Desarrollo de Sistemas Distribuidos

Resumen Completo — Temas 4 y 5

Universidad de Granada · Grado Ingeniería Informática · Curso 2024–25

Basado íntegramente en las diapositivas oficiales (slides 152–255)

Realizado por **@jcartov** · Más recursos en **repasaYA**

TEMA 4 — Sistemas Distribuidos: Modelos y Paradigmas

Modelos de Sistemas Distribuidos (SD)

Existen tres tipos de modelos complementarios para describir un SD:

- **Modelo Físico**: describe el hardware, las redes y los dispositivos. Captura la evolución del hardware distribuido a lo largo del tiempo.
- **Modelo Arquitectónico**: describe la estructura de las tareas de computación y comunicación. Define cómo se organizan, interconectan y despliegan los componentes del sistema (entidades, paradigmas de comunicación, roles, estrategias de despliegue).
- **Modelo Fundamental**: trata los aspectos de interacción, fallos y seguridad de forma independiente a la arquitectura concreta. Permite razonar sobre propiedades universales de los SD.

Modelo Físico: Evolución de los SD

1. **Primeros SD (años 80–90)**: redes de área local (LAN) que conectan estaciones de trabajo y servidores. Servicios básicos compartidos: impresión, archivos, correo. Escalabilidad limitada, ámbito local, sin conexión a Internet.
2. **Sistemas basados en Internet (años 90–2000)**: redes heterogéneas de gran escala con gran diversidad de HW/SW. Surge la necesidad de middleware de interoperabilidad. Aparecen los primeros problemas de apertura, escala y heterogeneidad.
3. **Sistemas Contemporáneos**: aparecen nuevas formas de distribución a escala global:
   - **Cluster**: conjunto de nodos homogéneos interconectados en rack por red de alta velocidad. Objetivo: HPC y servidores web escalables horizontalmente.
   - **Grid computing**: recursos distribuidos geográficamente entre organizaciones (virtuales), heterogéneos. Foco en compartición de recursos computacionales científicos.
   - **Cloud computing**: recursos virtualizados bajo demanda, modelo de pago por uso (IaaS / PaaS / SaaS). El proveedor gestiona la infraestructura física.
   - **IoT** (*Internet of Things*): red de dispositivos empotrados, sensores y actuadores físicos con capacidad de comunicación. Cómputo limitado, gran volumen de datos.
   - **System of Systems (SoS)**: SD compuesto de otros SD preexistentes, autónomos y heterogéneos, integrados para una función de mayor nivel.

Modelo Arquitectónico: Elementos

Entidades que se comunican

Las entidades son los actores del sistema distribuido:

- **Objetos**: paradigma orientado a objetos, con estado encapsulado e invocación de métodos.
- **Componentes**: módulos con interfaces bien definidas, composición explícita en tiempo de diseño o despliegue.
- **Servicios Web / Microservicios**: expuestos a través de HTTP/REST o SOAP, descubribles y componibles.

Paradigmas de Comunicación

Los paradigmas definen *cómo* se comunican las entidades:

1. **Comunicación entre procesos** (IPC): nivel bajo; sockets TCP/UDP, pipes, memoria compartida. Plataforma-específico.
2. **Invocación remota**: RPC (*Remote Procedure Call*), RMI (*Remote Method Invocation*). Modelo petición-respuesta con semántica de llamada local.
3. **Comunicación indirecta**: colas de mensajes, pub/sub. Los comunicantes no se conocen directamente: desacoplamiento en espacio y tiempo.

Roles y Responsabilidades

- **Cliente/Servidor (C/S)**: roles *asimétricos*. El cliente inicia solicitudes; el servidor espera en un puerto, procesa y responde. El servidor no inicia comunicación.
- **Peer-to-Peer (P2P)**: roles *simétricos*. Cada nodo actúa simultáneamente como cliente y servidor. Los recursos están distribuidos entre todos los nodos participantes.

Estrategias de Despliegue

- **On-site**: toda la infraestructura (servidores, red, SO, middleware, aplicación) es gestionada y alojada por la propia organización.
- **IaaS** (*Infrastructure as a Service*): el proveedor cloud ofrece máquinas virtuales, red y almacenamiento. La organización gestiona SO, middleware y aplicación.
- **PaaS** (*Platform as a Service*): el proveedor ofrece además el entorno de ejecución (runtime, BD, middleware). La organización solo gestiona la aplicación y los datos.
- **SaaS** (*Software as a Service*): el proveedor gestiona todo, incluyendo la aplicación. El usuario solo la usa a través del navegador o API.

Estilos Arquitectónicos

Estilo	Descripción detallada
Capas (*Layers*)	Organización lógica vertical: cada capa ofrece servicios a la inmediatamente superior y usa los de la inferior. Impone separación de preocupaciones. Ej.: TCP/IP, OSI.
Etapas (*Tiers*)	Distribución *física* en máquinas separadas. La separación en tiers es una proyección de las capas lógicas sobre nodos físicos distintos (presentación, negocio, datos).
Cliente/Servidor	Un servidor ofrece servicios a múltiples clientes concurrentes. Roles estables y diferenciados.
EDA	*Event-Driven Architecture*: los componentes reaccionan a eventos asíncronos. Fuerte desacoplamiento. Base del paradigma pub/sub.
Proxy	Intermediario entre cliente y servidor: añade seguridad, caché, balanceo de carga o conversión de protocolo. El cliente cree hablar con el servidor.
Broker	Componente central que desacopla completamente a productores y consumidores de servicios. Gestiona registro, descubrimiento y encaminamiento. Ej.: CORBA ORB, RabbitMQ.
Reflexión	El sistema puede *inspeccionarse* y *modificarse* a sí mismo en tiempo de ejecución (meta-nivel). Útil para adaptación dinámica y reconfiguración.

Paradigmas de Comunicación: Definiciones Detalladas

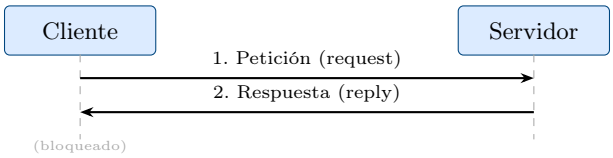
1. Petición / Respuesta (Request-Reply)

Definición: paradigma de comunicación *síncrono* en el que el cliente envía una solicitud y queda **bloqueado** esperando hasta recibir la respuesta del servidor. Constituye la base de RPC y RMI.

Características:

- **Sincronía**: el cliente no puede continuar su ejecución hasta recibir la respuesta.
- **Acoplamiento temporal**: ambas partes deben estar activas al mismo tiempo.
- **Semántica de invocación**: puede ser *at-most-once* (como mucho una vez, sin duplicados) o *at-least-once* (reintento en timeout), dependiendo del middleware.
- **Idempotencia**: las operaciones de solo lectura son idempotentes (pueden reinvocarse sin efectos secundarios); las de escritura pueden no serlo.

Diagrama:



Ejemplos: HTTP GET/POST, llamadas RMI en Java, llamadas XML-RPC.

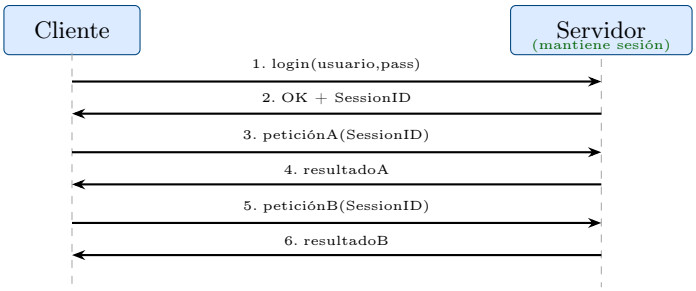
2. Comunicación Conversacional

Definición: extensión del modelo petición-respuesta para **múltiples intercambios secuenciales** dentro del contexto de una **sesión**. El servidor *mantiene estado* (contexto) sobre el cliente entre peticiones sucesivas.

Características:

- **Estado de sesión**: el servidor recuerda quién es el cliente y qué ha hecho hasta ahora (contrario al modelo sin estado de petición/respuesta pura).
- **Múltiples intercambios**: una "conversación" consta de varias peticiones-respuestas enlazadas lógicamente.
- **Establecimiento y cierre**: la sesión tiene inicio explícito (handshake, login) y fin (logout, timeout).
- **Identificador de sesión**: se usa un token o cookie para correlacionar peticiones con la sesión correcta.
- **Mayor acoplamiento**: ambas partes deben mantener el estado activo durante toda la conversación.

Diagrama (estilo diagrama de secuencia):



Ejemplos: protocolo FTP (login → LIST → GET → logout), carrito de compra HTTP con cookies, sesiones de videojuego online, SSH interactivo.

Diferencia clave con R/R básico: en R/R puro el servidor es *stateless* (sin estado entre llamadas); en conversacional el servidor acumula contexto a lo largo de la sesión.

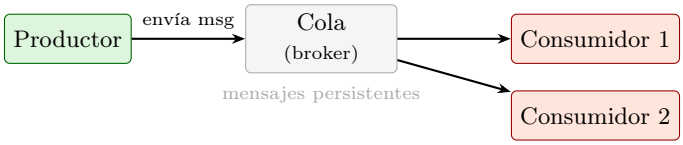
3. Mensajes Encolados (Message Queuing)

Definición: paradigma en el que el productor deposita mensajes en una **cola persistente** (gestionada por un broker de mensajes) y el consumidor los extrae cuando puede. El productor y consumidor **no necesitan estar activos al mismo tiempo**.

Características:

- **Desacoplamiento temporal**: el productor no espera al consumidor. Máxima independencia de ciclos de vida.
- **Desacoplamiento espacial**: el productor y el consumidor no se conocen entre sí directamente; solo conocen la cola.
- **Persistencia**: los mensajes se almacenan hasta ser consumidos (o hasta expirar). Garantiza entrega aunque el consumidor esté caído.
- **Procesamiento asíncrono**: el consumidor procesa a su propio ritmo, sin bloquear al productor.
- **Balanceo natural**: múltiples consumidores pueden procesar mensajes de la misma cola (*competing consumers*).

Diagrama:



Variantes middleware:

- **DCM** (*Data-Centric Middleware*): el mensaje tiene una estructura de datos tipada y fuertemente definida (esquema). El middleware enruta basándose en el contenido del mensaje.
- **DDS** (*Data Distribution Service*, estándar OMG): middleware pub/sub orientado a datos para sistemas de tiempo real con QoS configurable (fiabilidad, deadline, historia, durabilidad, lifespan, ownership, etc.). Muy usado en sistemas embebidos y militares.

Ejemplos: RabbitMQ, Apache Kafka, IBM MQ, Amazon SQS.

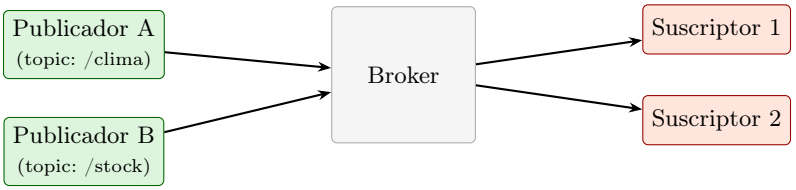
4. Publicación-Suscripción (Pub/Sub) y EDA

Definición: paradigma en el que los **publicadores** (*publishers*) emiten eventos/mensajes a **tópicos o canales**, y los **suscriptores** (*subscribers*) reciben automáticamente los eventos de los tópicos a los que se han suscrito, *sin conocer directamente a los publicadores.*

Características fundamentales:

- **Desacoplamiento en espacio**: el publicador no sabe quién suscribe, ni el suscriptor quién publica.
- **Desacoplamiento en tiempo**: el publicador y suscriptor no necesitan estar activos simultáneamente (depende de la implementación).
- **Tipos de suscripción**:
  - *Basada en tópico*: el suscriptor especifica un canal/tema (ej.: "/sensores/temperatura").
  - *Basada en contenido*: el suscriptor especifica un filtro sobre el contenido del mensaje (ej.: temperatura > 30).
- **Broker**: componente central que recibe publicaciones y las distribuye a suscriptores registrados. Actúa como intermediario.
- **Escalabilidad**: el publicador puede tener miles de suscriptores sin cambiar su código.

Diagrama Pub/Sub general (arquitectura broker):



Pull-based vs Push-based:

	Pull-based (polling)	**Push-based**
Quién inicia	El suscriptor pregunta periódicamente al broker	El broker notifica al suscriptor
Latencia	Alta (depende del intervalo de polling)	Baja (notificación inmediata)
Carga en suscriptor	Alta (polling continuo aunque no haya eventos)	Baja (solo procesamiento al recibir)
Carga en broker	Baja	Necesita mantener conexiones activas

Diagrama Pull vs Push:

Tecnologías Push-based:

- **HTML5 Server-Sent Events (SSE)**: canal *unidireccional* servidor → cliente sobre HTTP estándar. El cliente abre una conexión persistente y el servidor envía eventos de texto cuando ocurren. Soporte de reconexión automática. Limitado a texto/UTF-8.
- **WebSockets**: protocolo de canal *bidireccional full-duplex* sobre TCP. Se inicia con un handshake HTTP (HTTP Upgrade) y luego la conexión permanece abierta. Permite envío de datos binarios o texto en cualquier dirección. Baja latencia; ideal para aplicaciones en tiempo real (chats, juegos, dashboards).
- **Socket.IO**: biblioteca JavaScript que abstrae los WebSockets añadiendo características de alto nivel:
  - **Namespaces**: canales virtuales dentro de la misma conexión WebSocket.
  - **Rooms**: grupos de sockets dentro de un namespace para broadcast selectivo.
  - **Fallback automático** a HTTP long-polling si WebSockets no está disponible.
  - Reconexión automática y heartbeat.

Tabla comparativa: Pub/Sub vs. Petición-Respuesta

Propiedad	Pub/Sub	R/R	Explicación
Efficiency	Parcial	Parcial	Pub/Sub eficiente para muchos receptores; R/R para diálogo directo
Mobility support	✓	×	Pub/Sub tolera cambios de dirección IP; R/R requiere dirección estable
Adaptability	✓	×	Pub/Sub permite añadir suscriptores sin cambiar publicadores
Reliable Delivery	Parcial	✓	R/R tiene ACK directo; Pub/Sub depende del broker
Security	Parcial	✓	R/R tiene canal directo asegurable; en Pub/Sub el broker es punto de riesgo
Timeliness	Parcial	✓	R/R garantiza tiempo de respuesta; Pub/Sub añade latencia de broker

Servicios

Definición: un *servicio* es una unidad de funcionalidad discreta, autónoma y bien delimitada, accesible a través de una **interfaz formal y estandarizada**, independiente de la tecnología de implementación interna. Los servicios son los bloques de construcción de SOA y MSA.

8 Principios de Diseño de Servicios (Thomas Erl)

1. **Abstracción**: la interfaz del servicio oculta completamente los detalles de implementación interna (lenguaje, BD, algoritmo). El consumidor solo conoce el contrato.
2. **Contrato estandarizado** (*standardized service contract*): la interfaz es formal, explícita y publicada (WSDL para SOAP, OpenAPI/Swagger para REST). Define operaciones, tipos de datos, endpoints y restricciones.
3. **Débil acoplamiento** (*loose coupling*): dependencias mínimas entre el servicio y sus consumidores. Los cambios internos no rompen a los consumidores. Se logra mediante abstracciones e interfaces estables.
4. **Reutilización** (*reusability*): los servicios se diseñan para ser usados en múltiples contextos y aplicaciones distintas, no para un caso de uso único.
5. **Autonomía** (*autonomy*): el servicio tiene control total sobre su propia lógica y entorno de ejecución. No depende del estado de otros servicios para funcionar.
6. **Sin estado** (*statelessness*): el servicio no guarda estado de sesión entre invocaciones sucesivas. Cada llamada es autocontenida. Facilita la escalabilidad horizontal.
7. **Descubrimiento** (*discoverability*): el servicio publica metadatos (nombre, descripción, operaciones, endpoint) en un registro que permite localizarlo dinámicamente (UDDI, DNS-SD, Consul, Eureka).
8. **Componibilidad** (*composability*): los servicios pueden combinarse para construir servicios de mayor nivel (*service composition*). El resultado es también un servicio (principio recursivo).

SOA (Service-Oriented Architecture)

- Paradigma de **"software bajo demanda"**: los servicios se adquieren, descubren y usan en el momento necesario, sin integración ad-hoc previa.
- **Tres tipos de encapsulación** de lógica existente como servicio:
  1. **Sistemas legacy** (mainframes, ERPs, COBOL): se expone su funcionalidad como servicio mediante un adaptador/wrapper, sin reescribir la lógica.
  2. **Componentes a medida** (*custom-built*): nuevos servicios construidos desde cero siguiendo los principios SOA.
  3. **Composición de otros servicios**: un servicio que orquesta o agrega la funcionalidad de otros servicios existentes.
- **ESB** (*Enterprise Service Bus*): middleware central de SOA que actúa de intermediario para enrutamiento de mensajes, transformación de formatos, orquestación de servicios y gestión de seguridad. Todos los servicios se comunican a través del ESB.

Modelo Cliente/Servidor

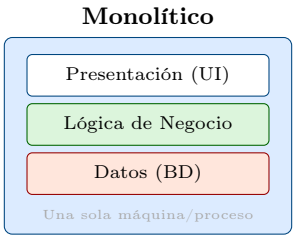
Modelos de N-Etapas (N-Tier)

- **Monolítico**: toda la lógica (presentación + negocio + datos) reside en un único proceso o máquina. Sin distribución. Fácil de desarrollar y depurar, pero no escala ni tolera fallos.
- **2-etapas (Two-tier)**:
  - *Fat client*: el cliente contiene la **lógica de negocio** además de la presentación. El servidor es un simple gestor de datos (BD). El cliente "sabe" cómo procesar los datos. Problema: actualizar la lógica requiere instalar nueva versión en cada cliente.
  - *Fat server* (*thin client*): el servidor concentra lógica de negocio y datos. El cliente es ligero (terminal, navegador). Toda la inteligencia está en el servidor; más fácil de mantener centralmente.

- **3-etapas (Three-tier)**: separación estricta en tres capas físicas:
  1. Capa de **presentación**: UI (navegador, app móvil). Solo renderiza y captura entrada.
  2. Capa de **lógica de negocio**: servidor de aplicaciones. Procesa las reglas y flujos.
  3. Capa de **datos**: SGBD o servidor de archivos. Persiste y consulta datos.

  Es el modelo más habitual en aplicaciones web modernas.
- **n-etapas (N-tier)**: múltiples capas intermedias adicionales (capa de caché, capa de mensajería, múltiples microservicios, CDN, etc.). Característica de arquitecturas empresariales y cloud-native.

Diagramas comparativos de los modelos:



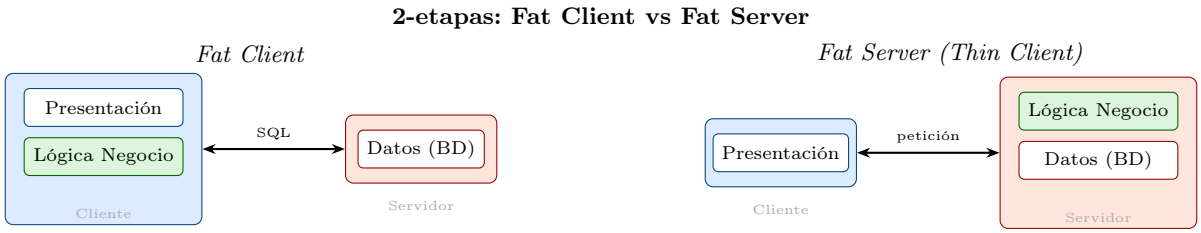
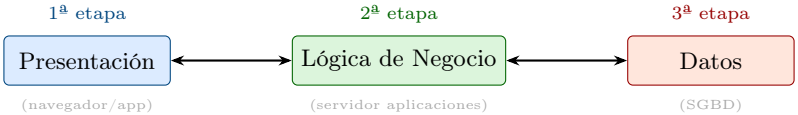


Diagrama 3-etapas:



Servidor de Aplicaciones

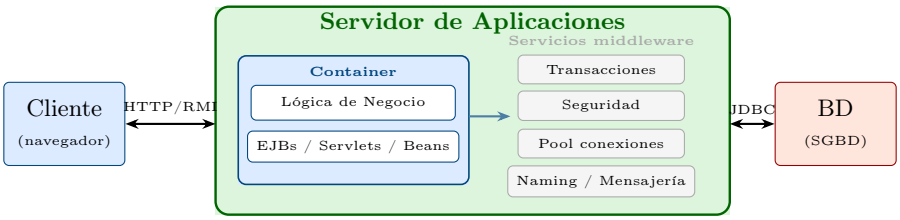
El **servidor de aplicaciones** es un **middleware híbrido**: implementa el **patrón container** combinado con servicios de middleware clásicos.

¿Qué es el patrón container? El *container* (contenedor) es el entorno de ejecución gestionado que rodea a los componentes de la aplicación. El desarrollador solo escribe la **lógica de negocio** (EJBs, Servlets, Spring Beans); el container se encarga *automáticamente* de toda la infraestructura:

- **Ciclo de vida**: instanciar, reutilizar (pooling) y destruir componentes.
- **Transacciones**: iniciar, confirmar (commit) o deshacer (rollback) transacciones distribuidas sin que el desarrollador escriba ese código.
- **Seguridad**: control de acceso declarativo (quién puede llamar a qué) sin código en la lógica de negocio.
- **Concurrencia**: gestión de hilos y peticiones concurrentes de forma transparente.
- **Pool de conexiones**: mantiene un conjunto de conexiones abiertas a la BD y las reutiliza entre peticiones (evita el coste de abrir/cerrar por cada llamada).

Además del container, ofrece **servicios de middleware**: naming/directorio (JNDI), mensajería (JMS), caché, clustering y balanceo de carga.

Diagrama: estructura interna del servidor de aplicaciones



Ejemplos: **JEE**/Jakarta EE (especificación estándar), **GlassFish** (referencia JEE), **WebSphere** (IBM), **JBoss/WildFly** (Red Hat), **Spring Boot** (ligero, embebe Tomcat).

Clasificación de Servicios TIC

1. **Servicios de transporte**: transmisión de datos entre nodos (TCP/IP, HTTP, SMTP, FTP). Garantizan que los datos llegan de A a B.
2. **Servicios de manejo de recursos**: gestión del almacenamiento y acceso a datos (sistemas de ficheros distribuidos, SGBD, LDAP, DNS). Permiten compartir recursos.
3. **Servicios de administración**: gestión de usuarios, autenticación, autorización, auditoría, monitorización de la PCD. Aseguran el correcto funcionamiento del sistema.

Servicios completos vs. IPC:

- **Servicios completos**: ofrecen funcionalidad de alto nivel lista para usar (ej.: un SGBD completo, un servidor de directorio LDAP, un sistema de colas de mensajes).
- **IPC** (*Inter-Process Communication*): mecanismos básicos de bajo nivel para comunicación entre procesos: sockets, pipes, memoria compartida, semáforos. El desarrollador construye la lógica de aplicación sobre ellos.

ArchiMate: Modelado de Arquitectura Empresarial

ArchiMate es un lenguaje de modelado de arquitectura empresarial (estándar The Open Group). Permite representar de forma unificada los componentes de negocio, aplicación e infraestructura y sus relaciones.

Define **tres capas**, cada una con dos **subcapas**:

Capa	Subcapa	Contenido
Negocio	*Servicio* (fachada)	Servicios de negocio visibles al exterior (lo que ofrece la organización)
	Realización (impl.)	Procesos de negocio, actores, roles, funciones que realizan el servicio
Aplicación	*Servicio* (fachada)	Servicios de aplicación expuestos a la capa de negocio
	Realización (impl.)	Componentes software, interfaces, flujos de datos que realizan el servicio
Infraestructura	*Servicio* (fachada)	Servicios de infraestructura (capacidad de cómputo, almacenamiento, red)
	Realización (impl.)	Hardware, dispositivos de red, plataformas técnicas, sistemas operativos

Las capas se relacionan verticalmente: la capa superior *usa* los servicios de la inferior.

Modelo Peer-to-Peer (P2P)

Definición: en P2P **cada nodo actúa simultáneamente como cliente y servidor**, aportando y consumiendo recursos (almacenamiento, CPU, ancho de banda, datos) sin necesidad de servidores centrales dedicados. La funcionalidad del sistema emerge de la cooperación de todos los nodos.

Características:

- **Servicio uniforme**: todos los nodos ofrecen el mismo tipo de servicio al sistema (ej.: almacenamiento de bloques de datos en BitTorrent).
- **Algoritmos distribuidos**: la localización de recursos y el balanceo de carga se resuelven mediante algoritmos distribuidos. Destaca la **DHT** (*Distributed Hash Table*): estructura de datos distribuida que asigna claves a nodos de forma determinista, permitiendo búsqueda en $O(\log n)$ saltos. Implementaciones: Chord, Kademlia (base de BitTorrent y Ethereum).
- **No centralizado**: no existe punto único de fallo (SPOF) ni cuello de botella central. El sistema es robusto ante fallos parciales.
- **Escalabilidad horizontal**: añadir nodos incrementa la capacidad del sistema, no solo la demanda.
- **Anonimato limitado**: se puede lograr cierto anonimato (los nodos intermedios no conocen origen/destino final), pero no es absoluto.

C/S vs. P2P

Aspecto	Cliente/Servidor	Peer-to-Peer
Roles	Asimétricos (cliente inicia, servidor responde)	Simétricos (cada nodo es ambos)
Escalabilidad	Vertical (más recursos en el servidor)	Horizontal (más nodos)
Punto único de fallo	Sí (el servidor)	No (descentralizado)
Administración	Centralizada y sencilla	Distribuida y compleja
Localización de recursos	Servidor de nombres/directorio central	Algoritmos distribuidos (DHT)
Relación entre modelos	C/S puede implementarse *sobre* P2P	P2P usa internamente C/S para algunas funciones

El **procesamiento cooperativo** combina ambos modelos: usa C/S para la interfaz de usuario y para funciones de control, y P2P para la distribución interna de datos y cómputo. Ej.: aplicaciones de videoconferencia usan un servidor de señalización (C/S) y comunicación directa entre pares para el media (P2P).

Rendimiento en SD

Principio fundamental: la **principal consideración de rendimiento** en un SD es **minimizar el número de mensajes enviados por la red**. La latencia de red (RTT: *Round-Trip Time*) domina sobre el tiempo de procesamiento local: una operación local (μs) vs. una llamada remota (ms). Agrupar datos en menos mensajes más grandes (*batching*) reduce el overhead.

Web Services

Los **Web Services** proporcionan una **interfaz de servicio descrita en IDL** (WSDL) para la interacción con los clientes. Permiten interoperabilidad entre plataformas y tecnologías heterogéneas sobre protocolos estándar de Internet (HTTP).

Pila tecnológica (de abajo a arriba):

1. **XML** (*eXtensible Markup Language*): formato universal de codificación y *marshalling/serialización* de datos. Autodescriptivo, legible por humanos y máquinas. Base de toda la pila.
2. **SOAP** (*Simple Object Access Protocol*): protocolo de mensajería basado en XML para invocar operaciones remotas sobre HTTP (u otros transportes). Un mensaje SOAP tiene: Envelope $\supset$ Header (metadatos: seguridad, trazabilidad) + Body (payload con la operación y parámetros). Orientado a RPC o Document-style.
3. **REST** (*REpresentational State Transfer*): estilo arquitectónico (no protocolo) que usa los verbos HTTP nativos sobre recursos identificados por URL. GET (leer), POST (crear), PUT (actualizar completo), PATCH (actualizar parcial), DELETE (borrar). Respuestas en JSON o XML. Alternativa *ligera* a SOAP; dominante en APIs modernas.
4. **WSDL** (*Web Services Description Language*): IDL (*Interface Definition Language*) para Web Services SOAP. Documento XML que define: tipos de datos (Types), mensajes (Messages), operaciones (PortType/Interface), enlace de protocolo (Binding) y dirección del servicio (Service/Port). Equivalente a un contrato formal.
5. **UDDI** (*Universal Description, Discovery and Integration*): directorio/registro centralizado de Web Services. Permite a los proveedores publicar y a los consumidores descubrir servicios disponibles. Actúa como "páginas amarillas" de servicios. En la práctica fue poco adoptado; hoy se usan API Gateways y registros de servicios (Consul, Eureka).
6. **Seguridad**: **WS-Security** (autenticación con tokens, cifrado y firma digital de mensajes SOAP a nivel de mensaje, no solo de transporte). También: WS-Trust, WS-Policy.
7. **Coordinación**: **WS-Coordination**, **WS-AtomicTransaction** (protocolo de commit en 2 fases para transacciones distribuidas entre múltiples Web Services).
8. **Coreografía y orquestación**: **WS-BPEL** (*Business Process Execution Language*) define flujos de trabajo complejos entre múltiples servicios (orquestación ejecutable). **WS-CDL** define la coreografía (colaboración entre iguales).

Diagrama de la pila (de abajo a arriba):

Seguridad / Coordinación / Coreografía (WS-Security, WS-BPEL)
Descubrimiento: **UDDI** (registro de servicios)
Descripción: **WSDL** (IDL + endpoint URL)
Comunicación: **SOAP** / **REST** (sobre HTTP)
Codificación / Serialización: **XML**

MSA: Arquitectura de Microservicios

Definición y Motivación

Microservicios: estilo arquitectónico que estructura una aplicación como un conjunto de **servicios pequeños, independientes y de propósito único**. Cada servicio puede desplegarse, escalar y actualizarse *de forma autónoma*.

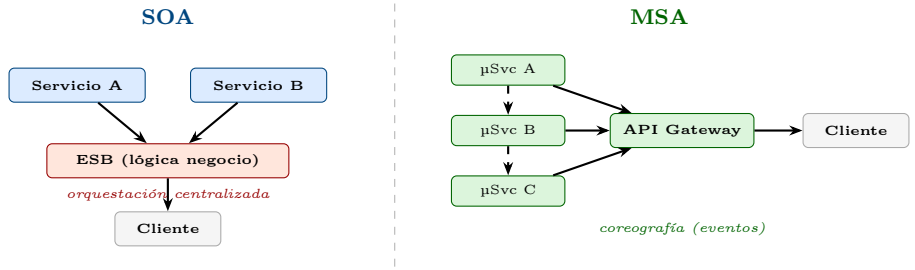
- **Motivación**: **rapidez y agilidad** en el desarrollo, compatible con metodologías Scrum y DevOps (CI/CD).
- **División vertical**: cada microservicio abarca toda su pila propia (presentación, lógica, datos). Contrasta con SOA, que divide horizontalmente por capas técnicas.
- **Despliegue independiente**: un servicio puede actualizarse sin redesplegar el resto de la aplicación.

Despliegue: On-site / IaaS / PaaS / SaaS

Modelo	Gestiona el equipo	Gestiona el proveedor
On-site	Todo (app, datos, runtime, SO, HW, red)	Nada
IaaS	App, datos, runtime, SO	Virtualización, servidores, red, almacenamiento
PaaS	App, datos	Runtime, SO, HW, middleware, red
SaaS	Solo uso de la aplicación	Todo

SOA vs. MSA: 5 Dimensiones

Dimensión	SOA	MSA
Compartición componentes	Alta: reutilización máxima entre sistemas	Mínima: la duplicación es aceptable
Granularidad	Servicios gruesos, versátiles y multipropósito	Servicios finos, de propósito único
Coordinación	Orquestación centralizada (ESB controla el flujo)	Coreografía descentralizada (cada servicio reacciona a eventos)
Middleware	ESB pesado con lógica de negocio	API Gateway ligero solo para enrutamiento/auth
Interoperabilidad	Entre sistemas empresariales heterogéneos	Dentro del ecosistema del equipo de desarrollo

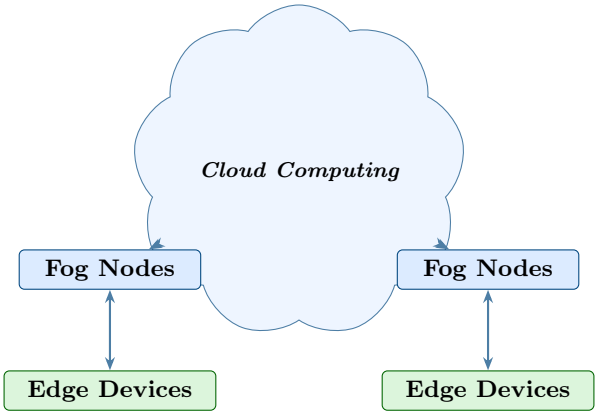


Computing Continuum: Cloud, Fog, Mist, Edge, IoT

Computing Continuum: jerarquía de capas de computación que van desde la nube remota hasta los dispositivos físicos, procesando los datos lo más cerca posible de su origen para reducir latencia y tráfico.

$$\text{Cloud} \longrightarrow \text{Fog} \longrightarrow \text{Mist} \longrightarrow \text{Edge} \longrightarrow \text{IoT}$$

Capa	Descripción
Cloud	Centros de datos remotos. Gran capacidad de cómputo y almacenamiento.
Fog Nodes	Nodos intermedios (routers, gateways) que pre-procesan datos antes de subirlos a la nube. Reducen latencia y tráfico.
Mist	Capa entre Fog y Edge: nodos aún más cercanos al dispositivo (p.ej. microcontroladores con algo de cómputo). Menor latencia que Fog con más capacidad que el propio dispositivo IoT.
Edge Devices	Computación en el borde de la red, próxima a los dispositivos físicos. Latencia mínima.
IoT	Sensores, actuadores y dispositivos empotrados que generan datos. Capacidad de cómputo muy limitada.



Modelo Fundamental

Los modelos fundamentales capturan propiedades universales de todos los SD, independientes de la arquitectura concreta. Hay tres aspectos: interacción, fallos y seguridad.

Modelo de Interacción

Rendimiento de canales de comunicación:

- **Latencia**: retardo entre el inicio de la transmisión de un mensaje desde un proceso y el inicio de su recepción por otro. Incluye: tiempo de transmisión en la red, retardo de acceso a la red (aumenta con la carga) y tiempo de los servicios del SO en emisor y receptor.
- **Ancho de banda**: cantidad total de información que puede transmitirse en un tiempo dado. Cuando muchos canales usan la misma red, comparten el ancho de banda disponible.
- **Jitter**: variación en el tiempo de entrega de mensajes sucesivos. Relevante en datos multimedia: si muestras de audio se reproducen con intervalos distintos, el sonido se distorsiona.

Relojes y temporización: cada ordenador tiene su propio reloj. Los relojes se **desvían** del tiempo perfecto a distintas **tasas de deriva** (*drift rate*): incluso leyendo al mismo tiempo, dos procesos pueden obtener valores diferentes. **No existe tiempo global en un SD**.

Dos modelos extremos:

- **SD Síncrono**: existen cotas conocidas para (a) el tiempo de ejecución de cada paso de proceso, (b) el retardo de transmisión de mensajes y (c) la tasa de deriva de relojes. Permite usar *timeouts* para detectar fallos de procesos con certeza.
- **SD Asíncrono**: no existen límites para ninguno de los anteriores (un paso puede tardar desde un picosegundo hasta un siglo; un mensaje puede llegar en tiempo negligible o en años). Modela exactamente Internet.

Ordenación de eventos — Relojes lógicos de Lamport [1978]: los relojes físicos no se pueden sincronizar perfectamente. Lamport propuso un modelo de tiempo lógico para inferir el **orden causal** entre eventos de procesos distintos sin recurrir a relojes físicos. Sabemos que un mensaje se recibe *después* de haberse enviado, lo que permite establecer una ordenación lógica entre pares de eventos.

Modelo de Fallos

Los fallos se clasifican según su severidad (de menor a mayor):

- **Fallos por omisión**: proceso o canal no realiza acciones que debería.
  - **Omisión de proceso (crash)**: el proceso se detiene y no ejecuta ningún paso más de su programa. Si otros procesos pueden *detectarlo con certeza* ⇒ **fallo-parada** (*fail-stop*), posible en SD síncrono con timeouts y mensajes garantizados.
  - **Omisión de comunicación**: el canal no transporta el mensaje ("pérdida de mensaje"). Causas: buffer lleno en receptor o pasarela intermediaria, o error de transmisión de red. Se distingue: omisión de envío, de recepción y de canal.
- **Fallos arbitrarios (Bizantinos)**: peor semántica posible. El proceso puede establecer valores incorrectos, devolver un valor incorrecto en respuesta a una invocación u omitir/realizar pasos arbitrarios no previstos. Un canal puede corromper mensajes, entregarlos duplicados o inventarlos.
- **Fallos de temporización**: solo en SD síncronos. Ocurren cuando el tiempo de ejecución de un proceso, el retardo de mensajes o la deriva del reloj exceden las cotas definidas.

Enmascaramiento de fallos: ocultar un fallo o convertirlo en uno más aceptable.
Ej.: réplicas enmascaran un crash (disponibilidad); sumas de comprobación convierten un fallo arbitrario (corrupción) en fallo por omisión (mensaje descartado).

Fiabilidad de la comunicación = **validez** + **integridad**:

- **Validez**: cualquier mensaje en el buffer de salida se entrega *eventualmente* al buffer de entrada del receptor (ningún mensaje se pierde indefinidamente).
- **Integridad**: el mensaje recibido es idéntico al enviado, y *no se entregan mensajes duplicados*. Las amenazas: protocolos que retransmiten sin rechazar duplicados (solución: números de secuencia) y usuarios malintencionados que inyectan, reproducen o manipulan mensajes.

Paradigmas de Comunicación Indirecta: Tipos Adicionales

Además de colas de mensajes y pub/sub, el modelo arquitectónico contempla:

- **Comunicación en grupo (*group communication*)**: entrega de mensajes a un **conjunto** de destinatarios (uno-a-muchos). Los destinatarios se unen a un grupo con un identificador; el emisor envía al grupo sin conocer a los miembros individuales.
- **Espacios de tuplas (*tuple spaces*)**: los procesos depositan **tuplas** (elementos de datos estructurados) en un espacio compartido persistente; otros procesos las leen o eliminan especificando **patrones de interés**. Desacoplamiento en espacio y tiempo. Ej.: Linda, JavaSpaces.
- **Memoria compartida distribuida (DSM)**: abstracción que permite a procesos sin memoria física compartida leer y escribir estructuras de datos compartidas *como si estuvieran en su espacio de direcciones local*, ocultando la distribución.

TEMA 5 — Desarrollo de Sistemas Distribuidos

Definiciones Fundamentales

Downsizing: migra *legacy systems* a computadoras más pequeñas y baratas dividiéndo (GUI, lógica y datos).

Rightsizing: ídem que downsizing pero a una plataforma adecuada según características de las máquinas (e.g. bases de datos en *mainframes*).

Upsizing: conecta varias aplicaciones para que sean utilizadas como un todo desplazando algunos datos y parte del procesamiento al PCD para ser compartidas.

Los 3 configuraciones de 3-etapas vistas (anterior tema) proporcionan la **estrategia para la transición de aplicaciones centralizadas a distribuidas**.

Middleware:

- Software necesario para convertir una aplicación centralizada en distribuida.
- Capa intermedia que facilita la comunicación, despliegue y configuración de una aplicación distribuida.

PCD (Plataforma de Computación Distribuida):

$$\text{PCD} = \text{Hardware} + \text{Software (S.O.} + \textit{middleware} + \ldots)$$

Metodología de Desarrollo de SD

4 Formas Básicas de Desarrollo

Ingeniería de aplicaciones distribuidas: tiene que abordar construcción, comprobación, explotación y mantenimiento.

Formas básicas:

1. Desarrollo de nueva aplicación dada una PCD.
2. Distribuir (*downsizing*) en una PCD existente.
3. Desarrollo de nueva aplicación y PCD (p.e. STR).
4. Desarrollo de un PCD dado un conjunto de aplicaciones distribuidas (*upsizing*).

Ciclo de Vida del Desarrollo de SD

Lado izquierdo (control, top→bottom) / Lado derecho (implantación+comprobación, bottom→top):

- **Definición de Requisitos y Análisis** (rendimiento, interoperatividad, portabilidad, integración, ...)
- **Arquitectura Global**: Partición · Asignación · Especificación intersistema
- **Arquitectura Subsistemas (0..n)**: Partición · Asignación · Especificación intersistema
- **Diseño Detallado**: Paradigmas de Programación · Diseño local · Entornos de soporte

⇒ **Implantación Local y Comprobación**: Implantación software · Implantación BD · Comprobación

⇒ **Implantación Subsistema y Comprobación**: Implantación del interfaz · Síntesis del subsistema · Comprobación del subsistema

⇒ **Implantación Global y Comprobación**: Implantación del interfaz · Síntesis Global · Comprobación del sistema global

⇒ **Comprobación del Usuario Final y Operación del Sistema**

Estrategias Rightsizing

Las elecciones de arquitectura se realizan a **diferentes niveles** (global, subsistema, ..., local); el número de niveles depende del tamaño del PCD.

Hay que tomar decisiones según **estrategias *rightsizing*** suponiendo existencia de PCD:

1. Análisis de la aplicación centralizada.
2. Identificar aplicaciones que no se dividen.
3. Aplicaciones a dividir: distribuir gradualmente interfaces de usuarios, programas y datos.

Arquitectura de SD

Definición (ISO/IEC/IEEE FDIS 42010:2011)

Architecture: *"Fundamental concepts or properties of a system in its environment, embodied in its elements, relationships, and in the principles of its design and evolution."*

La arquitectura se puede definir como una **"estructura con una visión"**: proporciona una vista integrada de un sistema siendo estudiado o diseñado, combinando estructura con intención y principios de diseño.

La arquitectura es a la vez:

- **Producto** que guía a: *gestores* (diseño de procesos de negocio) y *desarrolladores* (construcción del software alineada con objetivos y políticas de negocio).
- **Proceso** que consiste en los pasos para el diseño, implementación, mantenimiento y cambio del sistema operacional: Idea → Design (modelos formales, análisis) → Use (visualización para stakeholders) → Management (mantenimiento, control de versiones).

Qué debe especificar la Arquitectura

- **Componentes funcionales**: procesos, datos, GUIs, servicios.
- **Asignación de componentes** a máquinas de la PCD.
- **Modelos de interconectividad** (C/S, P2P, ...) entre los componentes de las aplicaciones.

- **Características del PCD** que soportará la aplicación (SO, lenguajes/entornos, BD, middleware, ...).
- Modelos **estructurales** (diagramas de clases UML) y de **comportamiento** (diagramas de estados, secuencia, ...).

La **metodología** combina técnicas matemáticas (teoría de colas, modelos de rendimiento) con heurísticas y reglas de recorte. Se genera un **modelo de aplicación** asociado a un **modelo de PCD**.

Proceso Arquitectónico: 3 Fases

Entradas: Modelo de Aplicación + PCD + Requisitos de Gestión + Estándares + Requisitos de Interoperatividad/Portabilidad.

1. **Descomposición / Partición**:
   - El sistema de aplicación se descompone en subpartes que se agrupan en **objetos**: datos (diversas unidades) y lógica (clientes y servidores).
   - **Posibilidades de asignación** de K objetos en n computadoras: $K \times 2^n$ (un mismo objeto puede asignarse a varias máquinas para replicación). El espacio de decisión es enorme.
   - **Partición** (agrupación de subpartes) para *reducir* el espacio de posibilidades:
     - Identificar objetos con localización específica (requisitos de usuario: datos locales al usuario; seguridad: datos confidenciales en nodo seguro; gestión: datos administrativos en nodo central).
     - Agrupar objetos para alcanzar objetivos: minimizar comunicaciones entre grupos, maximizar paralelismo, agrupar datos relacionados (compras).
     - Combinar máquinas en subredes para simplificar el problema de asignación.
2. **Asignación**: determinar en qué nodo(s) ejecuta cada componente/grupo de datos.
   - **Asignación estática**: fijada en tiempo de diseño/despliegue. Simple pero inflexible.
   - **Asignación dinámica**: varía en tiempo de ejecución (balanceo de carga, migración de datos, elasticidad cloud).
   - Estrategias básicas: **Centralización** (datos delicados o críticos en un único nodo controlado), **Duplicación** (copias de solo lectura en múltiples nodos, ej.: servidor de nombres DNS), **Replicación** (copias con actualización coordinada automática; requiere protocolos de consistencia).
   - **Coste/beneficio** estimado en: costo de almacenamiento, costo de desarrollo, tráfico en comunicaciones, tiempo de respuesta, disponibilidad.
3. **Interconexión**: elección de mecanismos de comunicación entre los componentes.
   - Sistemas C/S o BD, paradigmas de comunicación (RPC, mensajería, pub/sub), lenguajes de programación, middleware de comunicación.

Modelo de Referencia RM-ODP

Definición

RM-ODP (*Reference Model for Open Distributed Processing*): estándar **ISO/IEC 10746**.

Modelo de referencia que proporciona un **marco de trabajo para la estandarización del Procesamiento Distribuido Abierto**. Soporta distribución, independencia de plataforma y tecnología, y portabilidad, además de una **arquitectura de empresa**.

4 Recomendaciones Básicas del Estándar

1. **Introducción**: ámbito, justificación, explicación de conceptos clave y arquitectura general del modelo.
2. **Bases**: definición de conceptos y marco de análisis para normalización.
3. **Arquitectura**: especificación de las características que califican a un SD como "abierto". Define los *puntos de vista* (viewpoints), la subdivisión del sistema y la relación entre sus componentes.
4. **Semántica arquitectónica**: formalización de los conceptos de modelado interpretando conceptos en términos de construcciones de diferentes técnicas estándar de descripción formal.

5 Puntos de Vista (Viewpoints)

Un **punto de vista** (*viewpoint*) es una **subdivisión de la especificación de un sistema**. Cada uno de los siguientes puntos de vista utiliza los mismos conceptos básicos.

Analogía: igual que en dibujo técnico se representa un objeto 3D mediante varias proyecciones (alzado, perfil, planta), RM-ODP representa un SD distribuido desde 5 puntos de vista complementarios e igualmente necesarios para una especificación completa.

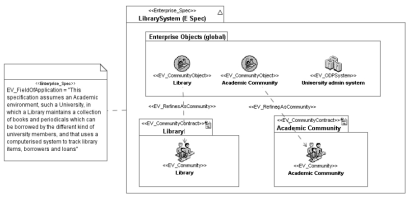
Viewpoint	Descripción detallada
Empresa (*Enterprise*)	Propósito, ámbito y **políticas del sistema** que describen los **requisitos de negocio** (p.ej.: un producto o toda una organización) y cómo alcanzarlos.
Información (*Information*)	**Tipos y semántica de la información** manejados en el sistema, cómo se comparte —p.ej., permisos— y su procesamiento.
Computacional (*Computational*)	**Descomposición funcional del sistema** en objetos que interaccionan a través de interfaces, permitiendo la distribución del sistema.
Ingeniería (*Engineering*)	**Infraestructura requerida** para dar soporte a la distribución en el sistema, i.e. las interacciones a nivel abstracto entre componentes computacionales del sistema (p.ej.: servicios de *middleware*).
Tecnología (*Technology*)	**Elección de tecnologías** para procesamiento, funcionalidad y presentación de la información, hardware y software del sistema distribuido.

Ejemplo: Biblioteca Templeman (Univ. Kent at Canterbury)

Aplica los 5 viewpoints a un sistema de préstamo de biblioteca universitaria (ver diagramas UML):

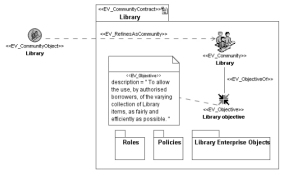
- **Empresa**: comunidades Library / Academic Community / Univ. admin; contratos; procesos de negocio (BorrowItem, ReturnItem, FineMember...)
- **Información**: esquemas invariantes/estáticos; tipos de acciones con parámetros; máquina de estados del *Borrower*: Applied → Eligible/Refused → [Normal ↔ Fined ↔ Suspended] → Archived.
- **Computacional**: *LibrarySystemMainFunctionality* con interfaces IFineInfo, IItemInfo, IMemberInfo, ILoanInfo, IBorrow, IItemMgmt, IMemberMgmt, ILibraryMgmt.
- **Ingeniería**: nodos, cápsulas, BEOs y canales de comunicación.
- **Tecnología**: instancias concretas (PC, LAN, WAN, AppServer, EIS Server).

- **Punto de vista de la Empresa**



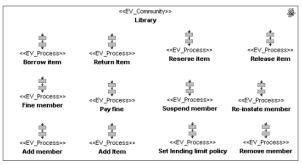
Empresa 1: Spec. empresarial global (LibrarySystem E Spec)

- **Punto de vista de la Empresa**



Empresa 2: Community Library en UML

- **Punto de vista de la Empresa**

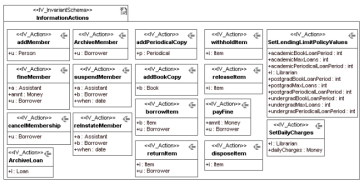


Empresa 3: Procesos de negocio (BorrowItem, ReturnItem...)

- **Punto de vista de la Información**

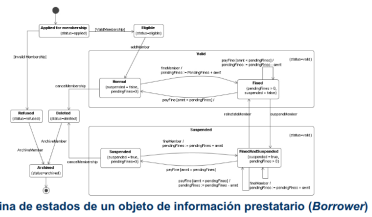
Información 1: Esquemas invariantes y estáticos

- **Punto de vista de la Información**



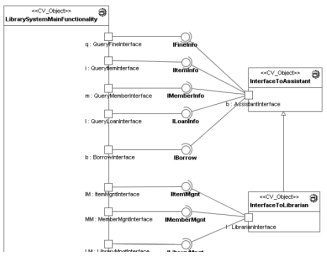
Información 2: Tipos de acciones con parámetros

- **Punto de vista de la Información**



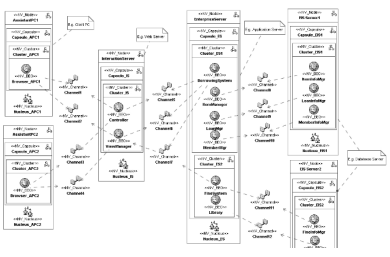
Información 3: Máquina de estados del Borrower

- **Punto de vista de la Computación**



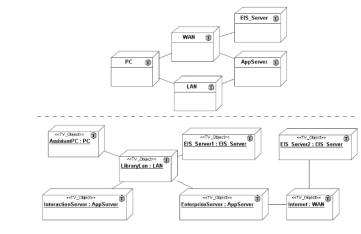
Computacional: LibrarySystemMainFunctionality e interfaces

- **Punto de vista de Ingeniería**



Ingeniería: Nodos, cápsulas, BEOs y canales

- **Punto de vista de Tecnología**



Tecnología: Instancias concretas (PC, LAN, WAN, AppServer)

Replicación

Replicación: técnica para **automáticamente mejorar los servicios** (no siempre es necesaria ni apropiada). Consiste en mantener **múltiples copias** (*réplicas*) de datos u objetos en distintos nodos del sistema distribuido, de forma que el conjunto se presente como un único objeto lógico.

Motivaciones: ¿Por qué replicar?

1. **Rendimiento**:
   - **Caché vs. replicación**: la caché almacena copias *parciales* y *no necesariamente completas* de los datos (páginas web, bloques de BD). La replicación mantiene copias *completas* y *coordinadas*.
   - Trivial para **datos inmutables** (solo lectura): cualquier copia es válida siempre. Sin overhead.
   - Costoso para **datos mutables**: cada actualización debe propagarse a *todas* las réplicas de forma consistente. Este coste es el principal desafío de la replicación.
2. **Disponibilidad**:
   - **Fallos en servidor**: con n servidores, cada uno con probabilidad p de fallo (independiente):

     $$\boxed{\text{Disponibilidad} = 1 - p^n}$$

     Ejemplo: $p = 0{,}01$, $n = 1 \Rightarrow 0{,}99$ (dos nueves); $n = 3 \Rightarrow 1 - 10^{-6}$ (seis nueves).
   - **Desconexiones**: el usuario puede trabajar con una copia local, pero surgen **conflictos de actualización** al reconectar si otros también han actualizado (problema de merge).
3. **Tolerancia a Fallos**:
   - Alta disponibilidad *no equivale* automáticamente a **datos estrictamente correctos**: una réplica disponible puede estar desactualizada.
   - Un servicio tolerante a fallos debe garantizar **comportamiento correcto** incluso durante fallos: datos actualizados, consistentes tras operaciones, entregados a tiempo.
   - Importante garantizar que **al menos una réplica puede proporcionar el servicio**, o que quedan suficientes servidores (sistema de **votación por quórum**) en caso de fallos en sistemas complejos y coordinados (fallos Bizantinos).

Requisitos Comunes de la Replicación

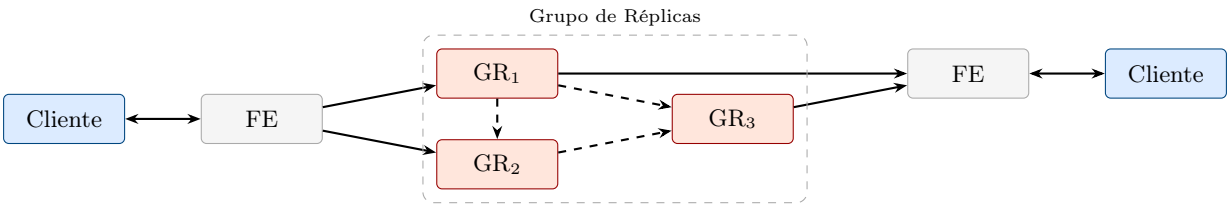
- **Transparencia**: los clientes (a) no advierten la existencia de varias copias físicas de los mismos datos (organizados como objetos lógicos individuales únicos), y (b) esperan que las operaciones devuelvan un único conjunto coherente de valores aunque internamente se ejecuten en varias réplicas físicas.
- **Consistencia**: las operaciones sobre un conjunto de objetos replicados producen resultados que cumplen las *especificaciones de corrección* del sistema. Puede ser más o menos estricta según la aplicación. Las réplicas *no necesitan ser idénticas en todo instante* (pueden existir inconsistencias temporales, ej.: por desconexiones o actualizaciones en vuelo).

Modelo General: Gestores de Réplica (GR) y Front-End (FE)

- **Gestor de Réplica (GR)**: componente que mantiene y gestiona una réplica de los datos. Puede ser un servidor dedicado (sistema C/S) o la propia aplicación cliente (útil cuando hay desconexiones frecuentes). Para garantizar consistencia, cada GR debe:
  - Tener capacidad de **recuperación** en caso de fallo (estado persistente + log).
  - Ser **sin estado** entre peticiones (reproducible a partir del log).
  - Aplicar operaciones de forma **atómica** y **determinista** (misma entrada ⇒ misma salida/estado siempre).

  El conjunto de GRs puede ser **estático** (fijo, conocido) o **dinámico** (los GRs entran/salen en tiempo de ejecución).
- **Front-End (FE)**: intermediario entre el cliente y el sistema de réplica. Recibe las peticiones del cliente, las comunica a uno o varios GRs y devuelve la respuesta al cliente. Puede estar integrado en el cliente o ser independiente.

Arquitectura general:



Fases Principales del Proceso de Replicación

1. **Comunicación**: el FE comunica la petición a 1 o n GRs (según el modelo de replicación).
2. **Coordinación**: los GRs se coordinan entre sí para garantizar:
   - **Ordenación** de mensajes: FIFO (mismo orden que el emisor), Causal (respeta causalidad), Total (todos los GRs procesan en el mismo orden absoluto).
   - **Acuerdo** en caso de fallo de algún GR.
3. **Ejecución**: los GRs ejecutan la petición. Puede ser *tentativa* (sin confirmar aún el resultado final), esperando la fase de acuerdo.
4. **Acuerdo**: los GRs alcanzan acuerdo sobre el efecto de la petición antes de *consumarla* (hacerla visible). Ej.: protocolo de commit en 2 fases (2PC) para transacciones distribuidas.
5. **Respuesta**: 1 o n GRs responden al FE, que responde al cliente.
   - **1 GR responde**: suficiente para alta disponibilidad (cualquiera que haya procesado correctamente).
   - **Mayoría de GRs responden**: necesario para tolerar fallos Bizantinos (en los que un GR responde incorrectamente, no solo falla).

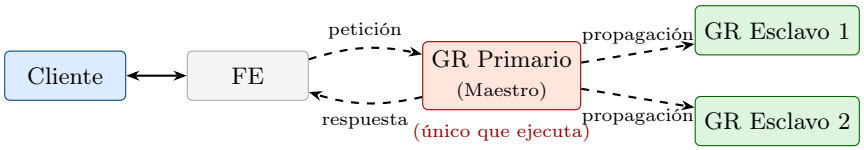
Tipos de Replicación

1. Replicación Pasiva (Primary-Backup / Maestro-Esclavo)

Un único GR **primario** recibe y ejecuta *todas* las peticiones. Propaga el estado actualizado a los **GRs esclavos** (backups). Si el primario falla, un esclavo es elegido nuevo primario.

- Los FEs se comunican **solo con el GR primario** para peticiones de actualización (lectura puede ir a esclavos).
- El primario ejecuta la operación y envía actualizaciones (estado/log) a los esclavos mediante propagación.
- Los esclavos *no* procesan peticiones directamente; solo mantienen una copia actualizada.
- Si el primario falla, se ejecuta un protocolo de elección para designar un nuevo primario entre los esclavos.
- **Ventaja**: simple, consistencia garantizada (un solo punto de decisión).
- **Desventaja**: el primario es cuello de botella y punto único de fallo lógico (aunque los esclavos cubren el fallo físico).

Diagrama:

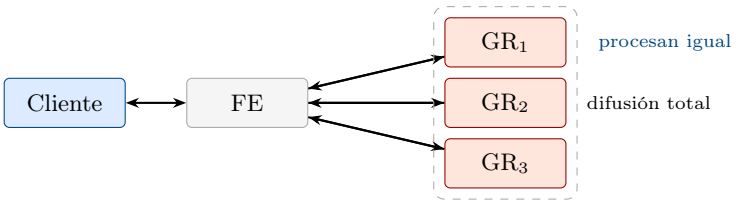


2. Replicación Activa

Todos los GRs están organizados como un **grupo** y procesan *todas* las peticiones simultáneamente e independientemente. Se usa **difusión totalmente ordenada** (*totally-ordered multicast / atomic broadcast*) para garantizar que todos los GRs reciben los mensajes en el mismo orden.

- Los FEs comunican peticiones al **grupo completo** de GRs mediante multicast totalmente ordenado.
- Como todos reciben los mensajes en el mismo orden, los GRs procesan las mismas operaciones en el mismo orden $\Rightarrow$ llegan al mismo estado $\Rightarrow$ no necesitan fase de acuerdo adicional.
- Cada GR procesa y responde **independientemente** (el FE toma la primera respuesta o las valida por mayoría).
- La caída de un GR **no tiene impacto en el rendimiento**: los demás continúan sin interrupción.
- **Ventaja**: alta disponibilidad y tolerancia a fallos sin degradación de rendimiento.
- **Desventaja**: mayor tráfico de red (todos reciben todos los mensajes); el multicast totalmente ordenado es costoso.

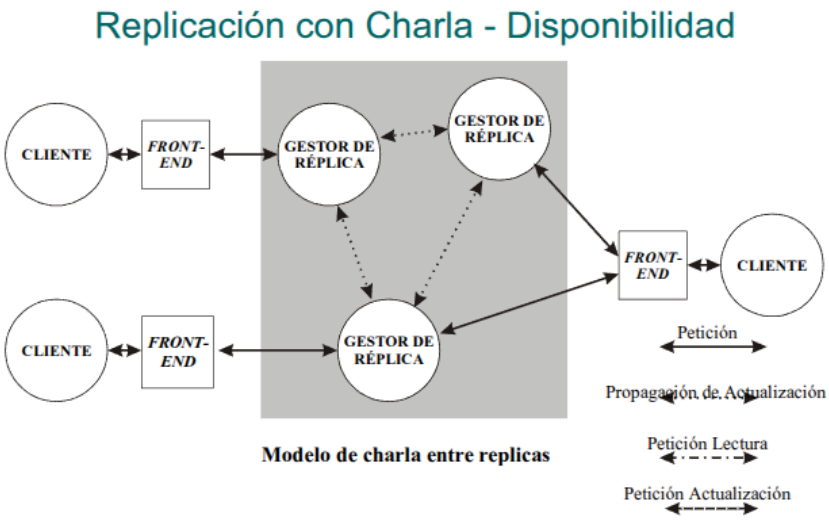
Diagrama:



3. Replicación con Charla (Gossip Protocol)

Los GRs intercambian **mensajes periódicos** ("charla" / gossip) para propagarse actualizaciones mutuamente de forma epidémica. Consistencia relajada con estrategia optimista o pesimista.

- Los FEs pueden comunicar peticiones a **cualquier GR** que elijan (por proximidad, disponibilidad, carga).
- Periódicamente, cada GR selecciona aleatoriamente otro GR y le envía las actualizaciones que tiene y el otro no. Proceso *epidémico*: las actualizaciones se propagan por el sistema de forma gradual.
- Estrategia **optimista** (asíncrona): el GR procesa localmente y responde al cliente de inmediato; la propagación al resto ocurre después. El cliente puede ver datos temporalmente inconsistentes.
- Estrategia **pesimista** (síncrona): el GR espera confirmación de un quórum antes de responder.
- Los clientes obtienen un servicio *eventualmente* consistente a lo largo del tiempo (*eventual consistency*).
- La caída de un GR **no tiene impacto en el rendimiento** de los demás.
- Adecuado para entornos con desconexiones frecuentes y redes móviles.



Modelo de charla entre réplicas — FEs eligen cualquier GR; GRs se propagan actualizaciones periódicamente (flechas punteadas)

Consistencia en Replicación: Síncrona vs. Asíncrona

Tipo	Comportamiento	Estrategia
Asíncrona	Las operaciones se aplican primero en el GR local y las actualizaciones se propagan *después* a los demás (inmediatamente, convenientemente o periódicamente). El cliente continúa tras el procesamiento local.	**Optimista**: asume que los conflictos son raros; los resuelve a posteriori.
Síncrona	Las peticiones de actualización se ordenan *totalmente* en todas las réplicas. El cliente *espera bloqueado* hasta que la petición ha sido procesada y confirmada por **todos** los GRs.	**Pesimista**: previene conflictos garantizando que todos tienen el mismo estado antes de responder.

Técnicas para Balancear Consistencia, Rendimiento y Disponibilidad

1. **Esquemas basados en *quórum***:
   - Una operación se considera completada cuando un **subconjunto mínimo** (quórum) de GRs la ha confirmado. No es necesario esperar a todos.
   - Permite cierto grado de **procesamiento concurrente**: algunos GRs pueden continuar procesando otras peticiones antes de que la operación actual complete en todos.
   - Interesante en **redes móviles** con particionamiento y desconexiones frecuentes, donde esperar a todos los GRs podría ser imposible.
   - Ejemplo: con 5 GRs, un quórum de lectura de 3 y escritura de 3 garantiza que siempre hay solapamiento de al menos 1 GR entre cualquier operación de lectura y escritura (consistencia de quórum).
2. **Causalidad** (*causal consistency*):
   - Permite un **mayor grado de concurrencia** al relajar el requisito de orden total: solo se requiere que las peticiones causalmente relacionadas (A causa B) sean procesadas en el mismo orden por todos los GRs. Las peticiones independientes (concurrentes) pueden llegar en diferente orden.
   - Usa **relojes vectoriales** para rastrear las dependencias causales entre operaciones.
   - Es más débil que la consistencia secuencial pero más fuerte que la eventual.

Resumen Visual: Conceptos Clave para el Examen

Tabla Comparativa: Tipos de Replicación

Aspecto	Pasiva (P-B)	Activa	Gossip
¿Quién ejecuta?	Solo el GR primario	Todos los GRs	El GR contactado
FE comunica a	Solo al primario	Todos (multicast total)	Cualquier GR
Consistencia	Fuerte	Fuerte	Relajada (eventual)
Fallo de GR	Elección de nuevo primario	Sin impacto	Sin impacto
Comunicación entre GRs	Propagación de estado	Difusión totalmente ordenada	Mensajes periódicos
Concurrencia	Baja (cuello de botella)	Alta	Alta

Tabla: Puntos de Vista RM-ODP

Viewpoint	Pregunta central	Artefactos típicos
Empresa	¿Para qué existe? ¿Qué políticas y objetivos?	Diagrama de comunidades, procesos de negocio
Información	¿Qué información maneja y cómo evoluciona?	Diagrama de clases, máquinas de estado
Computacional	¿En qué objetos/servicios se descompone?	Diagrama de componentes, interfaces
Ingeniería	¿Cómo se distribuye la infraestructura?	Diagrama de despliegue, nodos, canales
Tecnología	¿Qué tecnología concreta se usa?	Diagrama de red, inventario tecnológico

Fórmulas Clave

$$\boxed{\text{Disponibilidad con replicación} = 1 - p^n}$$

$$\boxed{\text{PCD} = \text{Hardware} + \text{SO} + \text{Middleware}}$$

$$\boxed{\text{Asignaciones posibles} = K \times 2^n \quad (K \text{ objetos, } n \text{ máquinas})}$$

Paradigmas: Tabla Rápida

	Petición/Respuesta	Conversacional	Mensajes Encolados / Pub/Sub
Estado en servidor	Sin estado	Con estado (sesión)	Sin estado / depende
Sincronía	Síncrono (bloqueante)	Síncrono por intercambio	Asíncrono
Desacoplamiento	Bajo	Bajo	Alto (espacio + tiempo)
Ejemplo	HTTP REST	FTP, SSH	RabbitMQ, Kafka, Socket.IO

SOA vs. MSA: Resumen

	SOA	MSA
Granularidad	Gruesa (versátil, multipropósito)	Fina (propósito único)
Middleware	ESB pesado (lógica en el bus)	API Gateway ligero
Coordinación	Orquestación centralizada	Coreografía descentralizada
BD	Puede ser compartida	Una BD por servicio
Equipo	Grandes equipos funcionales	Equipos pequeños "you build it, you run it"
Despliegue	Menos frecuente, más pesado	Continuo (CI/CD), contenedores

Kubernetes

Kubernetes (K8s): plataforma open source para **orquestar contenedores**. Define APIs centradas en aplicaciones para gestionar cómo los contenedores se aprovisionan con almacenamiento, red y seguridad. Permite la **reconciliación continua** del estado deseado de todos los despliegues de aplicaciones.

Motivación: resolver el **problema de infrastructure drift** — la configuración de los servidores diverge con el tiempo (parches distintos, actualizaciones no replicadas). Kubernetes gestiona el estado de forma declarativa mediante YAML/JSON.

Contenedores e Imágenes OCI

Un **contenedor** es una instancia en ejecución de una **imagen OCI** (Open Container Initiative). Una imagen OCI es un *tarball con capas*, donde cada capa contiene binarios Linux y ficheros de la aplicación. El container runtime (Docker, containerd, CRI-O) desempaqueta la imagen y lanza el proceso en el host.

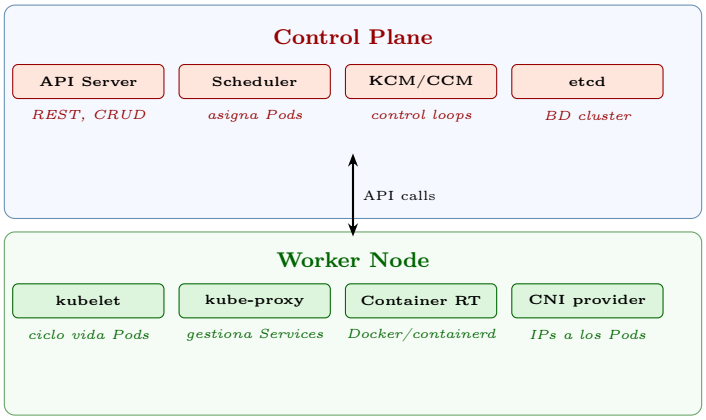
Ventajas: aislamiento de dependencias entre aplicaciones (dos apps con versiones distintas de la misma librería coexisten en contenedores separados). Resuelve el problema "*it runs on my machine*": la misma imagen ejecuta igual en cualquier host.

- **Image registry**: repositorio de imágenes (Docker Hub, Harbor, etc.). Los nodos descargan imágenes en tiempo de ejecución.
- **Inmutabilidad**: usar siempre tag de versión o SHA, *nunca* `latest` en producción (el tag no es inmutable, el SHA sí).
- **Servidores inmutables**: K8s + contenedores permiten correr servidores inmutables — el SO base solo cambia al actualizar el SO entero.

Terminología Clave

Término	Definición
Pod	Unidad atómica mínima desplegable. Encapsula uno o más contenedores que comparten red y almacenamiento. Usa *Linux namespaces* (PID, net, IPC, cgroup, mnt, user).
Node	Máquina (VM o física) que ejecuta un proceso `kubelet`. Unidad de cómputo del clúster.
kubelet	Agente Kubernetes en cada nodo. Gestiona el ciclo de vida de los Pods y reporta estado al API server via heartbeat.
Control Plane	Cerebro del clúster. Incluye API server, scheduler, KCM, CCM y etcd.
kubectl	Herramienta CLI para interactuar con el API server (hace llamadas REST).
etcd	Base de datos clave-valor del clúster. Solo el API server se comunica con él.
Namespace	Agrupación lógica de objetos API dentro del clúster (aislamiento/jerarquía).

Arquitectura



Componentes del Control Plane

- **kube-apiserver**: servidor REST HTTP. Valida y gestiona CRUD sobre todos los objetos API. **Único que habla con etcd**. Todo acceso al estado pasa por él. Los **Admission Controllers** (parte del API server) autentican y autorizan cada petición; soporta webhooks para integrar sistemas externos de auth/authz.
- **kube-scheduler**: asigna Pods a nodos considerando CPU/memoria disponible, restricciones de política, **node labels** (etiquetas de nodo para scheduling en hardware específico), **Pod affinity** (atrae Pods a nodos que cumplen reglas), **Pod anti-affinity** (repele Pods de ciertos nodos) y **taints/tolerations** (un nodo con taint solo acepta Pods con la toleration correspondiente).
- **kube-controller-manager (KCM)**: ejecuta múltiples **control loops** (bucles de reconciliación): node controller, replication controller, endpoints controller, etc. Funciona como un termostato: detecta desviación del estado deseado y actúa para corregirla.
- **cloud-controller-manager (CCM)**: gestiona integración con APIs del cloud provider (load balancers externos, routes, storage). Hace K8s cloud-agnóstico. Sustituye partes del KCM dependientes de vendor.
- **etcd**: almacén clave-valor distribuido. Es la fuente de verdad del estado del clúster.

Componentes del Worker Node

- **kubelet**: agente en cada nodo. Garantiza que los Pods asignados al nodo estén corriendo. Envía heartbeat al API server.
- **kube-proxy**: gestiona reglas de red (iptables) para los Services en cada nodo.
- **Container Runtime** (Docker, containerd, CRI-O): ejecuta los contenedores via CRI.
- **CNI provider**: proporciona red y dirección IP a cada Pod.

Objetos API Principales

Objeto	Descripción
Pod	Unidad mínima. Raramente se crea directamente; lo crean objetos de nivel superior.
Deployment	El más usado. Gestiona un conjunto de Pods replicados (microservicios sin estado).
StatefulSet	Para apps con estado (BD). Garantiza naming ordinal, almacenamiento persistente ligado al Pod, arranque ordenado.
DaemonSet	Ejecuta exactamente un Pod en *cada* nodo del clúster (agentes: logging, monitoring, networking).
Job	Ejecuta un Pod como proceso batch; termina cuando completa.
ReplicaSet	Mantiene un número deseado de réplicas de un Pod (usado internamente por Deployment).

Services: Tipos de Exposición

Tipo	Descripción
ClusterIP	IP interna del clúster. Solo accesible desde dentro. Balanceo interno entre Pods.
NodePort	Abre un puerto en cada nodo del clúster. Accesible desde fuera del clúster vía `NodeIP:NodePort`.
LoadBalancer	Crea un load balancer externo en el cloud provider. Expone el servicio públicamente con IP externa.

Características Clave

- **Declarativo**: toda la infraestructura se define en YAML/JSON. `kubectl apply` aplica el estado deseado.
- **Eventualmente consistente**: K8s reconcilia continuamente el estado actual hacia el estado deseado (no garantía inmediata).
- **Service Discovery**: via DNS (CoreDNS). Los Pods se encuentran por nombre de Service.
- **Autoescalado**:
  - *HorizontalPodAutoscaler (HPA)*: más réplicas de Pods.
  - *VerticalPodAutoscaler (VPA)*: más recursos (CPU/RAM) al Pod.
  - *Cluster Autoscaler*: más nodos en el clúster.
- **Self-healing**:
  - *Pod falla*: kubelet lo detecta via **liveness probe** (sonda de salud) o porque el proceso muere, y lo reinicia.
  - *Nodo falla*: KCM detecta pérdida de heartbeat, marca nodo offline, reprograma Pods en otros nodos. Comandos: `kubectl cordon` ***node*** (impide nuevos Pods) + `kubectl drain` ***node*** (desaloja los existentes).
- **Rolling updates**: actualización sin downtime con `kubectl edit / apply / patch`. **Requisito de la aplicación**: debe soportar *graceful shutdown* (apagado ordenado: finalizar transacciones en curso, no dejar datos corruptos) para que K8s pueda reiniciarla sin pérdida de datos.
- **Almacenamiento**: **PersistentVolume (PV)** y **PersistentVolumeClaim (PVC)** desacoplan el almacenamiento del Pod. StorageClass API.
- **RBAC**: autenticación y autorización granular sobre los objetos API.

Interfaces Estándar (Pluggable)

Interfaz	Función
CNI (*Container Networking Interface*)	Proporciona IP addresses y red a los Pods. Pluggable: Calico, Flannel, Cilium...
CRI (*Container Runtime Interface*)	Define cómo K8s lanza/para contenedores. Permite cambiar runtime (containerd, CRI-O).
CSI (*Container Storage Interface*)	Permite a vendors de storage integrarse sin modificar el código K8s.

CRDs y Operators

- **Custom Resource Definition (CRD)**: permite extender la API de Kubernetes definiendo *nuevos tipos de objetos API*. Se aplica con `kubectl apply -f crd.yaml` y permite usar `kubectl` con el nuevo tipo como si fuera nativo.

- **Operator**: patrón que combina un CRD con un controlador personalizado para gestionar aplicaciones complejas con estado (ej.: bases de datos como CockroachDB, PostgreSQL). El Operator automatiza operaciones que normalmente requieren intervención humana: backups, escalado, actualizaciones, recuperación de fallos.
- **PodDisruptionBudget (PDB)**: garantiza un mínimo de Pods disponibles durante operaciones disruptivas (drains, actualizaciones).

Alta Disponibilidad (HA) y SLA

Alta disponibilidad se mide en **número de nueves** (*nines*) de uptime anual:

$$99{,}9\,\% \approx 8{,}76 \text{ h/año de downtime} \qquad 99{,}99\,\% \approx 52 \text{ min/año} \qquad 99{,}999\,\% \approx 5 \text{ min/año}$$

K8s facilita alcanzar estos SLAs mediante: replicación de Pods (Deployments), self-healing (reinicio automático), rolling updates sin downtime y scheduling multi-zona.

Control Loops

K8s es una **máquina de reconciliación de estado** basada en **control loops** (bucles de control). Cada controlador observa el estado actual, lo compara con el estado deseado, y actúa para eliminar la diferencia. Analogía: un termostato. Controlan: binding de storage, creación/escalado/migración de contenedores, rutas IP, endpoints de load balancing.

Los controladores principales: kubelet, KCM (node, replication, endpoints controllers), CCM, scheduler.

Comandos kubectl Clave

Comando	Acción
`kubectl apply -f file.yaml`	Aplica/actualiza el estado deseado definido en el fichero.
`kubectl get <tipo>`	Lista objetos (pods, nodes, deployments, services...).
`kubectl describe <obj>`	Muestra detalle y eventos de un objeto (útil para debug).
`kubectl exec -it <pod> - sh`	Acceso remoto al interior de un contenedor.
`kubectl scale -replicas N dep`	Escala un Deployment/StatefulSet/ReplicaSet.
`kubectl cordon node`	Marca nodo como no-schedulable (no recibirá Pods nuevos).
`kubectl drain node`	Desaloja todos los Pods de un nodo (para mantenimiento).
`kubectl logs <pod>`	Muestra los logs de un Pod.
`kubectl api-resources`	Lista todos los tipos de objetos API disponibles.

Cuándo NO usar Kubernetes

- **HPC** (*High Performance Computing*): los contenedores añaden latencia; si la aplicación es sensible a nano/microsegundos, K8s puede no ser adecuado.
- **Legacy**: aplicaciones con requisitos hardware/software muy específicos que no se pueden containerizar.
- **Migración con poco beneficio**: si la aplicación monolítica no se va a descomponer en componentes, la migración aporta poco.

Seminario DSD — Web de las Cosas (WoT)

Tema central: "Hacia las futuras sociedades digitales inteligentes interconectadas" — cómo los sistemas distribuidos e IoT permiten ciudades inteligentes, ecosistemas digitales y sostenibilidad.

Contexto: Ciudades Inteligentes y Society 5.0

- **Ciudad inteligente (smart city)**: mediante tecnología, garantizar sostenibilidad energética, movilidad y calidad de vida. Objetivo ODS + Agenda 2030 + España 2050.
- **Urbanización**: en 2015 el 50 % de la población mundial era urbana; se prevé el 66 % para 2050. Hoy (2026) ya: 57 % (Mundo), 75 % (Europa), 83 % (España). Nuevas previsiones: 63 %, 78 %, 85 %.
- **Society 5.0** (Japón): integración de IA, IoT, smart cities y Sistemas Distribuidos para crear una sociedad centrada en la persona y sostenible.
- Principal problema energético: **interoperabilidad** entre sistemas con protocolos distintos e incompatibles.
- Tecnología e interconexión digital clave para **España 2050** y la **Transición Digital**.

Ecosistemas Digitales Distribuidos

Aplicaciones: Edificios, Hogar, Puertos, Tráfico, Invernaderos, Salud, Industria, Agricultura, Energía, Medioambiente.

- **Parte física**: Sensores (tráfico semiautomático o automático) + Actuadores.
- **Control de sistemas físicos de manera remota** (Edge computing).
- **IoT > WoT**: el Web of Things extiende IoT añadiendo interoperabilidad web estándar.

Ejemplo — Puerto Inteligente:

- **Arquitectura + Datos Distribuidos**: seguimiento de contenedores, carga y descarga, gestión de buques. Interoperabilidad entre sistemas distribuidos.
- **Smart Mobility**: semáforos, accidentes, tráfico, flujos, transporte público, comportamiento humano.
- **Smart Buildings**: edificios verdes (autosuficientes, sostenibles, energía casi nula); datos de inventario, clima (iluminación, confort, energía, seguridad).

El Problema Central: Interoperabilidad

El mayor problema de los ecosistemas distribuidos es la **interoperabilidad**: los dispositivos usan protocolos distintos e incompatibles. Solución: **estándares abiertos** + **modelos semánticos**.

Organizaciones de estándares relevantes: **ISO, OMG, IEEE, ACM, W3C**.
W3C (World Wide Web Consortium, 450+ miembros: Amazon, Google, Oracle...):

- Publica *Recomendaciones* (gratuitas) y *Estándares* (de pago).
- Mandato principal: **Accesibilidad e Interoperabilidad** en la web.
- Creó el estándar **WoT**: publicación v1.0 en 2023, WoT 1.1 en 2023–24.

Web of Things (WoT)

WoT es la propuesta del W3C para lograr **interoperabilidad** en ecosistemas IoT: define las "cosas" como *software compatible, testeado, validado, certificado y listo para su consumo.*

Dispositivos Ciberfísicos (IoT) tienen: Almacenamiento + Conexión a red + Capacidad de Cómputo. Forman parte de WoT y tienen papel clave en **Edge Computing** (evitando tránsito en la red → menor latencia).

Ecosistemas WoT son:

- Menos estáticos → **dinámicos**
- No supervisados → **evolutivos**
- Escalables → **adaptables**

Thing Description (TD)

Cada "cosa" en WoT se describe mediante una **Thing Description**:

- **Propiedades** (*Properties*): estado observable del dispositivo.
- **Acciones** (*Actions*): operaciones que se pueden invocar sobre el dispositivo.
- **Eventos** (*Events*): notificaciones asíncronas que emite el dispositivo.

Formato: **JSON-LD** (JSON con semántica RDF). Los esquemas permiten **validación** y **parseo** de las TDs.

Arquitectura WoT: Componentes + Especificación de Componentes + Servicios de Mediación.

Preguntas Trampa Frecuentes

- **"Más réplicas siempre mejoran consistencia"**: **FALSO**. Mejoran disponibilidad y rendimiento, pero *complican* la consistencia (más coordinación necesaria).
- **"Push garantiza tiempo real"**: **FALSO**. Depende del broker, la red y los subscriptores.
- **"Microservicio = servicio pequeño"**: **FALSO**. Lo importante es autonomía, cohesión y despliegue independiente, no el tamaño.
- **"Ingeniería RM-ODP = tecnología concreta"**: **FALSO**. Eso es el viewpoint de *Tecnología*. Ingeniería describe infraestructura abstracta de distribución.
- **"Secuencial = linealizable"**: **FALSO**. Linealizable respeta el orden real global; secuencial solo el orden de cada cliente.
- **"Disponible = correcto"**: **FALSO**. Alta disponibilidad no implica datos actualizados o consistentes.

- **"Un broker centralizado escala bien"**: **FALSO**. Es cuello de botella y punto único de fallo; una red de brokers distribuye matching/routing y tolera fallos.
- **"Si sobran réplicas, el servicio sigue sin impacto visible"**: **MATIZ**. Puede haber coste por menor capacidad, reconfiguración, failure detection o pérdida de quorum.
- **"Pub/Sub garantiza seguridad igual que request/reply"**: **FALSO**. El desacoplamiento dificulta saber fuente/destino en los extremos; la seguridad requiere autenticación, autorización, control de subscripciones, confidencialidad y auditoría en broker/red.

Glosario de Conceptos Importantes

Definiciones pensadas para responder rápido y con precisión en examen.

Acoplamiento espacial Grado en que el emisor conoce la identidad o localización del receptor. Request/reply está acoplado espacialmente; Pub/Sub lo reduce porque el publicador envía al broker, no al subscriptor concreto.

Acoplamiento temporal Grado en que emisor y receptor deben estar activos al mismo tiempo. Colas y algunos Pub/Sub permiten desacoplamiento temporal almacenando mensajes o eventos.

Adaptabilidad Capacidad de cambiar componentes, subscripciones, configuración o topología sin rediseñar toda la aplicación. Pub/Sub la favorece con altas y bajas dinámicas de subscriptores.

API Gateway Componente de entrada en microservicios: expone APIs, enruta peticiones, aplica seguridad y oculta la estructura interna de servicios.

Arquitectura Estructura con visión: conceptos, propiedades, elementos, relaciones y principios de diseño/evolución de un sistema en su entorno.

Arquitectura de microservicios (MSA) Estilo arquitectónico basado en servicios autónomos, cohesivos y desplegables independientemente, organizados por dominios de negocio y expuestos mediante APIs.

Arquitectura orientada a servicios (SOA) Estilo que organiza aplicaciones como servicios reutilizables con contratos definidos, bajo acoplamiento, composición y uso frecuente de middleware para integración heterogénea.

Asignación Decisión arquitectónica que determina en qué máquinas o nodos se ubican datos, interfaces, programas o servicios. Puede ser estática o dinámica.

Asíncrono El emisor/cliente no espera a que todos los receptores/réplicas completen la operación. Mejora disponibilidad y rendimiento, pero puede relajar consistencia.

Autonomía de servicio Un servicio se desarrolla, mantiene y controla de forma independiente, con alto control sobre su entorno de ejecución.

Broker Intermediario que recibe mensajes/eventos, aplica enrutamiento o matching y los entrega a consumidores/subscriptores. Puede ser cuello de botella o punto crítico si no se distribuye.

Byzantine failure / fallo bizantino Fallo arbitrario: un componente puede comportarse de forma incorrecta, inconsistente o maliciosa. Tolerarlo suele requerir mayoría o protocolos más costosos que para crashes.

Caché Copia cercana de datos para mejorar rendimiento. No siempre mejora disponibilidad: puede contener solo parte de los datos o estar obsoleta.

Causalidad Si un evento A influye en B, deben observarse en ese orden. La consistencia causal lo respeta, pero permite distintos órdenes en eventos concurrentes independientes.

Cliente/Servidor (C/S) Clientes solicitan servicios y servidores los proporcionan. Puede organizarse en 2, 3 o n etapas.

Cloud computing Capacidad de cómputo, almacenamiento y servicios proporcionados elásticamente desde centros de datos. Favorece escalabilidad y gestión centralizada.

Cola de mensajes Intermediario que almacena mensajes hasta que el receptor puede procesarlos. Favorece comunicación asíncrona y desacoplamiento temporal.

Comunicación conversacional Cada mensaje depende del estado de conversación previo; no es autocontenido. Aumenta complejidad ante fallos y reconexiones.

Comunicación indirecta Comunicación mediante intermediario (broker, cola, grupo, espacio de tuplas, DSM). Reduce acoplamiento pero añade sobrecoste y dificultad de gestión.

Consistencia Propiedad que indica si las operaciones sobre objetos replicados cumplen la especificación de corrección esperada. Puede ser fuerte o relajada.

Consistencia eventual Si no llegan nuevas actualizaciones, todas las réplicas terminan convergiendo al mismo estado. Durante ese tiempo pueden devolver valores distintos.

Consistencia secuencial El resultado equivale a alguna secuencia legal de operaciones que respeta el orden de cada cliente, pero *no* necesariamente el orden real global. Más débil que la linealizabilidad.

Contrato de servicio Descripción de lo que ofrece un servicio: operaciones, formato de comunicación y condiciones de uso. En servicios web: WSDL.

Coordinación Mecanismo por el que réplicas o componentes acuerdan orden, estado o efecto de operaciones para mantener consistencia.

Desacoplamiento Reducción de dependencias directas entre componentes. Puede ser espacial, temporal o de sincronización.

Descomposición División de una aplicación en subpartes funcionales, datos, objetos o servicios. Previa a decisiones de partición y asignación.

Determinismo Dada la misma secuencia de entradas, una réplica produce el mismo estado y salidas. Esencial en replicación activa.

Disponibilidad Capacidad de un servicio para estar accesible y responder. No implica que las respuestas sean estrictamente correctas o actualizadas.

Downsizing Migración de sistemas legacy a máquinas más pequeñas y baratas, dividiendo interfaz, lógica y datos.

Duplicación Copia de datos o componentes para comodidad, rendimiento o disponibilidad, *sin* necesariamente coordinación automática fuerte (a diferencia de la replicación).

Edge computing Procesamiento cerca del usuario, sensor o actuador. Reduce latencia y tráfico hacia cloud, pero opera con recursos más limitados.

Evento Notificación de que ha ocurrido un cambio de estado o situación relevante. En Pub/Sub los publicadores emiten eventos y los subscriptores reciben notificaciones.

Fail-stop Modelo de fallo donde un proceso se detiene y *otros pueden detectarlo*. Más fácil de tratar que un crash no detectable o un fallo bizantino.

Fat client El cliente contiene más lógica o procesamiento. Puede reducir tráfico pero complica actualización, seguridad y mantenimiento.

Fat server / thin client La lógica se concentra en el servidor y el cliente es ligero. Facilita mantenimiento y compatibilidad, pero carga más al servidor.

Fog computing Capa intermedia entre cloud y edge/IoT que acerca cómputo y almacenamiento a redes locales o gateways.

Front-End (FE) Componente que recibe peticiones del cliente y las comunica a uno o varios gestores de réplica, ocultando al cliente la complejidad de la replicación.

Gestor de réplica (GR) Componente que mantiene una réplica física y ejecuta operaciones sobre ella. Varios GR cooperan para ofrecer un objeto o servicio lógico.

Gossip / charla Técnica de replicación donde los GR intercambian periódicamente actualizaciones. Da alta disponibilidad y soporta desconexión, pero ofrece consistencia relajada.

Heterogeneidad Existencia de distintos SO, lenguajes, protocolos, redes, dispositivos o plataformas. SOA y servicios web buscan manejarla.

HTTP Protocolo de petición/respuesta de la Web. Base de servicios web y APIs REST.

Ingeniería de aplicaciones distribuidas Disciplina que aborda construcción, comprobación, explotación y mantenimiento de aplicaciones distribuidas sobre una PCD.

Interconexión Decisión arquitectónica sobre cómo se comunican los componentes: C/S, P2P, Pub/Sub, colas, BD, RPC, RMI...

Interoperabilidad Capacidad de sistemas heterogéneos para trabajar juntos. Objetivo de SOA, Web Services, WSDL/SOAP/REST y middleware.

IoT Conjunto de dispositivos físicos, sensores y actuadores conectados, con restricciones de energía, capacidad y conectividad.

Kubernetes Plataforma de orquestación de contenedores que mantiene el estado deseado: despliega, escala, reinicia y reprograma cargas de trabajo.

Linealizabilidad Consistencia fuerte: cada operación parece ocurrir *instantáneamente* entre su invocación y respuesta, respetando el orden real global. Más fuerte que la consistencia secuencial.

Mensaje Unidad de comunicación entre procesos o componentes. Puede ser petición, respuesta, evento, notificación o elemento de cola.

Microservicio Componente autónomo y desplegable independientemente que implementa una funcionalidad cohesionada de un dominio. No tiene por qué ser pequeño.

Middleware Capa software intermedia que facilita comunicación, configuración, despliegue, seguridad, transacciones e integración en sistemas distribuidos.

Modelo arquitectónico Descripción del sistema en términos de entidades computacionales, comunicación, estilos y responsabilidades.

Modelo fundamental Modelo abstracto que razona sobre interacción, fallos o seguridad, independientemente de una tecnología concreta.

Modelo físico Descripción de nodos, redes e interconexión hardware de un sistema distribuido.

Multicast Envío de un mensaje a un grupo de receptores. Puede necesitar garantías de fiabilidad y orden para ser útil (ej.: replicación activa).

n etapas Arquitectura C/S con más de dos capas, separando presentación, lógica, datos y servicios intermedios.

Orquestación Gestión automatizada del despliegue, escalado, recuperación y ubicación de contenedores o servicios. Kubernetes es un orquestador.

P2P / Peer-to-Peer Nodos actúan como proveedores y consumidores de recursos. No depende de servidor central imprescindible; requiere algoritmos distribuidos para localizar datos y balancear carga.

Partición Agrupación de subpartes de una aplicación para reducir complejidad y facilitar asignación. También: partición de red que separa grupos de nodos.

PCD Plataforma de Computación Distribuida: hardware + SO + middleware + servicios que soportan aplicaciones distribuidas.

Petición/respuesta Un cliente envía una petición y espera respuesta del servidor. Directo y fácil de razonar, pero menos flexible ante muchos receptores o movilidad.

Pod Unidad mínima planificable de Kubernetes. Agrupa uno o más contenedores que comparten red, volúmenes y ciclo de vida.

Primary-backup Replicación pasiva: un primario ejecuta operaciones y propaga actualizaciones a backups. Si falla, un backup se convierte en primario.

Pub/Sub (Publish/Subscribe) Publicadores emiten eventos a un intermediario; subscriptores reciben notificaciones según sus intereses sin conocerse mutuamente.

Pull-based Pub/Sub Los subscriptores consultan eventos desde el broker. Permite agrupar recepción, pero introduce polling y latencia.

Push-based Pub/Sub El broker envía eventos directamente a subscriptores. Adecuado para eventos impredecibles, pero exige conexiones y gestión de entrega.

Quorum Subconjunto suficiente de réplicas para realizar una operación. Para consistencia, los quorums de lectura/escritura y escritura/escritura deben solaparse, evitando que operaciones conflictivas ocurran en conjuntos disjuntos.

Réplica Copia física de un dato u objeto lógico mantenida por un gestor de réplica.

Replicación Mantenimiento de varias copias coordinadas de datos o servicios para mejorar rendimiento, disponibilidad o tolerancia a fallos.

Replicación activa Todas las réplicas ejecutan las mismas operaciones en el mismo orden total. Requiere determinismo y difusión fiable totalmente ordenada. Si las réplicas reciben distinto orden o ejecutan no determinísticamente, divergen.

Replicación asíncrona Una actualización se aplica localmente y se propaga después. Aumenta disponibilidad pero puede generar inconsistencias temporales.

Replicación pasiva Un primario ejecuta y backups copian estado/actualizaciones. Tolera bien el no determinismo porque solo el primario decide. Costes: cuello de botella, detección de fallo y elección de nuevo primario.

Replicación síncrona La actualización se coordina con todas o suficientes réplicas antes de responder. Mejora consistencia pero aumenta latencia y reduce disponibilidad.

REST Estilo de servicios web orientado a recursos, normalmente sobre HTTP. Se contrapone a SOAP (orientado a interfaces/mensajes).

Rightsizing Migración o redistribución de un sistema a las plataformas adecuadas para cada componente, no necesariamente las más pequeñas.

RM-ODP Modelo de referencia para Procesamiento Distribuido Abierto. Define 5 vistas: empresa, información, computacional, ingeniería y tecnología.

RPC Remote Procedure Call: invocación de un procedimiento remoto como si fuera local. Encaja con petición/respuesta y modelo C/S.

RMI Remote Method Invocation: invocación de métodos sobre objetos remotos en entorno orientado a objetos distribuidos.

Servicio Colección de atributos y comportamientos ofrecidos por un recurso a través de interfaces bien definidas.

Servicio sin estado Minimiza o evita mantener estado de cliente entre invocaciones. Mejora escalabilidad y recuperación.

Síncrono El emisor/cliente espera a que la operación avance hasta cierto punto acordado. En replicación: esperar a réplicas o quorum.

SOAP Protocolo de mensajería en Web Services, con XML sobre HTTP u otros transportes.

Thin client Cliente ligero con poca lógica local. Depende más del servidor, pero facilita mantenimiento y actualización.

Timeliness / puntualidad Capacidad de acotar o garantizar tiempos de entrega/respuesta. Más difícil en Pub/Sub por desacoplamiento y dependencia del broker.

Tolerancia a fallos Capacidad de mantener comportamiento *correcto* pese a fallos. No equivale simplemente a estar disponible.

Transparencia de replicación El cliente no advierte varias copias físicas y opera sobre un objeto lógico único.

UDDI Servicio de directorio para publicar y descubrir servicios web, asociado a WSDL.

Upsizing Integración de varias aplicaciones para usarlas como un todo, desplazando datos y procesamiento a una PCD para compartirlos.

WebSocket Comunicación bidireccional persistente entre cliente y servidor, útil para notificaciones push en web.

WSDL Web Services Description Language: lenguaje para describir interfaz, operaciones y ubicación de un servicio web.