# Capítulo 1 - Introducción

## Sistemas centralizados, descentralizados y distribuidos

### Califique y justifique los siguientes sistemas bajo estos conceptos en centralizados, descentralizados y distribuidos

Gmail
- Centralizado (gestionado por google)
- Distribuido (en muchos svrs)

Spotify / Netflix
- Centralizado (hay solo una fuente de verdad)
- Distribuido (en muchos svrs)

NAS
- Centralizado

Home Assistant
- Centralizado (el home assitant es el cerebro)

– Distribuido (los otros componentes lo hacen distribuido)

IA en datacenter
- Centralizado (Una única organización lo administra)

– Distribuido (El uso de GPUs y entrenamiento)

Un cluster de virtualización de computadoras
- Centralizado (Gestionado por un OS)

– Distribuido (Muchos Compus)

Tótem en parque
- Centralizado (es un totem unico con un centro de verdad)

### Elija un sistema de su preferencia: Descríbalo, clasifíquelo y justifique.

**Descripción:** Gmail es un servicio de correo electrónico provisto por Google. Permite a los usuarios enviar y recibir correos mediante

protocolos estándar como SMTP, IMAP y POP, además de ofrecer funcionalidades adicionales como filtrado de spam, búsqueda y almacenamiento en la nube.

**Clasificación:** Centralizado – distribuido.

**Justificación:** La gestión de usuarios, autenticación, filtrado y control del servicio son centralizados bajo la administración de Google (una única autoridad). Sin embargo, para garantizar disponibilidad y escalabilidad global, Gmail se ejecuta en múltiples servidores distribuidos en datacenters a lo largo del mundo, lo que lo convierte también en un sistema distribuido a nivel de infraestructura.

**Elija un sistema centralizado pero distribuído y justifique su respuesta**

**Descripción:** Spotify es un servicio de streaming de música bajo demanda. Los usuarios acceden al catálogo a través de internet desde múltiples dispositivos.

**Clasificación:** Centralizado – distribuido.

**Justificación:** El control del servicio, el catálogo y los algoritmos de recomendación están bajo una autoridad central (Spotify). Sin embargo, la entrega del contenido está soportada por una infraestructura distribuida a través de CDNs y servidores en diferentes regiones, lo que permite alta disponibilidad y baja latencia.

**Elija un sistema descentralizado y justifique su respuesta**

**Descripción:** Mastodon es una red social de microblogging de software libre que funciona bajo el protocolo ActivityPub. Los usuarios se registran en instancias independientes (servidores) y pueden interactuar con usuarios de otras instancias gracias a la federación.

**Clasificación:** Descentralizado.

**Justificación:**

- Cada instancia de Mastodon es autónoma: administra a sus usuarios, sus reglas, su almacenamiento y su moderación.

- No existe una autoridad central que controle toda la red; las instancias se comunican entre sí mediante protocolos abiertos (federación).
- La “verdad” de los datos de cada usuario está en el servidor donde se registró, pero puede intercambiar información con otros servidores para permitir interacciones sociales entre comunidades distintas.
- Esto implica que si una instancia cae o se desconecta, el resto de la red sigue funcionando (no hay un único punto de fallo global).

**Dado un server plex corriendo en una raspberry donde tengo todos los videos familiares en el mismo disco. ¿Cuándo pasaría a ser un sistema distribuido y por qué?**
Plex pasa a ser distribuido cuando la responsabilidad de almacenamiento y/o entrega de los videos no está en un solo servidor, sino en varios nodos que cooperan para brindar el servicio.

## Problemas de los sistemas distribuidos

**Las computadoras se rompen**
**Ejemplo del problema:** si un servidor físico falla (por hardware dañado o apagado inesperado), el servicio se interrumpe porque era el único que lo ejecutaba.

**Ejemplo de solución:** desplegar el mismo software en varias computadoras en paralelo, de manera que si una se rompe, la otra puede seguir respondiendo.

**Mecanismos concretos:** clusters con balanceo de carga, failover automático, servidores redundantes. Por ejemplo, en una base de datos, tener un nodo primario y un nodo réplica listo para asumir el rol si el primario falla.

**No saben con quién comunicarse**
**Ejemplo del problema:** en un sistema distribuido, un nodo no sabe la dirección actual de otro nodo con el que necesita comunicarse (porque cambió IP, se reinició o escaló horizontalmente).

**Ejemplo de solución:** tener un servicio centralizado de descubrimiento, donde cada nodo registra su ubicación y capacidades, y otros nodos pueden consultarlo para encontrarlo.

**Mecanismos concretos:** DNS dinámico, service discovery en Kubernetes (etcd o Consul), servidores de registro para microservicios. Por ejemplo, un pod nuevo en Kubernetes se registra automáticamente y otros pods pueden encontrarlo usando el nombre del Service, sin preocuparse de la IP concreta.

**No se ponen de acuerdo**
**Ejemplo del problema:** varios nodos tienen copias del mismo dato o deben decidir quién lidera una tarea, y cada uno puede pensar algo distinto. Si no hay acuerdo, la consistencia se rompe o el sistema puede fallar.

**Ejemplo de solución:** usar algoritmos de consenso que garanticen que todos los nodos acuerden la misma decisión, incluso si algunos fallan o se retrasan.

**Mecanismos concretos:**

- Raft o Paxos: se usan en bases de datos distribuidas o sistemas de coordinación.
- Etcd en Kubernetes: asegura que todos los nodos del clúster tienen el mismo estado de configuración mediante consenso Raft.
- Zookeeper en Hadoop/HBase: coordina líderes y mantiene consistencia de metadatos.

**Se corta la comunicación**
**Qué significa:** en sistemas distribuidos, los nodos dependen de la red para coordinarse. Si hay una partición de red (un corte, fallo de enlace, congestión), algunos nodos quedan aislados y no pueden comunicarse entre sí. Esto puede provocar inconsistencias o pérdida de disponibilidad temporal.

**Ejemplo real:**

En un clúster de bases de datos replicadas, si un nodo queda aislado por un corte de red, puede que siga aceptando escrituras sin saber lo que pasó en los demás nodos (problema de particiones).

En sistemas de mensajería, un broker aislado puede dejar de recibir o enviar mensajes hasta que se restaure la conexión.

**Soluciones típicas:**

Reintentos automáticos: cuando se restablece la conexión, los nodos sincronizan el estado.

Tolerancia a particiones: diseñar el sistema según el teorema CAP, eligiendo entre consistencia o disponibilidad.

Replicación eventual: permitir que los nodos continúen operando y luego converjan cuando la red vuelva a estar disponible.

**Ejemplos concretos:**

- Apache Kafka reconfigura líderes de particiones cuando un broker queda aislado.
- Sistemas como Cassandra o DynamoDB usan consistencia eventual para sobrevivir a cortes temporales de red.

**Se pierden los bits**
**Qué significa:** durante la comunicación entre nodos, los datos pueden corromperse o perderse debido a ruido en la red, congestión o fallos físicos. Esto provoca que los mensajes lleguen incompletos o incorrectos.

**Ejemplo real:**

En protocolos de transmisión sin control, como UDP, un paquete puede perderse o corromperse.

Esto puede afectar streaming, VoIP o juegos en línea si no se implementa control adicional.

**Soluciones típicas:**

Detección de errores: checksums, CRC, hashes para identificar datos corruptos.

Retransmisión: protocolos como TCP reenvían los paquetes que se detecta que faltan o están dañados.

Redundancia o codificación: añadir información extra (FEC, Forward Error Correction) para recuperar datos sin necesidad de retransmisión.

Ejemplo concreto: TCP garantiza entrega fiable reensamblando los paquetes y reenviando los que se pierden; los videojuegos que usan UDP suelen implementar su propio sistema de verificación y retransmisión de paquetes críticos.

**Pueden venir intrusos**

**Qué significa:** en sistemas distribuidos, mensajes o solicitudes pueden ser interceptados, falsificados o enviados por actores no autorizados, comprometiendo datos y servicios.

**Ejemplo real:** un atacante que hace man-in-the-middle o envía solicitudes fraudulentas a una API de un servicio distribuido.

**Soluciones típicas:**

Autenticación: verificar que quien envía la solicitud es quien dice ser (Bearer tokens, API keys, certificados digitales).

Autorización: controlar qué acciones puede realizar cada usuario/nodo.

Cifrado en tránsito: TLS/SSL para proteger datos mientras se envían por la red.

Firewall y control de acceso a la red: limitar qué nodos pueden conectarse entre sí.

**Ejemplo concreto:**

APIs REST usan Bearer tokens o API keys para autenticar clientes antes de permitirles acceso.

Sistemas de mensajería cifrada como Signal garantizan que los mensajes solo puedan ser leídos por los destinatarios autorizados.

**Obvio… hay bugs en el código**
Qué significa: todo software puede contener errores que provoquen fallos inesperados, desde un componente hasta la caída de un servicio completo en sistemas distribuidos.

Ejemplo real: un microservicio que lanza una excepción no controlada y hace que toda la aplicación deje de responder.

Soluciones típicas:

Tolerancia a fallos: diseñar el sistema de manera que un error en un componente no afecte a los demás.

Fail-fast y recuperación rápida: los componentes detectan errores de inmediato, fallan rápido y otros nodos o procesos pueden asumir la carga.

Patrones concretos:

Circuit breaker: evita que llamadas repetidas a un componente fallido colapsen el sistema.

Supervisión de actores (Actor model, Erlang/Elixir): un supervisor reinicia procesos fallidos automáticamente.

Chaos testing: Netflix con Chaos Monkey prueba fallos para asegurarse que el sistema se recupera sin afectar al usuario final.

Ejemplo concreto: en Erlang/Elixir, cada proceso actor puede fallar sin derribar el sistema; un supervisor reinicia el proceso automáticamente.

## Arquitectura de software

## Tipos de sistemas

**Mitos**

1. La red es confiable.
2. La latencia es cero.
3. El ancho de banda es infinito.
4. La red es segura.
5. La topología no cambia.

6. Hay un solo administrador.
7. El costo de transporte es cero.
8. La red es homogénea.

**Sistema P2P como BitTorrent**

- La red es segura (falso): porque cualquiera puede conectarse, incluso nodos maliciosos que suban archivos corruptos, malware o ficheros falsos.
- La latencia es cero (falso): cada peer puede estar en otra parte del mundo, con distintas velocidades de conexión, lo que introduce retardos importantes.

**Sistema IoT en una casa inteligente**

- La red es segura (falso): las cámaras y dispositivos IoT suelen estar expuestos en internet sin suficiente protección → son un blanco fácil para ataques (ejemplo: botnets como Mirai).
- Hay un solo administrador (falso): si todo pasa por un hub central y ese nodo falla, toda la red se cae; en realidad, la administración puede estar distribuida o incluso fuera del control del usuario (fabricante, cloud).
- La red es confiable (falso): los dispositivos pueden desconectarse, tener interferencias de WiFi o incluso ser comprometidos desde dentro de la red.
- La topología no cambia (falso): los dispositivos IoT se agregan o eliminan constantemente (ej. enchufes inteligentes nuevos, sensores que se rompen).

**Sistema en la nube**

- La red es confiable (falso): fallas en data centers, caídas de servicios (ej. outage de AWS en una región).
- La latencia es cero (falso): depende de la ubicación del usuario y la región de la nube (ej. un cliente en LatAm usando un server en EE.UU. tiene más latencia).
- El ancho de banda es infinito (falso): transferir TBs entre regiones es lento y muy costoso.

- La red es segura (falso): vulnerabilidades, errores de configuración (ej. buckets S3 públicos).
- La topología no cambia (falso): las VMs, contenedores y servicios escalan dinámicamente.
- El costo de transporte es cero (falso): mover datos entre regiones o salir a internet cuesta bastante.
- La red es homogénea (falso): distintas instancias, regiones, proveedores; no todo se comporta igual.
- Hay un solo administrador (falso): hay miles de ingenieros y sistemas detrás.