GRUPO:		Nota	C	I	N
NOMBRE:		APELLIDOS:			
FIRMA:		DNI:			

Este examen consta de 40 preguntas de tipo test. En cada cuestión debe seleccionar una única alternativa de entre todas las propuestas. Escriba para ello una "X" en la casilla que elija. El valor de cada cuestión es 0.25 puntos en caso de respuesta correcta y -0.05 en caso de error (equivalente a 1/5 del valor correcto).

Si duda, escriba un asterisco "*" o un número "1", "2"... al final del texto del apartado correspondiente y utilice el espacio en los márgenes de la hoja para explicar su respuesta, que deberá ser forzosamente breve.

El examen puede ser completado en hora y media, pero se dispone de 2 horas para finalizar.

1. Los sistemas distribuidos

	En caso de ser escalables, no podrán ser sistemas concurrentes. Todo sistema distribuido es un sistema concurrente. En un sistema concurrente habrá múltiples actividades que colaborarán entre sí. En el caso particular del sistema distribuido, las actividades podrán estar ubicadas en múltiples ordenadores. Un sistema "no concurrente" estaría formado por una sola actividad y jamás podría ser "distribuido".
	Utilizan sólo mensajes como mecanismo de intercomunicación. Las actividades de un sistema distribuido necesitarán utilizar mensajes para intercambiar información cuando residan en ordenadores diferentes. Sin embargo, nada prohíbe que varias actividades del sistema residan en un mismo nodo y utilicen memoria compartida para intercomunicarse (un fichero, por ejemplo, en caso de tratarse de varios procesos; una variable global, en caso de tratarse de varios hilos de ejecución...).
	Pueden estar formados por un único proceso. Un sistema distribuido debería estar formado por múltiples actividades que se ejecuten en múltiples ordenadores, que colaboren entre sí, y que ofrezcan una imagen de sistema único. Esos tres requisitos no pueden ser satisfechos simultáneamente por un único proceso.
	Actualmente sólo siguen el modelo de computación en la nube. Cualquier aplicación concurrente que utilice la red será un ejemplo de sistema distribuido. No se exige que se desplieguen en un entorno elástico basado en virtualización, como es el caso de los sistemas de computación en la nube.
	Todas las anteriores.
X	Ninguna de las anteriores.

2. Los roles desarrollador, proveedor de servicios, administrador de sistema y usuario

X	Están claramente diferenciados en un sistema SaaS, pero no siempre están desempeñados por diferentes agentes.
	En los ordenadores personales están desempeñados conjuntamente por el propietario del ordenador. Normalmente el propietario de un ordenador no es un programador avanzado capaz de desarrollar todas las aplicaciones que necesita. Por tanto, el rol "desarrollador" no está desempeñado por el usuario en la mayoría de los casos.
	En los "*mainframes*" el usuario también actuaba a veces como administrador del sistema. Los roles de usuario y administrador del sistema estaban claramente diferenciados en aquellos sistemas. Las tareas de administración eran delicadas y no podían ser realizadas por los usuarios.
	En los centros de cómputo empresariales resulta trivial y barato gestionar los roles de proveedor de servicios y administrador de sistema. Ni resulta trivial ni barato. Hay que recurrir a personal especializado para desarrollar estas tareas. Eso supone un coste económico alto para la empresa.
	Todas las anteriores.
	Ninguna de las anteriores.

3. Ejemplos de servicios SaaS

	Linux Linux es un ejemplo de sistema operativo. No sigue el modelo SaaS de computación en la nube. El modelo SaaS consiste en ofrecer una aplicación concreta como servicio elástico. Un sistema operativo no puede considerarse una aplicación.
X	Google Drive Google Drive es el servicio ofrecido por Google para: facilitar cierto espacio de almacenamiento de ficheros a través de la red y acceder a las aplicaciones de Google Docs (gestión de presentaciones, hojas de cálculo, procesamiento de textos...). Si que cumple con lo que se espera de un modelo SaaS: son aplicaciones ofrecidas como servicio a través de la red.
	ZeroMQ ZeroMQ es un middleware de comunicaciones. Se utilizará junto a otras herramientas a la hora de desarrollar aplicaciones distribuidas escalables, pero no es una aplicación. Por tanto, no respeta el modelo de servicio SaaS.
	Microsoft Word Todavía identificamos a Microsoft Word con una aplicación que se instala en un sistema Windows y que se ejecuta de manera local. Bajo ese supuesto, no respeta el modelo de servicio SaaS.
	Todas las anteriores.
	Ninguna de las anteriores.

4. Sobre los modelos de servicio en la computación en la nube:

X	En el modelo IaaS el proveedor ofrece redes y máquinas virtuales para que el "usuario" despliegue allí sus aplicaciones distribuidas.
	En el modelo SaaS el proveedor ofrece redes y máquinas virtuales para que el "usuario" despliegue allí sus aplicaciones distribuidas. Un proveedor en el modelo SaaS debe ofrecer una aplicación como servicio.
	En el modelo PaaS el proveedor ofrece aplicaciones distribuidas, garantizando su escalabilidad y mantenibilidad para que el usuario las utilice directamente. Eso es lo que ocurre en el modelo SaaS.
	En el modelo IaaS se necesita un proveedor PaaS subyacente para que los servicios IaaS puedan facilitarse a los usuarios. Justo al contrario. Un modelo PaaS suele utilizar una infraestructura escalable (que puede ser facilitada por un IaaS). Por tanto, PaaS necesita como nivel inferior a IaaS.
	Todas las anteriores.
	Ninguna de las anteriores.

5. Se han visto diferentes modelos o niveles de servicios en la nube...:

X	Hay tres: IaaS, PaaS y SaaS. Infraestructura como servicio (IaaS), plataforma como servicio (PaaS) y *software* como servicio (SaaS).
	Hay cinco: Fiables, Disponibles, Seguros ("Safe"), Mantenibles y Seguros ("Secure"). Estos son los atributos de un sistema robusto. No son modelos de servicios.
	Hay cinco: Secuenciales, Causales, Procesadores, FIFOs, y Cachés. Estos son ejemplos de modelos de consistencia. No son modelos de servicios.
	Hay dos: Activos y Pasivos. Son ejemplos de modelos de replicación. No son modelos de servicios.
	Todas las anteriores.
	Ninguna de las anteriores.

6. Mecanismos necesarios en los sistemas de computación en la nube:

X	Virtualización. La virtualización es un mecanismo mediante el que se facilita la administración de los recursos hardware en un sistema de computación en la nube. Es la base para proveer una infraestructura elástica.
	Gestión secuencial (sin concurrencia). Cualquier sistema distribuido es concurrente. Al menos habrá una actividad por cada agente o componente que participe en el sistema.
	Escalabilidad. La escalabilidad es un objetivo de la computación distribuida (y, por tanto, de la computación en la nube). No es un mecanismo.
	Consistencia fuerte. La consistencia se considera una propiedad de un sistema. No es un mecanismo. En cualquier caso, la gestión de la consistencia necesita algún protocolo de replicación que controle los accesos de lectura y escritura. En caso de requerirse una consistencia fuerte, ese protocolo exigiría una sincronización pesada y el sistema no sería escalable.
	Todas las anteriores.
	Ninguna de las anteriores.

7. El despliegue de una aplicación distribuida...:

X	Planteará algunos problemas prácticos: actualización de componentes, mantenibilidad, disponibilidad...
	Está automatizado en los sistemas IaaS actuales. No es así. Es uno de los objetivos de los sistemas PaaS. No está resuelto con procedimientos automatizados en los sistemas IaaS.
	Está gestionado por el usuario en los sistemas SaaS actuales. No lo gestiona el usuario en esos sistemas, sino el proveedor.
	Es un problema ya resuelto desde los sistemas basados en "*mainframes*". No. Todavía es un problema abierto en ciertos entornos y para ciertas tareas incluidas en ese despliegue. No está resuelto de manera general en los sistemas distribuidos. La afirmación del primer apartado menciona los aspectos a considerar para "resolverlo".
	Todas las anteriores.
	Ninguna de las anteriores.

8. Un modelo teórico sencillo de sistema distribuido suele considerar que...:

X	Los procesos son agentes secuenciales.
	No hay ningún tipo de fallo. Se asumía que los procesos podían fallar, parando.
	Los procesos no necesitan comunicarse entre sí. Si los procesos no se comunicaran, no habría un sistema distribuido. No podrían colaborar. Hablaríamos de un conjunto de procesos no relacionados.
	Los eventos relevantes son externos en todos los casos y provocados por el transcurso del tiempo. Se distingue entre eventos de entrada, de salida e internos. Los de entrada y salida forman el superconjunto de eventos externos. Es decir, no todos son externos. Los eventos externos están asociados a la comunicación entre agentes (no se generan sólo porque transcurra el tiempo).
	Todas las anteriores.
	Ninguna de las anteriores.

9. Respecto a la sincronía en un modelo teórico simple de sistema distribuido...:

	Habrá procesos sincrónicos si todos ellos son capaces de gestionar exactamente un evento en cada paso del algoritmo.
	Habrá canales sincrónicos cuando se pueda acotar el tiempo de propagación y entrega de cada mensaje.
	Habrá comunicación sincrónica cuando el emisor se bloquee mientras no reciba una respuesta del receptor.
	Tendríamos relojes sincrónicos si el reloj de cada nodo estuviera sincronizado con un “tiempo real global”.
X	Todas las anteriores. Esas fueron las definiciones utilizadas en el modelo teórico simple para cada uno de esos conceptos.
	Ninguna de las anteriores.

10. En los sistemas distribuidos escalables, al comparar servidores multi-hilo con servidores asincrónicos...:

	Los asincrónicos plantean el problema de suspenderse con cada petición recibida. Al contrario: minimizan las necesidades de bloqueo o suspensión.
X	Los asincrónicos suelen estar orientados a eventos y se corresponden mejor con un modelo teórico sencillo de sistema distribuido.
	Los multi-hilo siempre ofrecen mejor rendimiento al poder realizar múltiples acciones simultáneamente. Sí que permiten realizar múltiples acciones simultáneamente, pero cuando esas acciones impliquen acceder a algún recurso compartido habrá que utilizar mecanismos de control de concurrencia que introducirán bloqueos y penalizarán el rendimiento.
	Los multi-hilo ofrecen una gestión de estado más sencilla, pues no hay que preocuparse por las "guardas" ni por las "acciones" asociadas a los "eventos". Es cierto que no habrá "guardas" ni "acciones" asociadas a ellas, pero la gestión de estado es más compleja, a pesar de ello. En los multi-hilo habrá que identificar apropiadamente las secciones críticas y protegerlas con los mecanismos de sincronización adecuados. Así, la gestión resulta ser más compleja, provocando con frecuencia el bloqueo de las actividades.
	Todas las anteriores.
	Ninguna de las anteriores.

11. Los servidores multi-hilo...:

	Proporcionan un modelo de programación asincrónico. Al contrario, el modelo de programación asincrónico está proporcionado por los servidores asincrónicos. Los servidores multi-hilo necesitan mecanismos de sincronización para no generar inconsistencias al modificar recursos compartidos.
	No requieren herramientas de sincronización al acceder a recursos compartidos. Ya explicado en el apartado anterior. Sí que necesitan ese tipo de herramientas.
	Minimizan el uso de mensajes entre agentes, incrementando la escalabilidad de los servicios que implantan. Es posible que no requieran tantos mensajes para intercomunicar a los diferentes agentes, pero eso no garantiza una mayor escalabilidad. Al tener que utilizar herramientas de sincronización y acceder a los recursos compartidos en exclusión mutua, se introducirán bloqueos prolongados cuando haya que soportar altas cargas de trabajo. Por tanto, la escalabilidad no mejora.
	Pueden aumentar la escalabilidad del servicio que implanten si en la mayoría de las operaciones se utilizan recursos compartidos. Podría mejorar la escalabilidad SI NO SE UTILIZARAN recursos compartidos. Si cada hilo tuviera su conjunto propio de recursos y jamás llegara a bloquearse, podría incrementarse el rendimiento y la escalabilidad mejoraría. Por desgracia, los hilos de un mismo servidor suelen compartir recursos.
	Todas las anteriores.
X	Ninguna de las anteriores.

12. Los servidores asincrónicos...:

	Utilizan un modelo de programación dirigido por eventos. Cierto. En un servidor asincrónico se van atendiendo las diferentes peticiones recibidas a medida que se complete el servicio de la anterior. Cada petición se gestiona como un evento.
	Pueden implantarse utilizando NodeJS y ZeroMQ. NodeJS permite implantar servidores asincrónicos en JavaScript. ZeroMQ es un middleware de comunicaciones basado en intercambio asincrónico de mensajes que puede utilizarse en NodeJS.
	Estructuran su código utilizando "*callbacks*", que son las acciones asociadas a determinados eventos. Puede estructurarse de esa manera.
	Evitan los problemas que plantean las secciones críticas. Cierto. Normalmente habrá un solo hilo de ejecución en cada proceso servidor. Así se evitan las secciones críticas, pues no hay recursos compartidos cuyo acceso concurrente deba protegerse.
X	Todas las anteriores.
	Ninguna de las anteriores.

13. Un middleware...:

	Generalmente impide que los componentes sean interoperables. Uno de los objetivos tradicionales de los middleware es facilitar la interoperabilidad entre múltiples componentes, independientemente de quien los desarrolle.
	Cuando gestione la intercomunicación estará por debajo del nivel de transporte. En un sistema distribuido la mayoría de los middleware utilizados suelen ubicarse sobre el sistema operativo. El nivel de transporte está facilitado por el propio sistema operativo; por tanto, el middleware se ubica sobre el nivel de transporte.
X	Suele respetar uno o más estándares y facilita el desarrollo de aplicaciones distribuidas. La mayoría de los middleware tienen esos objetivos.
	Obliga a utilizar una interacción cliente/servidor entre componentes. No es obligatorio. Se pueden tener middleware que faciliten un patrón de comunicación distinto al de "petición y respuesta" asumido en las interacciones cliente/servidor. Por ejemplo, "publish/subscribe". También se podría tener un middleware para proporcionar una imagen de memoria compartida, permitiendo que los procesos de diferentes ordenadores se comunicaran accediendo a objetos compartidos. Quizá no fuese escalable, pero sí suficiente para ciertos tipos de aplicación.
	Todas las anteriores.
	Ninguna de las anteriores.

14. Ejemplos de tipos de middleware y funciones que desarrollan:

	URL. Permite localizar los recursos en un sistema distribuido. Una URL es un tipo de nombre, no es un middleware.
	JavaScript. Permite desarrollar aplicaciones distribuidas. JavaScript es un lenguaje de programación. No es un middleware.
X	RPC. Proporciona un modelo de interacción cliente/servidor con transparencia de ubicación. Puede considerarse un tipo de middleware entre cuyos objetivos está el proporcionar transparencia de ubicación.
	SaaS. Gestiona la virtualización de recursos y proporciona herramientas para el desarrollo de servicios web escalables. SaaS es un modelo de servicios en la computación en la nube. No es un tipo de middleware. Además, la definición que aparece en este apartado no corresponde a un modelo SaaS sino a un PaaS.
	Todas las anteriores.
	Ninguna de las anteriores.

15. Las llamadas a procedimiento remotas:

	Se comportan exactamente igual a las llamadas a procedimiento locales El término "comportamiento" tiene dos dimensiones: la interfaz ofrecida a quien deba usar ese mecanismo y las operaciones llevadas a cabo para cumplir con su funcionalidad. Una RPC puede ofrecer una interfaz similar a la de una llamada a procedimiento local, pero la secuencia de pasos necesarios es distinta. Por tanto, no se comportan de igual manera.
	Utilizan características del procesador para pasar argumentos al servidor. Eso sucede en las llamadas locales. En una RPC se utilizarán mensajes para hacer llegar los argumentos al servidor.
X	Presentan modos de fallos diferentes que las llamadas locales. Una RPC puede fallar cuando haya problemas en la transmisión de los mensajes o cuando falle el ordenador donde se encuentre el procedimiento servidor. En una llamada local jamás se producirán esas situaciones de fallo.
	Precisan de un mecanismo de broadcast en el transporte. Podría ser necesario si se invocara un procedimiento de un servidor replicado. Sin embargo, el mecanismo RPC descrito en esta asignatura no asumía esa situación y, de manera general, no se necesita una difusión para implantar la RPC.
	Todas las anteriores.
	Ninguna de las anteriores.

16. Los estándares en computación distribuida...:

	Proporcionan una solución racional para resolver ciertos problemas.
	Facilitan la interoperabilidad.
	Proporcionan funcionalidad de alto nivel.
	Familiarizan a los programadores con las técnicas a utilizar.
X	Todas las anteriores. Las cuatro propiedades citadas se listan como atribuibles a cualquier estándar en computación distribuida en el material de teoría de TSR.
	Ninguna de las anteriores.

17. Una librería de invocación de métodos remotos...:

	Es un ejemplo de middleware. Cierto. Permite la intercomunicación de componentes en un sistema distribuido que utilice un modelo orientado a objetos.
	Se utiliza en Java RMI. Java RMI es un ejemplo de middleware de este tipo.
	Utiliza el protocolo SOAP en los servicios web. Es uno de los ejemplos citados en el material de teoría.
	Se utiliza en el estilo arquitectónico REST, tomando como base los protocolos HTTP. También se cita así en el material de teoría.
X	Todas las anteriores.
	Ninguna de las anteriores.

18. Los sistemas de objetos distribuidos...:

	Suelen ser escalables pues se pueden crear nuevos objetos en la aplicación cuando se requiera, sin ninguna limitación. Aunque sí que existe esa libertad para crear nuevos objetos cuando se considere necesario, los sistemas basados en objetos distribuidos difícilmente serán escalables. El código que habrá que ejecutar en cada componente de la aplicación realizará frecuentes llamadas a objetos remotos y eso ralentizará la ejecución.
	Ofrecen un modelo de programación asincrónico y, por tanto, fácilmente escalable. Que el sistema esté orientado a objetos no condiciona que se adopte un modelo de programación asincrónico para los servidores. Puede ser multi-hilo o asincrónico, o cualquier otro (si hubiese más). Son características independientes.
	Facilitan diseños con alta cohesión, generando componentes fácilmente reutilizables y con bajo acoplamiento. Sí que se proporciona, generalmente, alta cohesión. Sin embargo, el acoplamiento también suele ser fuerte, pues desde cualquier objeto se tendrá acceso a otros objetos y se utilizarán sus métodos para modificar su estado. Con ello se ofrece una imagen similar a la de una "memoria compartida". Como esto suele estar acompañado por transparencia de ubicación... al final el sistema difícilmente podrá escalar, pues se estarán realizando continuamente invocaciones a objetos remotos.
	Minimizan la contención y la generación de secciones críticas. Esto también favorece su escalabilidad. Cada objeto, como podrá ser invocado desde múltiples procesos, será gestionado como un recurso compartido. Por ello, si se admite concurrencia dentro del proceso que lo mantenga, los métodos que modifiquen sus atributos se convertirán en una sección crítica que habrá que proteger de manera adecuada. Eso reduce su escalabilidad.
	Todas las anteriores.
X	Ninguna de las anteriores.

19. Los middleware de mensajería...:

	Proporcionan en algunos casos un modelo de interacción asincrónico, altamente escalable. Varios patrones de interacción implantados en ZeroMQ proporcionan ejemplos de gestión asincrónica (pub/sub, dealer, router...). La asincronía es clave para mejorar la escalabilidad.
	Permiten que el programador desarrolle sus aplicaciones sin recurrir a la compartición de recursos. Esto favorece la escalabilidad. El uso de memoria compartida entre las actividades de una aplicación distribuida conlleva la aparición de secciones críticas y la necesidad de sincronización para protegerlas. La intercomunicación basada en mensajes permite evitar esas situaciones.
	Puedes usarse para la implantación de servidores replicados. Las réplicas de un determinado servidor necesitarán comunicarse intercambiando mensajes para llegar a respetar algún modelo de consistencia. Un middleware de "mensajería" permite implantar estos componentes.
	Incluyen a ZeroMQ como ejemplo de middleware persistente y sin gestor ("*brokerless*"). ZeroMQ mantiene los mensajes en memoria principal mientras no sea posible entregarlos a su destinatario. Con ello se garantiza cierto grado de persistencia en la comunicación. Además, no necesita ningún proceso gestor externo para realizar la intercomunicación. Basta con que los agentes a intercomunicar usen la biblioteca ZeroMQ.
X	Todas las anteriores.
	Ninguna de las anteriores.

20. Un middleware o sistema de mensajería se considera persistente cuando...:

	No bloquea a los emisores de mensajes. De esta manera mejora la escalabilidad de las aplicaciones. Esa es la definición de comunicación asincrónica. Se pedía comunicación persistente.
	Utiliza un servicio de nombres para encontrar los procesos servidores a los que debe entregar los mensajes. Así se logra transparencia de replicación. El servicio de nombrado no tiene ninguna relevancia al considerar si un sistema de comunicación es persistente o no.
X	Los mensajes se mantienen temporalmente en buffers gestionados por el "canal". El receptor no tiene por qué estar activo cuando el emisor envía el mensaje. Esta es la definición y la propiedad principal de la comunicación persistente.
	Se apoya en los niveles de red y transporte para realizar el encaminamiento y la gestión del canal de comunicación. Los servicios de encaminamiento y la gestión de conexiones tampoco tienen importancia a la hora de considerar la persistencia.
	Todas las anteriores.
	Ninguna de las anteriores.

21. Ejemplos de middleware:

	Java RMI. Middleware de invocación de métodos remotos propio de Java.
	ZeroMQ. Middleware de comunicaciones.
	RabbitMQ Middleware de comunicaciones.
	JINI. Middleware orientado a objetos, inicialmente desarrollado por Sun.
X	Todas las anteriores.
	Ninguna de las anteriores.

22. La escalabilidad horizontal...:

	Consiste en el reemplazo de un componente por otro con mayor capacidad de servicio. Esa es la definición de escalabilidad vertical.
	Implica que el sistema sea escalable y adaptable. Esa adaptabilidad será dinámica (reaccionando a las variaciones de carga) y autónoma (sin intervención humana). Esa es la definición de elasticidad.
	Garantiza la robustez del sistema. Puede mejorar el rendimiento, pero no influye para nada en la seguridad, por ejemplo. Por tanto, no garantiza la robustez. Para ello debería garantizar disponibilidad, fiabilidad, mantenibilidad y seguridad.
	Requiere un subsistema de monitorización (de la carga soportada y el rendimiento obtenido) y otro de actuación que automatice la reconfiguración. Esos subsistemas resultan necesarios para construir un sistema adaptable dinámicamente (elástico, en caso de que también fuese escalable).
	Todas las anteriores.
X	Ninguna de las anteriores.

23. Mecanismos para incrementar la escalabilidad de tamaño de un servicio:

	Reducir o eliminar las necesidades de sincronización entre los agentes que implanten el servicio. Si no se necesita sincronizar, no habrá bloqueos. Mejora la escalabilidad.
	Utilizar algoritmos descentralizados. Los algoritmos descentralizados permiten tomar decisiones a partir de la información local y soportan los fallos de cualquier componente. Por tanto, también reducen la necesidad de sincronización. Mejora la escalabilidad.
	Delegar todo el cómputo posible a los agentes clientes. Esto reduciría la carga en los servidores y permitiría atender a un mayor número de clientes. Mejora la escalabilidad.
	Replicar los componentes servidores, utilizando el modelo de consistencia más relajado que admita el servicio. Al tener los servicios replicados se podrá atender a un número más alto de clientes. Una consistencia relajada reduce las necesidades de sincronización entre réplicas. Mejora la escalabilidad.
X	Todas las anteriores.
	Ninguna de las anteriores.

24. Pueden ser causas de contención...:

	Adoptar un modelo de programación asincrónica. Reduce o elimina la necesidad de sincronización. No causará contención.
	Utilizar algoritmos descentralizados. Permite tomar decisiones locales. No necesita sincronización. No causará contención.
	Emplear un modelo de replicación pasiva. Es más escalable que el modelo de replicación activa. De entre los modelos de replicación, éste es el más recomendable. Permite realizar más fácilmente un equilibrado de la carga. Reduce la contención en caso de sistemas replicados.
X	Usar herramientas de sincronización. Introducirán bloqueos. Son causas de contención.
	Todas las anteriores.
	Ninguna de las anteriores.

25. La elasticidad...:

	Es una propiedad de cualquier clase de aplicación (tanto distribuida como no distribuida). La elasticidad implica escalabilidad y adaptabilidad a la carga. La combinación de ambas propiedades sólo tiene sentido en un sistema distribuido.
X	Requiere escalabilidad y adaptabilidad. Así fue definida.
	Sólo tiene sentido en los sistemas IaaS. Se puede exigir a cualquier tipo de sistema o aplicación distribuidos.
	Se obtiene al utilizar un modelo de replicación activa. El modelo de replicación activa ofrece una escalabilidad muy limitada. Por ello, no es la mejor base para desarrollar un sistema elástico.
	Todas las anteriores.
	Ninguna de las anteriores.

26. Se han visto tres dimensiones de escalabilidad...:

	Rendimiento, Persistencia y Disponibilidad. La persistencia no tiene nada que ver con la escalabilidad. Aunque una alta disponibilidad y un buen rendimiento sean aconsejables en un sistema distribuido escalable, no son las dimensiones que determinan si un sistema distribuido es escalable.
	Consistencia, Disponibilidad y Tolerancia al particionado de la red. Estas son las tres propiedades analizadas en el teorema CAP. De nuevo, no son las dimensiones de escalabilidad.
X	Tamaño, Distancia y Administrativa. Éstos sí son los ejes o dimensiones que determinan si un sistema distribuido puede considerarse o no escalable.
	Vertical, Horizontal y Oblicua. Otra alternativa para clasificar los diferentes tipos de escalabilidad es distinguir entre escalabilidad vertical (la que se obtiene reemplazando componentes) y la horizontal (obtenida añadiendo nuevos nodos al sistema). Sin embargo, en esta segunda clasificación no aparece ninguna "escalabilidad oblicua".
	Todas las anteriores.
	Ninguna de las anteriores.

27. La escalabilidad vertical...:

X	Incrementa la capacidad de servicio reemplazando componentes. Ésta es la definición enunciada en las clases de teoría.
	Mejora la disponibilidad de los servicios. La escalabilidad vertical permite mejorar el rendimiento de una sola máquina. Como no se añaden otros nodos al sistema, la disponibilidad de los servicios ejecutados no se incrementa. Si falla el nodo donde se ejecute ese servicio, el servicio queda indisponible.
	Incrementa la capacidad de servicio añadiendo nuevos nodos, recursos o componentes. Eso ocurre en la escalabilidad horizontal, no en la vertical.
	Mejora la seguridad de los servicios. Para mejorar la seguridad en un sistema distribuido hay que emplear otros mecanismos. La escalabilidad vertical no influye para nada en la seguridad.
	Todas las anteriores.
	Ninguna de las anteriores.

28. Un sistema escalable...:

	Tendrá que eliminar sus puntos de contención. Si hubiese puntos de contención que introdujeran "cuellos de botella" el sistema dejaría de incrementar su rendimiento y capacidad de servicio al aumentar la carga que esté soportando. Si eso sucediera, no sería escalable.
	Tendrá que ser robusto. De nada serviría tener un sistema capaz de incrementar su capacidad de servicio en función de la carga introducida si no fuera fiable, se garantizara su disponibilidad, se recuperase lo antes posible de las situaciones de fallo (mantenibilidad) y fuera seguro.
	Se podrá aumentar su capacidad de servicio cuando sea necesario. Ésta es su característica principal.
	No siempre tendrá una consistencia fuerte. Cuantos más nodos participen en el sistema distribuido (y el número será cada vez mayor si se emplea escalabilidad horizontal y la carga que soporta el sistema sigue creciendo) más difícil será mantener una consistencia fuerte entre las réplicas de cada elemento gestionado por el sistema.
X	Todas las anteriores.
	Ninguna de las anteriores.

29. El "*sharding*" o particionado de una base de datos distribuida (utilizado, por ejemplo, en MongoDB)...:

	Es una implantación del mecanismo de reparto de datos que potencia la escalabilidad de tamaño.
	Es una implantación del mecanismo de reparto de tareas que potencia la escalabilidad horizontal.
	Consigue aumentar el grado de concurrencia sin necesitar mecanismos de sincronización.
	No siempre exige mantener múltiples réplicas para un mismo elemento de la base de datos.
X	Todas las anteriores. En el "sharding" se reparten los datos mantenidos en la BD entre múltiples ordenadores (primer apartado), con lo que las tareas de gestión de estos datos también estarán repartidas entre múltiples procesos (segundo apartado). Ambas facetas mejoran la escalabilidad horizontal (escalabilidad de tamaño). Al efectuar este reparto cada proceso podrá atender, por su cuenta, una petición diferente. Así se aumenta el grado de concurrencia sin necesitar mecanismos de sincronización, pues no hay recursos compartidos entre los diferentes procesos servidores (tercer apartado). Por último, el "sharding" permite que haya un solo servidor en cada fragmento de la base de datos (cuarto apartado), aunque generalmente hay varios (utilizando replicación) para mejorar la disponibilidad.
	Ninguna de las anteriores.

30. Con el modelo activo de replicación...:

	Se garantiza la robustez de un servicio. La replicación (tanto si es pasiva como activa) mejora la disponibilidad y la mantenibilidad, pero no hace nada para mejorar la seguridad. Por tanto, la robustez no está garantizada. La fiabilidad dependerá de los componentes utilizados para implantar el sistema o servicio.
X	Se puede soportar el modelo de fallos de omisión general. Para soportar los modelos de fallos más severos (omisión general, arbitrarios) se suele recurrir a una comparativa entre las salidas generadas por las diferentes réplicas, seleccionando la respuesta mayoritaria. El modelo activo permite este tipo de gestión, pues múltiples réplicas gestionan directamente cada petición y responden todas al cliente, cuyo "stub cliente" realiza la comparativa antes de devolver el control al proceso cliente.
	No existirá riesgo de inconsistencia cuando se ejecuten operaciones no deterministas. Al contrario, si las operaciones fueran no deterministas cada réplica podría generar un resultado diferente. Si ese resultado afecta al estado mantenido en cada réplica, se llegaría a tener un estado diferente en cada réplica. Es decir, se generarían inconsistencias entre el estado de las réplicas.
	No se necesita ningún tipo de sincronización entre las réplicas para obtener una consistencia secuencial entre ellas. Aunque las réplicas no se comunican entre sí una vez han recibido cada petición, sí que se necesita sincronizarlas para llegar a una consistencia secuencial. Esa sincronización consiste en utilizar un protocolo de difusión de los mensajes que emitan los clientes para que sean entregados en el mismo orden en todas las réplicas destinatarias (difusión de orden total FIFO). Un protocolo de ese tipo requiere que los diferentes procesos decidan en qué orden deben entregarse esos mensajes "concurrentes". Para ello se necesita intercambiar un buen número de mensajes entre todos los participantes. Esto requiere ancho de banda y tiempo, ralentizando la ejecución.
	Todas las anteriores.
	Ninguna de las anteriores.

31. En el modelo pasivo de replicación...:

	Todas las réplicas de un servicio tienen un mismo rol. No es así. Hay una réplica primaria y todas las demás son réplicas secundarias. Son dos roles diferentes.
	Se puede soportar el modelo de fallos arbitrarios (o fallos "bizantinos"). Como sólo existe una réplica primaria que gestiona directamente cada petición, no se puede "votar" por un resultado. Así no puede soportarse un modelo de fallos arbitrarios.
	Se garantiza la seguridad (en sus dos acepciones: "security"/"safety") del servicio ofrecido. La seguridad no puede garantizarse empleando replicación. Los mecanismos a utilizar son otros (cifrado, uso de certificados, políticas de seguridad uniformes y bien implantadas…)
X	Se requiere menos cómputo que en el modelo activo cuando las operaciones implican un servicio prolongado pero modifican poco estado. Ésa es su ventaja principal al compararlo con el modelo activo. Aunque la operación requerirá mucho tiempo, ese tiempo solo se invierte en la réplica primaria. Al modificar poco estado, costará poco transferir ese estado a las réplicas secundarias y aplicarlo allí. En el modelo activo, todas las réplicas habrían tenido que ejecutar localmente la operación y les habría costado más a todas ellas. Con ello, utilizando replicación activa un solo servicio podría saturar a cierto conjunto de N nodos. Por el contrario, con replicación pasiva es posible desplegar N servicios en ese conjunto de nodos, ubicando la réplica primaria de cada uno en cada nodo. Un ejemplo de operaciones del tipo comentado en el primer párrafo es la ejecución de sentencias en una base de datos relacional. Sobre todo si sólo afectan a unos pocos registros y requieren el acceso a disco para recuperarlos y volverlos a guardar. Cada sentencia puede ejecutarse localmente en la réplica primaria. Solo se transmitirán las modificaciones a las réplicas secundarias cuando la transacción que las englobe finalice satisfactoriamente. El mensaje trasmitido será pequeño y el tiempo necesario para aplicar estos cambios en las réplicas secundarias podrá ser entre 10 y 200 veces menor a la ejecución local.
	Todas las anteriores.
	Ninguna de las anteriores.

32. **Seleccionar, de entre todas las alternativas propuestas, aquella que ofrezca la lista más extensa de modelos de consistencia satisfechos en la siguiente ejecución. No se admitirá ninguna alternativa con algún modelo de consistencia no respetado por la ejecución:**
P1:W(x)1, P2:W(x)2, P3:R(x)1, P3:W(x)3, P2:W(x)4, P4:R(x)2, P4:R(x)3, P4:R(x)1, P5:R(x)2, P4:R(x)4, P5:R(x)3, P5:R(x)1, P5:R(x)4

	Secuencial, caché, procesador, causal, FIFO. En esta ejecución se observa que: 1) P2 escribe primero el valor 2 y después el valor 4. No hay ningún proceso más que escriba dos valores. 2) P3 lee el valor 1 antes de escribir el valor 3. Por ello, si las ejecuciones mantuvieran el orden FIFO, en todos los lectores aparecerá antes el valor 2 que el 4. Si las ejecuciones mantuviesen el orden causal, el valor 1 debería aparecer antes que el 3. En esta ejecución los procesos 4 y 5 actúan como lectores. Ambos obtienen la misma secuencia: 2, 3, 1, 4. Por tanto, al compartir todos la misma secuencia sobre la misma variable (solo se utiliza la "x" en este ejemplo), la consistencia es caché. También se respeta que sea FIFO. La unión de caché y FIFO genera la consistencia "procesador". Sin embargo, el valor 3 es leído antes que el valor 1. Por tanto, no se respeta la consistencia causal. Por no ser causal, tampoco podrá ser secuencial. Resumen: las consistencias que se cumplen son **caché, FIFO y procesador.**
	Caché.
	Causal, FIFO.
	Procesador, FIFO.
X	Procesador, caché, FIFO.
	FIFO.

33. **Seleccionar, de entre todas las alternativas propuestas, aquella que ofrezca la lista más extensa de modelos de consistencia satisfechos en la siguiente ejecución. No se admitirá ninguna alternativa donde aparezca un modelo de consistencia no respetado por la ejecución:**
P1:W(x)1, P2:W(x)2, P3:R(x)1, P3:W(x)3, P2:W(x)4, P4:R(x)1, P4:R(x)3, P4:R(x)2, P5:R(x)2, P4:R(x)4, P5:R(x)1, P5:R(x)3, P5:R(x)4

	FIFO. Los cinco primeros accesos de esta ejecución son idénticos a los vistos en la cuestión 32. Sin embargo, hay diferencias en los procesos lectores. P4 lee 1, 3, 2 y 4. P5 lee 2, 1, 3 y 4. Ambas secuencias de lectura respetan la condición FIFO (2 antes que 4) y causal (1 ante que 4) explicadas en la solución de la actividad 32. Por no coincidir las secuencias de lectura en los procesos 4 y 5, la ejecución no será caché, ni podrá ser secuencial. Si no es caché, tampoco podrá ser "procesador". Por tanto, la solución correcta consistía en citar las consistencias **FIFO y causal**.
	Procesador, FIFO, caché.
X	Causal, FIFO.
	Causal, caché.
	Procesador, causal.
	Procesador, causal, caché, FIFO.

34. **Seleccionar, de entre todas las alternativas propuestas, aquella que ofrezca la lista más extensa de modelos de consistencia satisfechos en la siguiente ejecución. No se admitirá ninguna alternativa donde aparezca un modelo de consistencia no respetado por la ejecución:**
P1:W(x)1, P2:W(x)2, P3:W(x)3, P2:W(x)4, P4:R(x)3, P4:R(x)1, P4:R(x)2, P5:R(x)3, P4:R(x)4, P5:R(x)1, P5:R(x)2, P5:R(x)4

	Caché. Ninguno de los procesos escritores llega a leer los valores escritos por los demás procesos. Sólo P2 escribe dos valores, en orden 2, 4. Por tanto, las ejecuciones serán FIFO si en ellas aparece un 2 antes que un 4. En ese caso, también serán causales. P4 y P5 actúan como procesos lectores. Obtienen ambos la secuencia 3, 1, 2, 4. Por ser la misma secuencia en todos los lectores, se cumple la consistencia caché. Por ser FIFO y causal, será también secuencial. Al cumplirse tanto la caché como la FIFO, también se cumple la consistencia "procesador". Resumen: se cumplen las consistencias **caché, FIFO, procesador, causal y secuencial**.
	FIFO.
	Procesador, FIFO, caché.
	Causal, FIFO.
X	Secuencial, causal, procesador, FIFO, caché.
	Causal, caché.

35. Relación entre defectos, errores y fallos:

	Podrá haber fallos que no estén causados por ningún defecto. Para que haya un fallo tiene que haberse dado previamente algún error. A su vez, los errores no se generan si no están precedidos por algún defecto. Por tanto, el primer paso es la ocurrencia de un defecto. Si no existe ningún tratamiento, éste generará un error. Si tampoco hay una gestión que los resuelva, causarán fallos. Transitivamente, todos los fallos están causados por uno o más defectos.
X	Se generará un fallo si hay errores y no existe redundancia en el componente que ha sufrido esos errores.
	La replicación resuelve todas las situaciones de error, para que no se conviertan en fallos. La replicación es la gestión recomendada para que no haya fallos. Sin embargo, no todas las situaciones de error se podrán resolver mediante replicación, si ésta no es suficientemente cuidadosa. La replicación debe estar acompañada por mecanismos de diagnóstico adecuados y por ciertos protocolos que aseguren la corrección de las réplicas que hayan superado esa situación de error. Por ejemplo, cuando no se haya desplegado un número suficiente de réplicas o todas ellas dependan de la misma fuente de errores, los errores seguirán provocando fallos a pesar de la replicación. En esa situación, un error podría afectar a todas las réplicas de un determinado servicio y ninguna de ellas continuaría con su trabajo, dejando ese servicio indisponible.
	Con un mecanismo de diagnóstico adecuado y un tratamiento correcto se puede evitar que las situaciones de fallo provoquen errores. Ocurre lo contrario. Con esas gestiones se podría evitar que las situaciones de error se conviertan en fallos.
	Todas las anteriores.
	Ninguna de las anteriores.

36. Ejemplos de modelo de fallos:

	Activo, pasivo. No son ejemplos de modelos de fallos, sino de modelos de replicación.
	Canal y enlace, red, transporte. Los cuatro términos mencionados guardan relación con las comunicaciones pero no son modelos de fallos. "Canal y enlace" son dos términos sinónimos. "Caída y enlace" es el nombre de un modelo de fallos.
X	Parada, caída, omisión general. Estos son tres de los modelos explicados. Hay otros: omisión de recepciones, omisión de envíos, arbitrarios…
	Mantenibilidad, partición primaria, consistencia secuencial. No son ejemplos de modelos de fallos. La "mantenibilidad" es una propiedad o atributo que deben cumplir los sistemas robustos. La "partición primaria" es un tipo de gestión para afrontar las situaciones de particionado de la red. La "consistencia secuencial" es un modelo de consistencia aplicable a sistemas replicados.
	Todas las anteriores.
	Ninguna de las anteriores.

37. Atributos que definen un sistema robusto:

X	Fiabilidad, mantenibilidad, seguridad, disponibilidad. Estos son los cuatro atributos que aparecen en su definición. "Seguridad" tenía dos acepciones: evitación de desastres, por una parte, y confidencialidad y protección, por otra.
	Consistencia, atomicidad, aislamiento, durabilidad. Estas son las cuatro garantías que debe ofrecer toda transacción aceptada.
	Escalabilidad, elasticidad, adaptabilidad, eficiencia. Estos son atributos que guardan relación con el rendimiento de un sistema, pero no definen su robustez.
	Corrección, eficacia, facilidad de uso, vivacidad. Son cuatro ejemplos más de atributos aconsejables para cualquier sistema informático, pero no definen su robustez.
	Todas las anteriores.
	Ninguna de las anteriores.

A continuación presentamos el módulo nodejs, shared.js.

```
// shared memory module
//
var zmq = require('zmq');
var id = "";
var pb = zmq.socket('pub');
var sb = zmq.socket('sub');
var local = {};

sb.subscribe("");
pb.bind('tcp://*:8888');

function join(nodes) {
     nodes.forEach(function (ep) {
       sb.connect(ep);
     });
}
function myID(nid) {
     id = nid;
}

sb.on('message', function (name, value) {
     local[name] = value;
});

// initialization
exports.init = function(nodes, id) {
    join(nodes);
    myID(id);
}
// Write function
exports.W = function (name, value) {
     pb.send([name, value]);
     local[name] = value;
     console.log("W" + id + "(" + name + ")" + value + ":");
};
// Read function
exports.R = function (name) {
     console.log("R" + id + "(" + name + ")" + local[name] + ":");
     return local[name];
};
```

Este módulo implementa operaciones *Read* (**R)** y *Write* (**W**) sobre un conjunto de variables compartidas por un grupo de procesos.
Cuando un proceso quiere integrarse en el grupo de compartición, realiza un `require('shared.js')`, y lo inicializa con las URLs de los endpoints de los otros procesos en el grupo, y un identificador para sí mismo.
`shared.js` imprime un log con las operaciones **R**/**W** realizadas, incluyendo los valores. Esto representa la secuencia de eventos R/W en el proceso en el orden en que son producidos.
Suponer que lanzamos cuatro procesos, con ids 1,2,3,4, compartiendo la variable "X", y los lanzamos sobre la misma consola de comandos. Los eventos de todos los procesos, a medida que son impresos por `shared.js`, se van a mezclar en la salida de la consola. Suponer que ningún mensaje pub se pierde.
Responder a las siguientes dos preguntas en el contexto anterior.

38. Seleccionar la secuencia de reads/writes que puede observarse en la consola:

	R1(X)2:W1(X)4:R2(X)4:W3(X)67:R4(X)undefined: Se asume que la variable X no estaba inicializada. Por ello, si la primera operación en una ejecución fuese una lectura, como en este caso, el proceso correspondiente obtendría el valor "undefined". Eso no ocurre en esta ejecución. Por tanto, no sería posible observar esta secuencia. No tiene sentido obtener un 2 antes de que sea escrito por algún proceso. De hecho, ninguno de ellos llega a escribir ese valor.
X	R1(X)undefined:W1(X)4:R2(X)4:W2(X)67:R4(X)4: En este caso la primera lectura sí que obtiene un valor justificable, pues no se había realizado aún una escritura. A su vez, el proceso 4 lee un valor 4 después de que el proceso 2 haya escrito el valor 67. Eso también es admisible, pues puede que el envío realizado por el proceso 2 mediante el socket "pub" utilizado todavía no haya llegado al proceso 4.
	R2(X)2:W2(X)4:R1(X)4:R3(X)67:R4(X)undefined: Se da una paradoja similar a la de la primera ejecución: leer un valor antes de que éste haya sido escrito, asumiendo que llegará a escribirse en algún momento futuro.
	R1(X)2:W2(X)2:R1(X)4:W3(X)4:R4(X)5: Se da una paradoja similar a la de la primera ejecución: leer un valor antes de que éste haya sido escrito.
	Todas las anteriores.
	Ninguna de las anteriores.

39. El módulo shared.js implementa la consistencia…:

	Secuencial Para que fuera secuencial, el mecanismo de difusión de las escrituras debería preocuparse por llegar a un acuerdo sobre un mismo orden de entrega en todos los procesos destinatarios. Además, ese orden debería respetar el orden en que cada proceso haya realizado sus escrituras localmente (orden FIFO) y el orden causal. Los sockets "pub/sub" de ZeroMQ no se preocupan por garantizar un orden total en los mensajes propagados.
	Procesador La consistencia procesador requiere entrega en orden total (consistencia caché) y que ese orden total respete también el orden FIFO, aunque no el causal. Como ZeroMQ no proporciona orden total en los sockets "pub/sub", la consistencia procesador no está garantizada.
	Causal Para garantizar la consistencia causal suele ser necesario un etiquetado de los mensajes difundidos con los relojes vectoriales asociados a sus eventos de envío. En este caso no se ha llegado a realizar esa gestión.
X	FIFO La consistencia FIFO puede garantizarse si los mensajes se entregan en cada destinatario en el orden en que los envió su proceso emisor. Hay libertad para intercalar de diferentes maneras los mensajes difundidos por diferentes procesos. Eso puede llegar a respetarse con sockets "pub/sub".
	Todas las anteriores.
	Ninguna de las anteriores.

40. Objetivos generales de la replicación:

	Aumentar la disponibilidad. Cuantas más réplicas haya, más difícil será que un servicio no esté disponible cuando haya errores.
	Aumentar el rendimiento. Si se incorporan más réplicas a un servicio se podrá repartir la carga entre todas ellas y mejorará el rendimiento general.
	Disminuir el tiempo de servicio de las peticiones. Al repartir la carga entre todas las réplicas y aumentar el número de réplicas, cada proceso servidor recibirá un porcentaje menor de la carga global y podrá terminar más rápidamente el servicio de cada petición (pues el grado de concurrencia será menor y aparecerán menos conflictos entre las actividades que se ejecuten en una misma réplica).
	Implementar la escalabilidad horizontal. Como consecuencia de todo lo explicado en los tres apartados anteriores, el servicio replicado mejora su escalabilidad. La escalabilidad obtenida mediante replicación se conoce como escalabilidad horizontal.
X	Todas las anteriores.
	Ninguna de las anteriores.