correcta. Debe responderse en otra hoja. Las respuestas correctas aportan 0.5 puntos a la calificación del examen. Las erróneas descuentan 0.167 puntos.

TEORÍA

1. El despliegue incluye la instalación inicial y la configuración de una aplicación. Además de esas tareas, el despliegue de un servicio también comprende…

A	La depuración de los programas. Falso. La etapa de depuración debe ser aplicada cuidadosamente antes de que los programas se distribuyan y puedan ser instalados. El despliegue comprende aquellas etapas posteriores a la distribución del software, la primera de las cuales sería su instalación. Por tanto, la depuración no forma parte del despliegue.
B	La gestión del ciclo de vida del servicio. Cierto. La gestión del ciclo de vida de un servicio incluye aquellas tareas aplicadas a un servicio tras su instalación (por ejemplo, su activación, desactivación, eliminación, etc.). Esas tareas están incluidas en el despliegue.
C	El desarrollo de la aplicación. Falso. Tal como se ha sugerido en el primer apartado (puesto que el desarrollo de los programas precede a su depuración), el desarrollo no forma parte del despliegue.
D	El diseño de la aplicación. Falso. Como el diseño precede al desarrollo y la depuración, tampoco forma parte del despliegue.

2. El objetivo principal de la inyección de dependencias es…

A	Resolver las dependencias entre componentes utilizando ficheros de configuración. Falso. La inyección de dependencias consiste en resolver las dependencias entre componentes de forma diferida y transparente, con una intervención mínima (idealmente, ninguna) por parte de los administradores. Los ficheros de configuración requieren intervención del administrador. Por tanto, debe evitarse su uso en la inyección de dependencias.

B	Eliminar todas las dependencias entre componentes durante la etapa de diseño. Falso. Los componentes necesitan información (interfaces, "endpoints"…) sobre los demás componentes para poder interactuar con ellos. Algunos de estos datos solo podrán ser conocidos durante el despliegue (por ejemplo, los "endpoints" de otros componentes) o una vez el componente ya esté funcionando (por ejemplo, el nuevo "endpoint" de un componente que haya migrado, el nodo servidor de una base de datos responsable de cierta clave primaria cuando se utilice particionado horizontal, etc.).
C	Que la resolución de dependencias sea lo más transparente posible para el desarrollador de los componentes. Cierto. La inyección de dependencias consiste en facilitar la resolución de dependencias durante la ejecución de los componentes, enlazando estos dinámicamente con objetos que mantengan la información necesaria. Esos objetos se comportan como "proxies": facilitan la interfaz adecuada y ofrecen una imagen local que oculta la interacción con otros componentes externos, evitando así que el programa cliente deba dedicar esfuerzos a la resolución de dependencias.
D	Evitar el uso de contenedores, pues estos penalizan el rendimiento. Falso. Los contenedores automatizan algunos pasos de resolución de dependencias. Así, son una ayuda para implantar diferentes mecanismos de inyección de dependencias.

3. Imaginemos un servicio que necesite 400 ms para procesar localmente cada petición que modifique su estado. Esas peticiones invierten 20 ms para transmitir a otras réplicas el estado modificado y 30 ms en aplicar esas modificaciones en ellas. Un mensaje de petición puede ser difundido (en orden total) en esa red en 3 ms. Una petición de lectura puede ser gestionada localmente en 20 ms. La proporción de accesos es: 80% accesos de lectura y 20% accesos de modificación.
Para escalar este servicio, la mejor aproximación será…

A	Replicarlo utilizando el modelo activo. Falso. Bajo el modelo activo todas las réplicas dedicarán 400 ms para procesar localmente cada petición de modificación. El modelo pasivo solamente introduce esa carga en la réplica primaria. Las demás solo tendrán que dedicar 30 ms de procesamiento local para aplicar esas modificaciones. Por ello, las réplicas secundarias solo necesitan dedicar 1/13 del tiempo utilizado tanto por las réplicas del modelo activo como por la réplica primaria para gestionar estas modificaciones. Así, si un servicio se despliega utilizando cuatro réplicas, todas (bajo el modelo activo) o la primaria (bajo el modelo pasivo) tendrán una carga alta para gestionar solicitudes de modificación. Esa carga será comparativamente muy baja en las réplicas secundarias en el modelo pasivo. Esas réplicas secundarias podrán ser utilizadas como primarias para otros servicios replicados, pues un alto porcentaje de su capacidad de cómputo permanecerá libre. Por tanto, el modelo activo no es una buena aproximación para gestionar este servicio replicado.

B	Replicarlo utilizando el modelo pasivo, procesando todas las solicitudes en la réplica primaria. Falso. Aunque esta es una alternativa mejor que la anterior, la que se describe en el apartado C es todavía mejor.
C	Replicarlo utilizando el modelo pasivo, pero permitiendo que las solicitudes de lectura sean procesadas por las réplicas secundarias. Cierto. Con esta configuración la carga puede equilibrarse mejor entre todas las réplicas. Obsérvese que en el modelo activo todas las peticiones deben ser propagadas a todas las réplicas para superar de esa manera fallos arbitrarios. Por tanto, tanto en el modelo activo (apartado A) como en el pasivo estricto (apartado B) no se llega a dar ninguna distribución de carga entre las réplicas.
D	No replicarlo, pues la replicación introduce demasiada coordinación y eso impide un escalado eficiente. Falso. Los servicios distribuidos deben superar las situaciones de fallo. Si no se replicara este servicio, se estaría introduciendo un punto único de fallo allí donde se despliegue el servicio. Aunque la replicación introduzca cierto nivel de coordinación, las peticiones de consulta pueden distribuirse entre las réplicas (véase descripción del apartado C) y, así, se podrá escalar el servicio de manera eficiente. Sin replicación, la única instancia debería atender todas las peticiones. Así, esa única instancia podría saturarse fácilmente.

4. En los modelos de consistencia centrados en datos, podemos decir que el modelo A es más fuerte que el modelo B en los siguientes casos:

A	A: causal, B: caché. Falso. Los modelos causal y caché no pueden compararse. Hay ejecuciones que son causales pero no caché y otras que son caché pero no causales. Un modelo A es más fuerte que otro B cuando todas las ejecuciones de A también respetan B y hay ejecuciones que respetan B pero no A.
B	A: FIFO, B: caché. Falso. Los modelos FIFO y caché no pueden compararse. Hay ejecuciones que son FIFO pero no caché y otras que son caché pero no FIFO.
C	A: causal, B: secuencial. Falso. El modelo secuencial es más fuerte que el causal.
D	A: causal, B: FIFO. Cierto. El modelo causal es más fuerte que el FIFO; es decir, todas las ejecuciones causales son también FIFO, pero hay ejecuciones FIFO que no son causales. Por ejemplo, en un sistema con tres procesos, esta ejecución… W1(x)1, R2(x)1, W2(x)2, W1(x)3, R2(x)3, R3(x)2, R3(x)1, R3(x)3, R1(x)2, … …es FIFO porque P1 ha escrito dos valores diferentes (1 y 3, en ese orden) sobre "x" y P3 y P2 han recibido ambos valores en el orden de escritura. Como no hay otros procesos que hayan escrito más de una vez, esto significa que todas las condiciones de la consistencia FIFO han sido respetadas en la ejecución. Sin embargo, P2 escribió el valor 2 una vez había leído el valor 1. Eso implica que ambos valores están causalmente relacionados (1 → 2), pero P3 no los obtuvo en ese orden causal. Por tanto, la ejecución no es causal y esto demuestra que el modelo FIFO es más relajado que el causal.

5. Los almacenes escalables NoSQL no soportan el modelo relacional porque…

A	El modelo relacional no admite replicación. Falso. Las bases de datos relacionales pueden replicarse, como cualquier otro servicio distribuido. Por ejemplo, hay múltiples ediciones de MySQL, Oracle o Microsoft SQL Server (entre otras compañías de software) que replican sus bases de datos.
B	El modelo relacional no admite particionado horizontal ("*sharding*"). Falso. Las bases de datos relacionales pueden particionarse horizontalmente. De hecho, las primeras propuestas de particionado horizontal se realizaron sobre bases de datos relacionales replicadas.
C	Los datos relacionales deben mantenerse en disco. Falso. Los datos deben mantenerse en disco en ese tipo de bases de datos, pero eso no impide que se apliquen múltiples técnicas de escalado.
D	Las transacciones en el modelo relacional necesitan mecanismos de control de concurrencia que pueden ser complejos. Cierto. Las transacciones utilizadas en las bases de datos relacionales se complican cuando se introduce replicación. Por ejemplo, para finalizar una transacción se necesita un algoritmo distribuido (2PC o 3PC) y ese algoritmo introduce una interacción fuerte entre los agentes participantes. Adicionalmente, los mecanismos de control de concurrencia pueden necesitar su extensión a una gestión distribuida (utilizando "locks" distribuidos, por ejemplo), y esto complica su gestión e introduce interacción fuerte entre los procesos participantes. Por ello, los almacenes NoSQL han renunciado a las transacciones con garantías ACID para mejorar su escalabilidad.

6. El teorema CAP…

A	…exige que los servicios escalables y disponibles utilicen siempre el modelo de consistencia estricto para tolerar así las particiones de la red. Falso. Al contrario, el teorema CAP dice que la consistencia estricta no puede mantenerse en servicios que pretendan ser altamente disponibles mientras se den particiones en la red.

B	...permite que los servicios escalables relajen su consistencia mientras la red permanezca particionada, asegurando así su disponibilidad. Cierto. El teorema CAP dice que los servicios distribuidos no pueden soportar simultáneamente consistencia fuerte, disponibilidad y tolerancia a las particiones de la red. Así, cuando la red se particiona debe elegirse entre disponibilidad o consistencia fuerte. Si un servicio prefiere estar disponible debe relajar su consistencia mientras la red esté particionada.
C	...no permite la implantación de servicios altamente disponibles utilizando modelos de consistencia fuertes. Falso. Los modelos de consistencia fuerte pueden utilizarse en servicios distribuidos. Lo que dice el teorema CAP es que si un servicio quiere mantener una consistencia fuerte mientras la red se particione, entonces tendrá que renunciar a su disponibilidad en todas las réplicas. Esto significa que en subgrupos minoritarios no podrá responderse a las peticiones de los clientes y, por tanto, esas réplicas no estarán disponibles. De hecho, el teorema CAP sí que tolera alta disponibilidad y consistencia fuerte. Para ello, el servicio debe desplegarse de tal manera que se garantice que no haya particiones en la red. Por ejemplo, en despliegues para unos pocos procesos clientes con pocas réplicas servidoras en un mismo laboratorio y con la red de comunicaciones replicada.
D	...no tiene sentido en los centros de datos de computación en la nube, pues jamás habrá particiones de la red en ellos. Falso. La computación en la nube ofrece la imagen de una escalabilidad ilimitada. Para ello, pueden utilizarse múltiples centros de datos para desplegar algunos servicios. Incluso en caso de utilizar un solo centro de datos, ese centro tendrá muchos "racks" de ordenadores y muchos segmentos de red. En esos casos se pueden seguir dando particiones en la red. Por tanto, las restricciones establecidas en el teorema CAP todavía son aplicables en ese tipo de despliegues.

7. Respecto a la escalabilidad de servicios se puede afirmar que…

A	Un mismo servicio no puede escalar horizontal y verticalmente. Falso. Las escalabilidades horizontal y vertical no son mutuamente excluyentes.
B	Los algoritmos descentralizados mejoran la escalabilidad de distancia. Falso. Los algoritmos descentralizados mejoran la escalabilidad de tamaño, pero no necesariamente la escalabilidad de distancia.
C	El particionado horizontal ("*sharding*") mejora la escalabilidad administrativa. Falso. El particionado horizontal mejora la escalabilidad de tamaño, pero no simplifica ni mejora la escalabilidad administrativa.
D	Evitar la contención es un factor clave para mejorar la escalabilidad de un servicio. Cierto. La contención evita que los servicios escalen ya que bloquea la ejecución de estos servicios.

8. Los objetivos principales de un subsistema de seguridad son:

A	Protección, control de acceso y seguridad física. Falso. El control de acceso y la seguridad física no son objetivos. Son mecanismos que pueden utilizarse para implantar políticas de seguridad.

B	Protección, gestión de la confianza y un buen soporte para mecanismos de cifrado. Falso. El cifrado es un mecanismo a utilizar para mejorar la confidencialidad e integridad de un sistema. La gestión de la confianza es un mecanismo para asegurar y evaluar la corrección de un sistema de seguridad. No forman parte de los objetivos sino del conjunto de mecanismos.
C	Contabilidad, integridad, confidencialidad y disponibilidad. Cierto. Estos son los cuatro objetivos que se han presentado en el Tema 8 para los subsistemas de seguridad.
D	Políticas robustas, mecanismos eficientes y garantías correctas. Falso. Estos son los tres tipos de herramientas necesarios para especificar, implantar y evaluar la corrección de un sistema de seguridad, respectivamente.

SEMINARIOS

9. ¿Cuál de las siguientes tareas NO se realiza al utilizar esta orden Docker?

```
docker run –it ubuntu /bin/bash
```

A	Ejecutar el programa /bin/bash en un contenedor. Falso. El último argumento en esta línea de órdenes indica que el programa a ejecutar en el contenedor será "/bin/bash".
B	Descargar la imagen "ubuntu:latest" desde Docker Hub si no la teníamos en el depósito de imágenes local. Falso. El argumento "ubuntu" especifica el nombre de la imagen a utilizar para generar el contenedor. Cuando no se especifica ninguna etiqueta, Docker asume "latest". Por tanto, la imagen a buscar será "ubuntu:lastest". Docker busca esa imagen en el depósito local. Si no está allí, Docker la descargará del Docker Hub y la mantendrá en el depósito local a partir de entonces.
C	Recoger la salida del contenedor que está siendo utilizado para ejecutar esa orden. Esa salida puede mostrarse mediante la orden `docker logs`. Falso. La salida del contenedor es almacenada por el servidor Docker y puede ser mostrada mediante la orden "docker logs" utilizando el identificador o nombre del contenedor como argumento.
D	Eliminar automáticamente este contenedor una vez su ejecución haya terminado. Cierto. Para que haga eso, la línea de órdenes debería incluir la opción "--rm=true". Sin ella, el contendor se mantendrá una vez finalice su ejecución.

10. La orden `docker commit a b` …

A	Crea un nuevo contenedor llamado "a" utilizando el Dockerfile ubicado en la carpeta "b". Falso. Para generar un nuevo contenedor debe utilizarse la orden "docker run" o "docker create".

B	Crea una nueva imagen "b" utilizando el Dockerfile de la carpeta "a". Falso. Para crear una nueva imagen a partir de un Dockerfile se debe utilizar la orden "docker build".
ℂ	Crea una nueva imagen "b" con el contenido actual del contenedor cuyo nombre o identificador es "a". Cierto. Esta es una descripción breve de lo que hace la orden "docker commit".
D	Realiza el "commit" de una transacción "a" que fue iniciada con una orden `docker pull` o `docker push`, generando un contenedor con ID "b". Falso. Docker no utiliza transacciones.

11. El módulo "cluster" de NodeJS se utiliza para…

A	…desplegar un conjunto de programas NodeJS en un "*cluster*" de ordenadores. Falso. Los módulos NodeJS se utilizan durante la etapa de desarrollo. No se utilizan durante la etapa de despliegue.
B	…gestionar múltiples hilos en un proceso NodeJS. Falso. Los programas JavaScript solo tienen un único hilo de ejecución. Es imposible gestionar múltiples hilos en ese lenguaje y NodeJS es un intérprete de JavaScript.
ℂ	…gestionar un conjunto de procesos NodeJS para que puedan compartir algunos recursos; p.ej., un puerto en el que escuchar y un mismo programa a ejecutar. Cierto. Ese es el objetivo de este módulo.
D	…implantar fácilmente múltiples modelos de consistencia de memoria. Falso. No está diseñado para este fin.

12. MongoDB utiliza los siguientes mecanismos para mejorar su escalabilidad:

A	Control de concurrencia distribuido. Falso. El control de concurrencia distribuido es más complejo e introduce mayor contención que el control de concurrencia local. Por ello, no mejora la escalabilidad.
B	Algoritmos descentralizados. Falso. Aunque los algoritmos descentralizados son un buen mecanismo para mejorar la escalabilidad, no se ha descrito ningún algoritmo descentralizado en la documentación sobre MongoDB.
ℂ	Replicación pasiva y particionado horizontal (o "*sharding*"). Cierto. MongoDB utiliza replicación pasiva y particionado horizontal (incrementando así su capacidad para atender concurrentemente múltiples peticiones clientes sobre varias instancias MongoDB) para ser escalable.
D	Escalabilidad administrativa. Falso. No se ha descrito ningún mecanismo de configuración de MongoDB para mejorar su escalabilidad administrativa y, de esa manera, mejorar su escalabilidad general.

13. El objetivo principal de los servidores de configuración en MongoDB es:

A	Adoptar el papel de árbitros cuando una réplica falle. Falso. El papel de árbitro solo puede ser desempeñado por un servidor "mongod", nunca por un servidor de configuración.

B	Controlar la distribución de datos entre las múltiples particiones horizontales ("shards") existentes. Cierto. Este es el principal objetivo de los servidores de configuración en MongoDB.
C	Respetar el teorema CAP cuando se dé una partición en la red. Falso. No se requiere la intervención de ningún servidor de configuración en MongoDB cuando la red se particione. Esa gestión se realiza en el ámbito de los "conjuntos de réplicas" ("*replica set*"). En esos casos, solo el subgrupo mayoritario podrá continuar y solo cuando mantenga más de la mitad de sus réplicas.
D	Detectar fallos en los nodos, iniciando un protocolo de recuperación cuando se dé algún fallo. Falso. La documentación de MongoDB no describe en detalle sus mecanismos de detección de fallos. No está claro qué elementos participan en esa detección. Sin embargo, esa gestión se realiza en el ámbito de los "conjuntos de réplicas", de manera interna a cada conjunto. Eso sugiere que la detección está integrada en las interacciones entre los servidores "mongod" que forman cada conjunto.

14. Considerando la clasificación temática de vulnerabilidades vista en el Seminario 8, se puede considerar cierta la siguiente afirmación:

A	La explotación de defectos en políticas de seguridad no puede automatizarse tan fácilmente como la explotación de contraseñas débiles. Cierto. Los defectos en las políticas de seguridad deben ser cuidadosamente analizados y comprobados por un atacante para identificarlos. Esto requiere mucho tiempo e interacción con el sistema. Por otra parte, hay múltiples listas de "contraseñas débiles". Una vez se conozca el identificador de un usuario, un atacante puede intentar un acceso remoto utilizando alguna de esas listas. Por tanto, la explotación de esta segunda vulnerabilidad puede ser fácilmente automatizada.
B	El "*phishing*" es una vulnerabilidad de tipo "error software". Falso. El "*phishing*" es una vulnerabilidad de "ingeniería social".
C	La protección física es un ejemplo de vulnerabilidad de ingeniería social. Falso. La protección física es un mecanismo de seguridad. No es una vulnerabilidad.
D	Un defecto en una política de seguridad de protección personal requiere menor interacción para ser explotada que un error software en el sistema operativo. Falso. Las vulnerabilidades de los sistemas operativos suelen estar documentadas (por ejemplo, en la descripción de las actualizaciones que las corrigen). Por tanto, no se necesita interactuar previamente con el sistema para explotar estas vulnerabilidades. Por el contrario, las políticas de seguridad de protección personal no deben ser conocidas por agentes externos y sus defectos, cuando los haya, necesitarán mayor interacción para ser identificados y explotados en un ataque.

15. Asumiendo este Dockerfile…

```
FROM zmq
RUN mkdir /zmq
COPY ./broker.js /zmq/broker.js
WORKDIR /zmq
EXPOSE 8000 8001
CMD node broker.js
```

¿Cuál de las siguientes afirmaciones es FALSA?

A	Necesitamos tener el fichero "broker.js" en el directorio del anfitrión en el que se encuentre este Dockerfile.

	Sí. La tercera línea del Dockerfile asume que ese fichero se encuentra en el mismo directorio del Dockerfile. Los nombres de ruta relativos utilizados como primer argumento de la orden COPY asumen ese directorio como origen de la ruta.
B	El programa a ejecutar en estos contenedores utiliza el puerto 8000 del contenedor y lo asocia al puerto 8001 del anfitrión. Falso. Esos números de puerto aparecen en la quinta línea del fichero. La orden EXPOSE lista los números de puerto que va a utilizar el contenedor para escuchar conexiones. Son puertos del contenedor. No se establece ninguna correspondencia con los puertos del anfitrión.
C	Por omisión, los contenedores generados a partir de este Dockerfile ejecutarán la orden "**node broker.js**". Sí. Esto se indica en la última línea del Dockerfile, utilizando la orden CMD.
D	Este Dockerfile asume la existencia de una imagen llamada "zmq" con una instalación válida del intérprete de JavaScript "node". Sí. La primera línea del fichero indica que la imagen a utilizar como base se llama "zmq". La última línea utiliza el intérprete "node". Por tanto, se puede asumir que "zmq" ya incluye una instalación correcta de ese intérprete, pues ninguna de las demás líneas del Dockerfile trata de instalar "node".

16. Imaginemos que el componente broker de la cuestión 15 se incluye en un fichero docker-compose.yml con estos contenidos (entre otros que correspondan a otros componentes):

```
version: '2'
services:
...
 bro:
  image: broker
  build: ../broker/
```

¿Cuál de las siguientes afirmaciones es FALSA?

A	Podemos iniciar al menos una instancia del componente **broker** con la orden **docker-compose up –d** Cierto. La orden "docker-compose up" inicia una instancia (o más, si estas fueron paradas previamente) de cada componente mencionado en el **docker-compose.yml**.

B	Una vez se ha iniciado el servicio, podemos tener 5 instancias del componente **broker** utilizando **docker-compose scale broker=5** Falso. El componente se llama "**bro**" en ese fichero. Por tanto, la línea a utilizar sería: **docker-compose scale bro=5**
C	Este fichero "docker-compose.yml" asume que el Dockerfile mostrado en la cuestión 15 se encuentra en la carpeta "../broker/". Cierto. Ese es el objetivo de la orden "**build**": establecer la ubicación del Dockerfile necesario para construir la imagen.
D	Podemos utilizar la orden **docker-compose stop bro** para parar todas las instancias de este componente **broker**. Cierto. Esa es la orden docker-compose necesaria para ello.

17. Supongamos un protocolo de replicación basado en un proceso secuenciador que utiliza un socket PUB ZeroMQ para propagar todos los eventos "write" a los procesos participantes, en orden de recepción. Esos eventos han sido recibidos mediante canales PUSH-PULL, cuyo socket PULL está en el secuenciador. Ese protocolo de replicación soporta los siguientes modelos de consistencia:

A	Solo el modelo estricto. Falso. El modelo de consistencia estricto requiere un alto grado de coordinación para ser implantado. Los sockets ZeroMQ soportan comunicación asincrónica. Por ello, una implantación sencilla basada en ZeroMQ no puede garantizar una consistencia estricta.
B	Solo el modelo caché. Falso. La implantación descrita en esta cuestión es la vista en el Seminario 5 para soportar el modelo de consistencia secuencial. La consistencia secuencial es más fuerte que la caché. Eso significa que los procesos también soportarán la consistencia caché, pero no solo ese modelo.
C	Solo el modelo causal. Falso. La implantación descrita en esta cuestión es la vista en el Seminario 5 para soportar el modelo de consistencia secuencial. La consistencia secuencial es más fuerte que la causal. Eso significa que los procesos también soportarán la consistencia causal, pero no solo ese modelo.
D	Secuencial, procesador, causal, caché y FIFO. Cierto. La implantación descrita en esta cuestión es la vista en el Seminario 5 para soportar el modelo de consistencia secuencial. La consistencia secuencial es más fuerte que la procesador, causal, caché y FIFO. Por ello, los procesos resultantes soportarán todos esos modelos de consistencia.

18. Dada la siguiente ejecución:
W1(x)1, R4(x)1, W2(y)2, W1(y)3, W2(x)4, R3(y)2, W3(x)5, R1(x)5, R2(x)1, R3(x)1, R4(y)3, R1(y)2, R3(y)3, R4(x)5, R3(x)4, R2(y)3, R4(y)2, R1(x)4, R2(x)5, R4(x)4.
Esa ejecución soporta estos modelos de consistencia:

A	Solo el modelo FIFO. Cierto. Hay cinco acciones de escritura en esta ejecución. Dos pertenecen a P1 (valor 1 sobre "x" y 3 sobre "y", en ese orden), dos más a P2 (valor 2 sobre "y" y 4 sobre "x", en ese orden) y otra a P3 (valor 5 sobre "x"). La consistencia FIFO exige que los valores escritos por un proceso sean leídos en orden de escritura por los demás procesos.

	Esto exige que el valor 1 deba preceder al valor 3 y el valor 2 al valor 4. Eso se ha respetado en todos los procesos. Esos dos pares de valores escritos pueden ser intercalados entre sí libremente por los lectores. Además, el valor 5 también puede intercalarse sin restricciones, pues ha sido escrito por otro proceso. En la ejecución, cada proceso ha visto estas secuencias de valores: P1: 1, 3, 5, 2, 4 P2: 2, 4, 1, 3, 5 P3: 2, 5, 1, 3, 4 P4: 1, 3, 5, 2, 4 Por tanto, los procesos respetan el modelo de consistencia FIFO. Comprobemos ahora si se cumplen los demás modelos de consistencia. Empezaremos con los modelos caché y causal. Caché no es comparable con FIFO y es más relajado que procesador y este último más relajado que secuencial. Causal es más fuerte que FIFO y más relajado que secuencial. Por tanto, si el caché no se soporta, tampoco se soportará el procesador (ni el secuencial) y si el causal no se soporta, tampoco se soportará el secuencial. La consistencia caché exige que todos los procesos acuerden una secuencia de valores para cada variable (considerando cada variable por separado). Tenemos tres valores para "x" (1, 4 y 5) y dos para "y" (2 y 3). Desafortunadamente, no hay ningún acuerdo sobre los valores de "x" (P1 y P4 han visto 1, 5, 4; P2 ha visto 4, 1, 5; y P3 ha visto 5, 1, 4). Eso es suficiente para decir que la ejecución no respeta el modelo caché. Por tanto, tampoco respetará el modelo procesador (ni el secuencial). Aparte, tampoco ha habido acuerdo sobre los valores de "y" (P1 y P4 han visto 3, 2; mientras P2 y P3 han visto 2, 3). Respecto a la consistencia causal, hay un "camino" de dependencias causales entre los valores 2 y 5 [W2(y)2, …, R3(y)2, W3(x)5,…] ya que el escritor del valor 5, P3, leyó el valor 2 antes de escribir el valor 5. Eso implica que todos los procesos deben ver el valor 2 antes que el 5 para cumplir con la consistencia causal. Eso no ocurre en los procesos P1 y P4. Por tanto, la ejecución no es causal y tampoco será secuencial. Por todo esto, la ejecución solo respeta el modelo FIFO.
B	Solo el modelo caché. Falso. Véase la explicación del apartado A.
C	Procesador, FIFO y caché. Falso. Véase la explicación del apartado A.
D	Secuencial, procesador, causal, caché y FIFO. Falso. Véase la explicación del apartado A.

19. Dado el siguiente programa servidor de descarga de ficheros…

```
var cluster = require('cluster');
var fs = require('fs');
var path = require('path');
var zmq = require('zmq');
var os = require('os');
const ipcName = 'Act2.ipc';
const dlName = 'ipc://'+ipcName;
if (cluster.isMaster) {
   var numCPUs = os.cpus().length;
   var rt = zmq.socket('router');
   var dl = zmq.socket('dealer');
   rt.bindSync('tcp://127.0.0.1:8000');
   dl.bindSync(dlName);
   rt.on('message', function() {
         msg = Array.apply(null, arguments);
         dl.send(msg);   });
   dl.on('message', function() {
         msg = Array.apply(null, arguments);
         rt.send(msg);   });
} else {
  var rep = zmq.socket('rep');
  rep.connect(dlName);
  rep.on('message', function(data) {
    var request = JSON.parse(data);
    fs.readFile(request.path, function(err,
data) {
      if (err) data = ' NOT FOUND';
       rep.send(JSON.stringify({
          pid: process.pid ,
          path: request.path ,
          data: data ,
          timestamp: new Date().toString()
       }))
    })
  })
}
```

Hemos tratado de ejecutar el programa, pero no parece hacer nada útil. Su principal problema es…

A	Los sockets ZeroMQ no admiten "**ipc://**" como transporte. Falso. El transporte "**ipc:**" puede usarse para comunicar dos o más procesos ubicados en un mismo ordenador. Está admitido.
B	No se ha creado ningún proceso trabajador del módulo "cluster". Cierto. Debemos generar múltiples procesos trabajadores (mediante la función **cluster.fork**) en el código del proceso maestro. No hay ninguna instrucción de este tipo en esta versión del programa mostrado en el enunciado.
C	Se está tratando de propagar mensajes internos del módulo "cluster" a través de un socket DEALER ZeroMQ. Falso. Los mensajes intercambiados entre los procesos maestro y trabajadores en este programa utilizan canales DEALER-REP de ZeroMQ. No pueden calificarse como mensajes "internos" del módulo "cluster". Un canal DEALER-REP en ZeroMQ puede utilizarse sin problemas. De hecho, es la implantación más sencilla para comunicar un broker con varios trabajadores en caso de utilizar un patrón ROUTER-DEALER en el proceso broker.
D	Un servidor no puede utilizar un socket ROUTER como su "endpoint". Falso. Hemos visto varios ejemplos de servicios replicados que utilizan un proceso broker para equilibrar la carga entre sus réplicas. Ese broker utiliza normalmente un socket ROUTER como "endpoint".

20. La vulnerabilidad OpenSSL Heartbleed descrita en el Seminario 8 es un ejemplo de vulnerabilidad que pertenece a las clases siguientes:

A	"Defecto en la política de trabajo" en su categoría y "Personal humano" en su origen. Falso. Véase la justificación del apartado B.

B	“Defecto en la lógica del software” en su categoría y “Biblioteca/middleware” en su origen. Cierto. Esta vulnerabilidad fue causada por un error en la lógica del software (los mensajes de “heartbeat” del protocolo SSL necesitaban una respuesta de un tamaño especificado por el cliente y ese tamaño no estaba limitado, pudiendo obtener así una copia de un fragmento de la memoria del servidor que podría contener información importante; por ejemplo, contraseñas o certificados) de una biblioteca (OpenSSL). Por tanto, su categoría es “defecto en la lógica del software” y su origen es “biblioteca/middleware” pues el problema afectaba a aquellos programas que estuvieran utilizando las versiones vulnerables de la biblioteca OpenSSL.
C	“Ingeniería social” en su categoría y “Desarrollador” en su origen. Falso. Véase la justificación del apartado B.
D	“Defecto en la lógica del software” en su categoría y “Personal humano” en su origen. Falso. Véase la justificación del apartado B.