Tema:	Corrección de la evaluación
Unidad de Organización Curricular:	PROFESIONAL
Nivel y Paralelo:	Sexto A
Alumnos participantes:	Lozada López Jonathan Israel
Asignatura:	Aplicaciones Distribuidas
Docente:	Ing. José Rubén Caiza Caizabuano

Pregunta 1
Explique la diferencia fundamental entre la comunicación basada en Remote Procedure Call (RPC) y la comunicación basada en mensajes en sistemas distribuidos. Respuesta revisada:

RPC: El cliente invoca procedimientos remotos como si fueran locales, normalmente de forma síncrona y fuertemente acoplada a la disponibilidad del servidor.
Mensajes: Los procesos se comunican enviando mensajes gestionados por colas u otros mecanismos (brokers), lo que posibilita mayor asincronía, desacoplamiento temporal y tolerancia a fallos (reintentos, buffer en cola).

Pregunta 2
Describa las principales diferencias entre la comunicación síncrona (por ejemplo, utilizando gRPC o REST) y la comunicación asíncrona (por ejemplo, utilizando colas de mensajes o Apache Kafka) en sistemas distribuidos, haciendo énfasis en acoplamiento, escalabilidad y tolerancia a fallos. Respuesta revisada:

Síncrona: El cliente espera la respuesta del servicio en la misma interacción. Esto genera acoplamiento más fuerte y dependencia inmediata (ambos deben estar disponibles), lo que limita la escalabilidad y reduce la tolerancia a fallos si el proveedor está caído o lento.
Asíncrona: Desacopla temporalmente a productor y consumidor; los mensajes se almacenan y se procesan después. Esto facilita mayor escalabilidad y tolerancia a fallos (reintentos, buffering, procesamiento paralelo), aunque introduce mayor complejidad en el rastreo de mensajes, la observabilidad y el manejo de idempotencia/orden.

Pregunta 3
Describe los diferentes modelos de consistencia (consistencia fuerte, consistencia débil, consistencia eventual) y brinda un ejemplo de situación en la que utilizarías la consistencia eventual en lugar de la consistencia fuerte. Respuesta revisada:

Consistencia fuerte: Todas las lecturas reflejan la última escritura de forma inmediata y global; ofrece un orden único y actual.
Consistencia débil: No garantiza que una lectura vea la última escritura; se prioriza disponibilidad/latencia sobre orden inmediato.
Consistencia eventual: Los nodos pueden ver valores distintos temporalmente, pero convergen si no hay nuevas escrituras.
Ejemplo de uso de consistencia eventual: Contadores de “me gusta” en una red social o la propagación de un catálogo de productos entre regiones; la precisión inmediata no es crítica y se priorizan latencia baja, disponibilidad y escalabilidad.

Pregunta 4
Compare brevemente la arquitectura monolítica con la basada en microservicios en cuanto a escalabilidad. Respuesta revisada:

Monolítica: Para escalar una parte se debe escalar toda la aplicación (replicar el monolito), lo que puede ser ineficiente si solo hay “hotspots” en módulos específicos.
Microservicios: Cada servicio se escala de manera independiente según su demanda (autoescalado, particionado por servicio), logrando mejor uso de recursos y elasticidad, a costa de mayor complejidad operativa.

Pregunta 5

¿Por qué es importante la sincronización en sistemas distribuidos y qué desafíos implica lograr un orden adecuado en la ejecución de procesos? Respuesta revisada:

Importancia: Garantiza un orden coherente de eventos y consistencia de datos entre procesos, compartiendo una referencia temporal útil para coordinar acciones.
Desafíos: Latencia de red, ausencia de un reloj global confiable y deriva de relojes; dificultad para establecer orden global/causal. Requiere técnicas como relojes lógicos (Lamport, vectores), orden total/causal y, cuando se necesita acuerdo, algoritmos de consenso (p. ej., Paxos, Raft), que añaden complejidad y costo en presencia de fallos.

Pregunta 6

- Proyecto: Crear un ASP.NET Core Web API (dotnet new webapi). Estructura típica: Controllers (por ejemplo, UsuariosController), Services (interfaces y lógica de negocio para SOA), Models/DTOs, Domain/Entities, Infrastructure/Persistence (DbContext, repositorios).
- Controladores: Exponen endpoints REST (GET/POST/PUT/DELETE), reciben/retornan DTOs, delegan la lógica a servicios inyectados por DI.
- Pruebas locales: Ejecutar con dotnet run y verificar en Swagger en https://localhost:<puerto>/swagger.
- Publicación en Linux:
    - dotnet publish -c Release -r linux-x64 -o ./publish (opcionalmente --self-contained true para no depender del runtime instalado).
    - Copiar la carpeta publicada al servidor (scp/rsync).
    - Configurar un servicio systemd que ejecute dotnet /ruta/MiApi.dll, establecer variables de entorno y usuario de servicio.
    - Opcional: Nginx como reverse proxy, certificados TLS, y abrir el puerto correspondiente en el firewall.
    -

Pregunta 7

El middleware es un conjunto de servicios intermedios que facilitan la comunicación e integración entre aplicaciones distribuidas. Su objetivo principal es proporcionar transparencia (de ubicación, acceso, fallos, concurrencia), ocultando la complejidad del sistema operativo y la red, y ofreciendo capacidades comunes como serialización, seguridad, descubrimiento, balanceo y transacciones para que los desarrolladores se enfoquen en la lógica de negocio.

Pregunta 8

Se recomiendan APIs ligeras (REST/gRPC) porque promueven bajo acoplamiento y modularidad entre microservicios.

- Beneficios: Independencia de despliegue, escalabilidad por servicio, interoperabilidad (HTTP/Protobuf), menor latencia y sobrecarga, facilidad de mantenimiento y versionado controlado, posibilidad de caching y gateways.
- Desafíos: Orquestación y coreografía de flujos, trazabilidad y correlación de solicitudes (distributed tracing), gestión de versiones y compatibilidad hacia atrás, seguridad (authn/z, mTLS), manejo de errores e idempotencia, evitar “chattiness” y diseñar contratos estables (contract testing).

Pregunta 9

- Configurar publicación: dotnet publish -c Release -r linux-x64 -o ./publish (definir runtime y, si se desea, --self-contained).
- Traslado: Copiar ./publish al servidor Linux (scp/rsync) y asegurarse de permisos/usuario de ejecución.
- Firewall y red: Abrir el puerto de la API o configurar Nginx como reverse proxy con TLS.

- systemd:
  - Crear /etc/systemd/system/miapi.service apuntando a dotnet /ruta/MiApi.dll y variables de entorno.
  - sudo systemctl daemon-reload; sudo systemctl enable --now miapi
  - Verificar estado y logs con systemctl status y journalctl -u miapi.
- Opcional: Variables de entorno para conexiones, logging, y health checks.

Pregunta 10

El middleware contribuye a la transparencia al ocultar detalles de la red y de la distribución:

- Transparencia de ubicación: el cliente no necesita saber dónde corre el servicio.
- Transparencia de acceso: unifica protocolos/formatos (p. ej., HTTP+JSON/Protobuf).
- Transparencia de réplica: oculta múltiples réplicas detrás de balanceo.
- Transparencia de fallos: reintentos, timeouts, circuit breakers, y recuperación.
- Transparencia de concurrencia: coordina acceso a recursos y orden de mensajes. Mediante servicios como descubrimiento, enrutamiento, serialización, seguridad y balanceo, el middleware hace que el sistema se perciba como uno solo, pese a estar distribuido.

Pregunta 11

RPC es un mecanismo para invocar procedimientos en otra máquina como si fueran locales. La transparencia se logra mediante stubs/proxies en el cliente y skeletons en el servidor: el stub empaqueta (marshalling/serialización) los parámetros, envía el mensaje por la red y espera la respuesta; el skeleton deserializa, ejecuta el procedimiento remoto y devuelve el resultado. Así se ocultan detalles de red, ubicación y transporte, presentando una llamada con la misma firma que una local.

Pregunta 12

Un productor publica eventos en un topic de Kafka; los brokers almacenan esos eventos en particiones en un log append-only. Los consumidores, organizados en consumer groups, leen en paralelo; cada consumidor procesa una o varias particiones y confirma offsets para asegurar relectura controlada y tolerancia a fallos. Es recomendable en escenarios de streaming y análisis en tiempo real, integración event-driven entre sistemas desacoplados, ingesta de grandes volúmenes (telemetría, logs), y pipelines ETL con alta disponibilidad y escalabilidad horizontal.

Pregunta 13

- Comunicación síncrona: el emisor se bloquea esperando que el receptor procese y responda; fuerte acoplamiento temporal y mayor impacto de latencia/fallos.
- Comunicación asíncrona: el emisor envía y continúa; la respuesta (si existe) llega después por otro canal/mecanismo; menor acoplamiento, mayor concurrencia, requiere correlación e idempotencia.
- Invocación remota (RPC): abstracción que ejecuta un procedimiento en otro nodo como si fuera local; típicamente espera la respuesta del procedimiento remoto (suele ser síncrona), ofreciendo mayor transparencia en la llamada gracias a stubs y marshalling, aunque puede implementarse en forma asíncrona.

Pregunta 14

- Monolítica: un solo procesador/máquina ejecuta todo el programa; componentes fuertemente integrados y comunicación por memoria/llamadas internas; hardware: una máquina estándar.
- Paralela: varios procesadores/núcleos fuertemente acoplados ejecutan partes del mismo programa en paralelo; comunicación rápida vía memoria compartida o interconexiones de alta velocidad; hardware: SMP/NUMA, multinúcleo o GPU.

- Distribuida: múltiples procesadores en máquinas distintas, débilmente acopladas, cooperan vía red (LAN/Internet), con latencia y fallos parciales; hardware: varios equipos conectados en red.

Pregunta 15

Caso práctico: en un clúster de base de datos replicada (p. ej., un almacenamiento de logs con Raft como etcd), el líder coordina escrituras, ordena entradas del log y replica a seguidores para mantener consistencia.

- Función del líder: secuenciar operaciones, gestionar replicación y servir escrituras/lecturas coherentes.
- Manejo de fallos: los seguidores detectan ausencia del líder por timeouts/heartbeats, inician una reelección (Raft/Paxos), forman quórum y eligen un nuevo líder. Durante la elección se restringen operaciones críticas para evitar inconsistencia. Una vez elegido, el nuevo líder anuncia su término/estado y los clientes redirigen sus solicitudes; el sistema se recupera sin intervención manual.
-

Pregunta 18

Ventajas:

- Escalabilidad y ejecución concurrente en varios nodos.
- Mayor confiabilidad y tolerancia a fallos al no depender de un solo nodo. Desventajas:
- Mayor complejidad: sincronización/consistencia, control de acceso y observabilidad distribuidas.
- Latencia y fallos de red: propagación de información más lenta y manejo de mensajes perdidos/duplicados.
-

Pregunta 19

Modelo Cliente-Servidor: Arquitectura donde el cliente solicita servicios y el servidor los provee.

- Cliente: gestiona la presentación/interfaz de usuario, orquesta solicitudes y presenta resultados.
- Servidor: implementa la lógica de negocio y el acceso a datos (validación, reglas, persistencia) y responde a las peticiones. Esta separación permite desacoplar presentación de lógica y datos.

Pregunta 20

SOA: Estilo arquitectónico que organiza funcionalidades como servicios independientes que representan procesos de negocio y se comunican mediante interfaces y estándares abiertos. Características principales:

- Servicios autónomos y de propósito claro, con contratos bien definidos.
- Acoplamiento débil e interoperabilidad entre plataformas (HTTP/SOAP/REST, XML/JSON).
- Reutilización y composición/orquestación de servicios para procesos más complejos.
- Descubrimiento/registro de servicios y versionado.
- Gobernanza de servicios (políticas, seguridad, SLAs) alineada con metas de negocio.

Pregunta 21

Arquitectura Cliente-Servidor: separa responsabilidades entre un cliente que solicita y presenta información, y un servidor que procesa la lógica y accede a los datos.

- Cliente: inicia peticiones, gestiona la interfaz/presentación y validaciones livianas; no implementa la lógica de negocio central ni accede directamente al almacenamiento persistente.
- Servidor: recibe múltiples solicitudes, ejecuta la lógica de negocio, aplica políticas de seguridad/autenticación/autorización, orquesta procesos y accede a datos (BD, archivos, servicios externos), devolviendo respuestas a los clientes.

Pregunta 22

Arquitectura por capas:

- Ventaja: modularidad y mantenibilidad al separar presentación, negocio y datos con responsabilidades jerárquicas claras.
- Desventaja: puede introducir sobrecarga en llamadas entre capas y limitar la escalabilidad/el despliegue independiente (suele desplegarse como un bloque). Arquitectura de microservicios:
- Ventaja: escalabilidad y despliegue independientes por servicio; mejor aislamiento de fallos y alineación por dominios.
- Desventaja: mayor complejidad operativa y de comunicación (observabilidad, versionado, resiliencia, datos distribuidos) con impacto en latencia entre servicios.

Pregunta 23

HTTP/REST:

- Eficiencia/recursos: típicamente JSON sobre HTTP; payloads más grandes y serialización más costosa; muy interoperable y caché amigable.
- Útil para APIs públicas, compatibilidad con navegadores y ecosistemas heterogéneos. gRPC:
- Eficiencia/recursos: HTTP/2 con multiplexación y streaming; Protobuf binario reduce tamaño y CPU, menor latencia y mejor throughput.
- Útil para comunicación servicio-a-servicio interna, alto QPS/baja latencia, streaming en tiempo real (telemetría, chat, IoT) y contratos tipados estrictos. Recomendar gRPC por encima de HTTP/REST cuando se priorizan rendimiento, baja latencia, streaming bidireccional y no se requiere compatibilidad directa con navegadores.

Pregunta 24

SOA: estilo que organiza la funcionalidad en servicios independientes con interfaces/contratos bien definidos (p. ej., XML/JSON sobre HTTP; SOAP/REST), que representan procesos de negocio y se integran mediante estándares abiertos.

- Ventajas: reutilización y separación de responsabilidades, interoperabilidad entre plataformas/tecnologías, flexibilidad para actualizar/escalar servicios de forma individual, y mejor alineación con procesos de negocio a través de contratos y gobierno de servicios.

Pregunta 25

Interoperabilidad en SOA: capacidad de que servicios desarrollados en lenguajes y plataformas diferentes se comuniquen usando estándares abiertos (HTTP, SOAP/REST, XML/JSON, WSDL/OpenAPI).

- Por qué es fundamental: facilita la integración de sistemas heterogéneos, reduce acoplamiento y costos de integración, permite reutilizar sistemas existentes y cambiar/actualizar componentes sin romper consumidores, habilitando ecosistemas escalables y evolutivos.
-

Pregunta 26

Los microservicios son un estilo arquitectónico que divide la aplicación en servicios pequeños, autónomos y enfocados a capacidades específicas del dominio, cada uno con su propio ciclo de vida y, a menudo, su propio almacenamiento de datos. Se comunican mediante interfaces bien definidas (APIs ligeras o eventos). Ventaja principal frente al monolito: escalabilidad y despliegue independiente por servicio, lo que mejora el mantenimiento y permite adoptar tecnologías distintas por equipo/servicio. A diferencia del monolito (que suele escalar y desplegarse como un todo), aquí solo escalas o actualizas el componente que lo necesita.

Pregunta 27

El middleware es la capa intermedia que facilita la comunicación, integración y gestión de servicios en sistemas distribuidos. En microservicios su rol incluye: interoperabilidad entre lenguajes/plataformas; comunicación y enrutamiento (p. ej., API Gateway, service discovery);

gestión de mensajes (colas/topics), seguridad (authn/z, mTLS), observabilidad y trazabilidad (logs/tracing), y soporte a transacciones/resiliencia (reintentos, circuit breakers). Con ello abstrae la complejidad de red y coordina aspectos transversales.

Pregunta 28
Los sockets son los puntos finales de comunicación entre cliente y servidor; permiten establecer conexiones y enviar/recibir datos de forma bidireccional. Para mensajería en tiempo real (chat), usaría sockets sobre TCP, y en aplicaciones web específicamente WebSockets, porque mantienen una conexión persistente, bidireccional y confiable (entrega ordenada y sin pérdidas), reduciendo latencia de ida y vuelta. UDP se consideraría cuando se prioriza latencia mínima y se tolera pérdida (p. ej., voz o gaming), no típico para chat de texto.

Pregunta 29
Correcta: b) Una aplicación cuyas partes lógicas y de procesamiento se reparten en múltiples nodos o máquinas, comunicándose a través de una red. Esto captura la esencia de una aplicación distribuida según la retroalimentación.

Pregunta 30
Afirmación falsa: c) "No se necesita configurar carpetas de aplicación para que IIS funcione". En IIS es necesario definir el sitio/aplicación, su carpeta física, permisos adecuados y probar en localhost antes de publicar; además de activar las características de IIS.

Pregunta 31
Correcta: c) Los microservicios se pueden desplegar, escalar y actualizar de forma independiente. Esto refleja su independencia y acoplamiento débil; las otras opciones implican uniformidad tecnológica o centralización, contrarias al enfoque de microservicios.

Pregunta 32 Correcta: c) Se usa comunicación síncrona (p. ej., gRPC) para consultas que requieren respuesta inmediata como disponibilidad o asignación de choferes. La asíncrona (p. ej., Kafka) se reserva para eventos y procesamiento en segundo plano como registrar consumo de combustible.

Pregunta 33
Correcta: a) Según CAP, no se puede garantizar simultáneamente consistencia, disponibilidad y tolerancia a particiones. Ante particiones, típicamente se elige entre CP (consistencia) o AP (disponibilidad).

Pregunta 34
Correcta: a) En Peer-to-Peer no hay servidor central; cada nodo actúa como cliente y servidor. Esto contrasta con Cliente-Servidor, donde existe un servidor dedicado que atiende a múltiples clientes.

Pregunta 35 Correcta: a) WebSockets permiten comunicación bidireccional y en tiempo real con menor latencia gracias a conexiones persistentes full-duplex. La opción b es falsa porque sí se mantiene una conexión; la c es incorrecta ya que WebSockets no están limitados a XML.

Pregunta 36
Correcta: a) Kafka. Plataforma distribuida de publicación–suscripción basada en topics y particiones, con brokers y consumer groups para alta escalabilidad y durabilidad.

Pregunta 37
Correcta: d) Sharding = particionar lógicamente los datos en múltiples nodos para repartir carga y lograr escalabilidad horizontal (no confundir con réplica o backups).

Pregunta 38

Correcta: c) Cliente, Servidor y Peer (par). En distribuidos un proceso puede solicitar servicios, proveerlos o actuar como par (ambos roles) según el contexto.

Pregunta 39
Correcta: c) Los heartbeats verifican periódicamente la vivacidad de nodos para detectar fallos/desconexiones; se usan en detección de fallos y elección de líder.

Pregunta 40
Correcta: a) RPC permite invocar métodos remotos como si fueran locales (modelo request–response), ocultando red/ubicación mediante stubs/proxies y marshalling.

Pregunta 41
SOAP es un protocolo con mensajes XML estandarizados (envelope, WSDL) y reglas estrictas; REST es un estilo arquitectónico que aprovecha HTTP y sus verbos para operaciones ligeras, usando formatos como JSON o XML y favoreciendo simplicidad y cacheabilidad.

Pregunta 42
Componentes básicos en SOA:

- Servicio: unidad funcional que implementa una capacidad de negocio.
- Contrato del servicio: qué expone el servicio (operaciones, pre/postcondiciones, políticas).
- Interfaz del servicio: definición técnica de los endpoints y operaciones (WSDL/OpenAPI).
- Registro de servicios: catálogo para publicar y descubrir servicios.

Pregunta 43
Ventaja principal: mayor rendimiento al permitir ejecución concurrente por hilos, aprovechando mejor los núcleos del procesador y atendiendo múltiples clientes en paralelo (con la debida sincronización para recursos compartidos).

Pregunta 44
Una aplicación distribuida es un sistema cuyo software se ejecuta en múltiples equipos conectados por una red y coopera para cumplir tareas comunes. Componentes/características fundamentales:

- Componentes distribuidos bajo modelo cliente-servidor.
- Comunicación en red mediante protocolos/middleware (p. ej., HTTP, gRPC, colas).

Pregunta 45
Una aplicación distribuida ejecuta sus componentes en varias computadoras conectadas en red que comparten recursos y colaboran. Componente fundamental:

- Cliente-servidor (y su comunicación a través de la red).

Pregunta 46
Una aplicación distribuida es aquella cuyos componentes se ejecutan en equipos/plataformas distintos conectados por una red y colaboran para un objetivo común. Componentes fundamentales: lado cliente, lado servidor y protocolos/middleware de comunicación para intercambiar mensajes.

Pregunta 47
Los microservicios son servicios pequeños e independientes que implementan capacidades específicas y se comunican entre sí mediante APIs bien definidas.

Pregunta 48

El middleware actúa como capa intermedia que desacopla y facilita la comunicación e integración entre servicios: gestiona mensajería/enrutamiento, seguridad (autenticación/autorización), y transacciones/coordination, común en entornos de microservicios.

Pregunta 49
El Algoritmo de Lamport asigna marcas temporales lógicas a los eventos para ordenarlos y establecer relaciones causales entre procesos en un sistema distribuido.

Pregunta 50
Escalabilidad independiente: cada servicio puede desplegarse y escalarse de forma autónoma según su demanda.

Pregunta 51
Comunicación síncrona: el emisor bloquea y espera la respuesta en la misma interacción. Comunicación asíncrona: el emisor no bloquea; envía mensajes a través de colas/eventos y el receptor los procesa después.

Pregunta 52
- Instalar Samba: sudo apt update && sudo apt install samba
- Crear carpeta y permisos: sudo mkdir -p /srv/samba/compartida; ajustar propietario/permisos según el caso
- Editar /etc/samba/smb.conf y agregar la sección del recurso compartido con su path y permisos (p. ej., [Compartida] path=/srv/samba/compartida browsable=yes writeable=yes guest ok=yes)
- Probar configuración: testparm
- Reiniciar servicio: sudo systemctl restart smbd
- (Opcional) Abrir firewall: sudo ufw allow samba
- Acceder desde Windows: \IP_DEL_UBUNTU\Compartida

Pregunta 53
Dos tipos de servicios del middleware:
- Servicios de desarrollo (p. ej., comunicación/RPC, naming, mensajería, transacciones).
- Servicios de administración (p. ej., seguridad, monitoreo, logging, configuración, auditoría).

Pregunta 54
RPC es un mecanismo genérico para llamadas remotas entre procesos/plataformas. RMI es específico de Java: invoca métodos de objetos remotos dentro de la JVM, con serialización de objetos y contratos Java.

Pregunta 55
Compartir recursos: aprovechar eficientemente hardware, datos y servicios entre múltiples nodos de la red.

Pregunta56
Ventajas: escalabilidad; tolerancia a fallos.
Desafío: complejidad de la comunicación entre nodos (sincronización y coordinación).

Pregunta 57
Es el mecanismo para designar un nodo coordinador (líder) en el clúster; es importante porque asegura coherencia y evita conflictos al centralizar la coordinación de tareas críticas.

Pregunta 58

Ventajas: escalabilidad horizontal; tolerancia a fallos/aprovechamiento de recursos distribuidos. Desventajas: complejidad en la comunicación y sincronización; mayores retos de seguridad/gestión de fallos.

Pregunta 59
Cliente; Servidor; Protocolo de comunicación (p. ej., HTTP/gRPC) entre ambos.

Pregunta 60
RPC = llamada a procedimiento remoto para invocar funciones en otra máquina como si fueran locales; ejemplo: gRPC.

Pregunta 61
SOA es un modelo de diseño que organiza la funcionalidad en servicios independientes que se comunican mediante interfaces/protocolos bien definidos, permitiendo reutilización y desacoplamiento.

Pregunta 62
Ventaja: mantenimiento y actualización más eficientes gracias a servicios aislados. Desventaja: aumento de interdependencias y complejidad si no se gobierna/diseña correctamente.

Pregunta 63
Ejemplo: un sistema ERP que integra finanzas, RR. HH. y ventas mediante servicios web (SOAP/REST), donde cada módulo expone servicios independientes consumidos por los demás.

Pregunta 64
Un proceso tiene su propio espacio de memoria independiente; un hilo comparte la memoria del proceso al que pertenece.

Pregunta 65
- Desacoplamiento entre productores y consumidores.
- Alta escalabilidad y tolerancia a fallos (buffering, reintentos y consumo paralelo).

Pregunta 66
Ejemplo banca: el cliente (app móvil/web) solicita consultas o transacciones; el servidor procesa la lógica, valida, gestiona la base de datos y retorna la respuesta. Ejemplo videojuegos: el cliente envía acciones y muestra el juego; el servidor mantiene la lógica/estado de la partida, sincroniza a los jugadores y responde con actualizaciones.

Pregunta 67
Alojar la aplicación en un servidor web y servir solicitudes HTTP a los clientes (publicación del sitio, aplicación en IIS, bindings y permisos), para que sea accesible vía HTTP.

Pregunta 68
Falso.
En CAP, disponibilidad implica responder aun ante fallos/particiones, aunque pueda ser con datos no totalmente actualizados.

Pregunta 69
Falso.
La replicación puede ser asíncrona; se permiten retrasos controlados entre nodos.

Pregunta 70
Falso.
HTTP es un protocolo; REST es un estilo arquitectónico que lo aprovecha, no son equivalentes ni intercambiables.

Conclusiones

En el diseño de sistemas distribuidos se evidencia que la elección de estilo de comunicación (síncrona vs. asíncrona), el modelo arquitectónico (monolito, por capas, microservicios o SOA) y los componentes de infraestructura (middleware, colas, protocolos como REST/gRPC, y patrones como RPC) determinan el acoplamiento, la escalabilidad y la tolerancia a fallos del sistema. La asincronía y el uso de mensajería (Kafka/RabbitMQ) fomentan el desacoplamiento temporal y la elasticidad, mientras que la sincronía simplifica flujos de consulta con respuesta inmediata a costa de mayor dependencia entre servicios. Conceptos fundacionales como CAP, consistencia (fuerte/eventual), relojes lógicos (Lamport) y consenso/líder (Raft) explican los límites y compensaciones prácticas en disponibilidad y coherencia. SOA y microservicios promueven modularidad, reutilización y despliegue independiente, pero requieren gobierno y observabilidad robustos. En conjunto, la arquitectura, los patrones de integración y los mecanismos de coordinación deben alinearse con los requisitos de negocio y no funcionales (latencia, throughput, resiliencia y evolución).

Recomendaciones

Usar comunicación síncrona (gRPC/REST) en operaciones de consulta/actualización que requieran respuesta inmediata y comunicación asíncrona (Kafka/colas) para eventos y procesos desacoplados; diseñar idempotencia, reintentos y control de orden.

Adoptar microservicios cuando existan límites claros de dominio y necesidades de escalado/despliegue independiente; acompañar con gobierno (contratos, versionado), trazabilidad distribuida (correlation IDs, OpenTelemetry), seguridad extremo a extremo (mTLS, OAuth2), y service discovery/gateways.

Seleccionar consistencia eventual donde la precisión inmediata no sea crítica (feeds, contadores) y consistencia fuerte donde la integridad transaccional sea prioritaria; documentar decisiones con base en CAP.

Implementar mecanismos de detección de fallos (heartbeats), elección de líder y consenso (Raft) en clústeres que requieran orden/replicación; probar con inyección de fallos y chaos testing.

Estandarizar contratos (OpenAPI/Protobuf), prácticas de CI/CD y observabilidad (logs, métricas, traces) para reducir el costo operativo y facilitar la evolución segura del sistema.

Referencias bibliográficas

- [1] A. S. Tanenbaum and M. van Steen, Distributed Systems: Principles and Paradigms, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2007.
- [2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, Distributed Systems: Concepts and Design, 5th ed. Boston, MA, USA: Addison-Wesley, 2011.
- [3] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA, USA: O'Reilly Media, 2017.
- [4] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Univ. of California, Irvine, 2000.
- [5] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm (Raft)," in Proc. 2014 USENIX Annual Technical Conference (USENIX ATC), 2014, pp. 305–319.
- [6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, pp. 558–565, 1978.
- [7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, vol. 33, no. 2, pp. 51–59, 2002.
- [8] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ, USA: Prentice Hall, 2005.
- [9] S. Newman, Building Microservices, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

- [10] Apache Kafka, “Documentation.” [Online]. Available: https://kafka.apache.org/documentation/ Accessed: Oct. 3, 2025.
- [11] gRPC, “What is gRPC?” [Online]. Available: https://grpc.io/docs/what-is-grpc/ Accessed: Oct. 3, 2025.
- [12] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in Proc. 19th ACM Symposium on Operating Systems Principles (SOSP), 2003, pp. 29–43.