Procesamiento de Grandes Volumenes de Datos

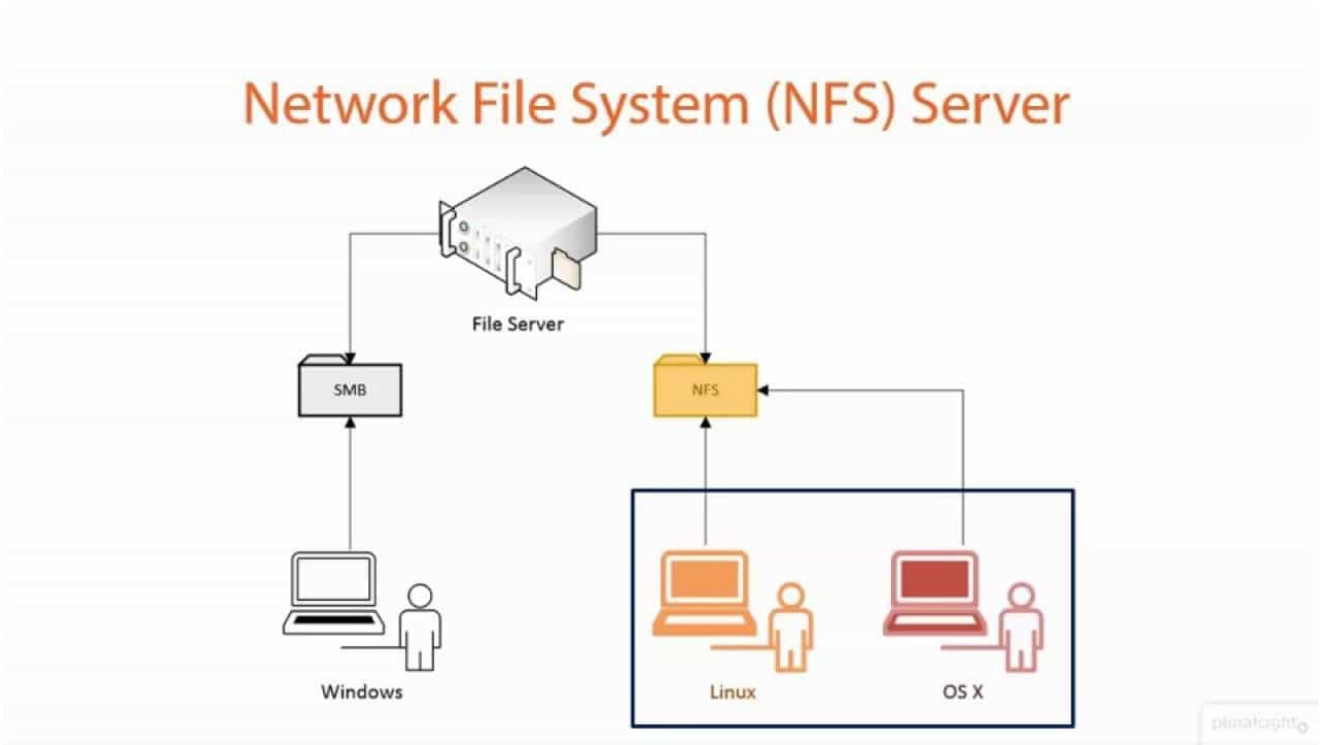
Conferencia 2: Fundamentos del procesamiento distribuido

Deborah Famadas Rodríguez

Universidad de la Habana

September 8, 2025

El Desafío de Google a Principios de los 2000



Desafíos

- **Volumen, Velocidad**
- **Fallos Constantes:** Diseñado para funcionar sobre miles de servidores *commodity* (baratos) que fallan constantemente.

¿Qué es una Base de Datos Distribuida?

Definición

Una colección de múltiples bases de datos, lógicamente interrelacionadas, que se encuentran físicamente repartidas en diferentes ubicaciones (nodos) y conectadas por una red.

- Los usuarios interactúan con el sistema como si fuera una única base de datos.
- La distribución es transparente para el usuario.

Arquitectura de Google File System (GFS)

Principios: Simplicidad, Escalabilidad y Tolerancia a Fallos

Arquitectura de Google File System (GFS)

Principios: Simplicidad, Escalabilidad y Tolerancia a Fallos

- **Plano de Control Centralizado:**
  - Un único nodo **Master** gestiona todos los metadatos: espacio de nombres, ACLs y el mapeo de archivos a sus componentes (chunks).
- **Plano de Datos Distribuido:**
  - Múltiples servidores **Chunkservers** almacenan los datos reales en sus discos locales.
  - Los archivos se dividen en **chunks** de gran tamaño (64 MB).
- **Separación de Flujos:**
  - El cliente contacta al Master solo para obtener metadatos.
  - La lectura/escritura de datos se realiza directamente con los Chunkservers, evitando que el Master sea un cuello de botella.
- **Tolerancia a Fallos:**
  - **Replicación de chunks**: Cada chunk se replica (por defecto 3 veces) en diferentes racks.
  - El estado del Master se protege mediante un log de operaciones replicado.

¿Por qué Sistemas Distribuidos? Parte 1: Escalabilidad

Escalabilidad:

Capacidad de un sistema para manejar una carga de trabajo creciente.

Escalabilidad Horizontal (Scale Out)

- Añadir más máquinas (nodos) al sistema.
- Escalabilidad casi ilimitada usando hardware genérico. Es la base de los sistemas distribuidos.

Ventajas y Desventajas

Ventajas

- Alta Disponibilidad.
- Rendimiento Mejorado (Latencia).
- Autonomía Local.

Desventajas

- **Complejidad:** Mucho más difícil de diseñar, implementar y depurar.
- **Coste:** Gestión de múltiples nodos.
- **Seguridad:** Múltiples puntos de posible ataque.
- **Consistencia de Datos:** El desafío principal.

Replicación: ¿Qué es y por qué?

Definición

Mantener copias idénticas de los mismos datos en múltiples nodos.

Replicación: ¿Qué es y por qué?

Definición

Mantener copias idénticas de los mismos datos en múltiples nodos.

Motivaciones Principales:

- **Alta disponibilidad:** Si un nodo falla, sus datos siguen disponibles en sus réplicas.
- **Recuperación:** Las réplicas pueden estar en centros de datos geográficamente distintos.
- **Escalabilidad de lectura y baja latencia:** Las solicitudes de lectura pueden distribuirse entre varias réplicas o ser atendidas por la réplica más cercana al cliente.

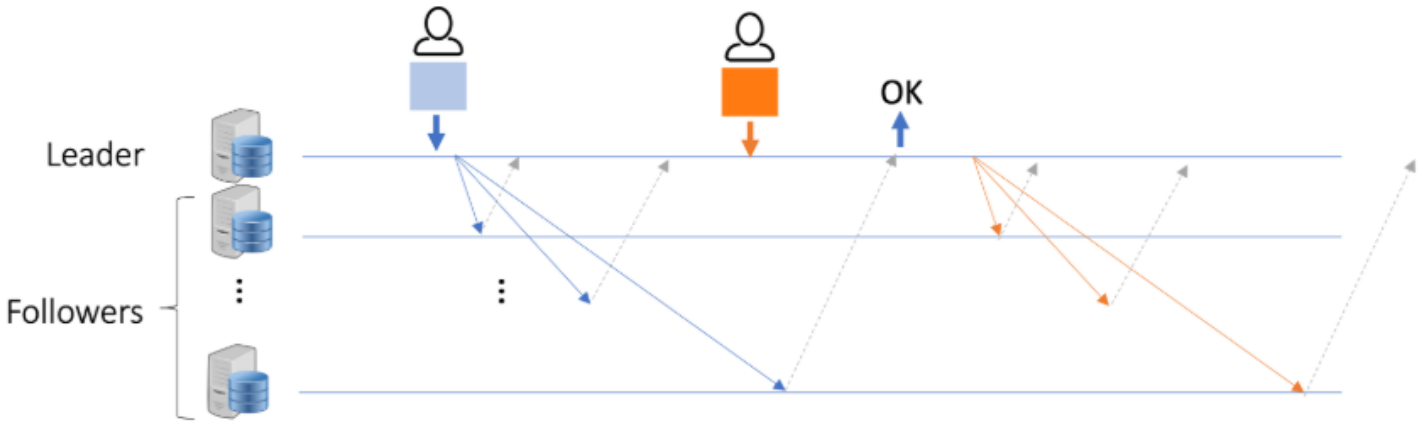
Arquitectura 1: Líder-Seguidor (Master-Slave)

- Es el modelo más común.
- **¿Cómo funciona?**:
  1. Un nodo es designado como el **Líder** (Master).
  2. Todas las escrituras (`INSERT`, `UPDATE`, `DELETE`) **deben** ir al Líder.
  3. El Líder procesa la escritura y luego propaga los cambios a todos sus **Seguidores** (Followers/Slaves).
  4. Las lecturas pueden ser atendidas por el Líder o por cualquiera de los Seguidores.

Replicación Síncrona vs. Asíncrona

Replicación Síncrona

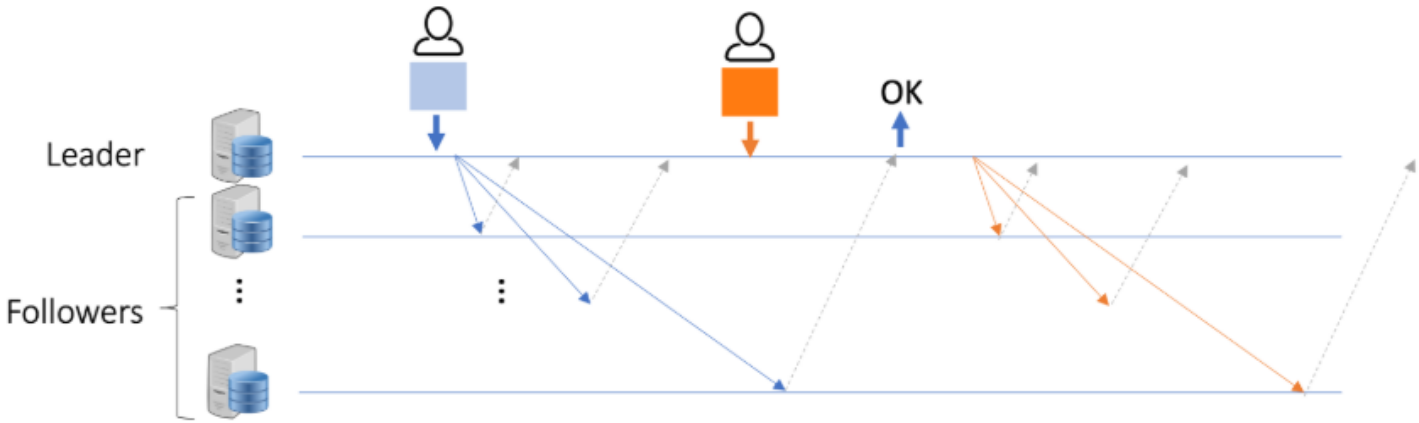
- El Líder espera la confirmación de (al menos) un Seguidor antes de responder al cliente.



Replicación Síncrona vs. Asíncrona

Replicación Síncrona

- El Líder espera la confirmación de (al menos) un Seguidor antes de responder al cliente.

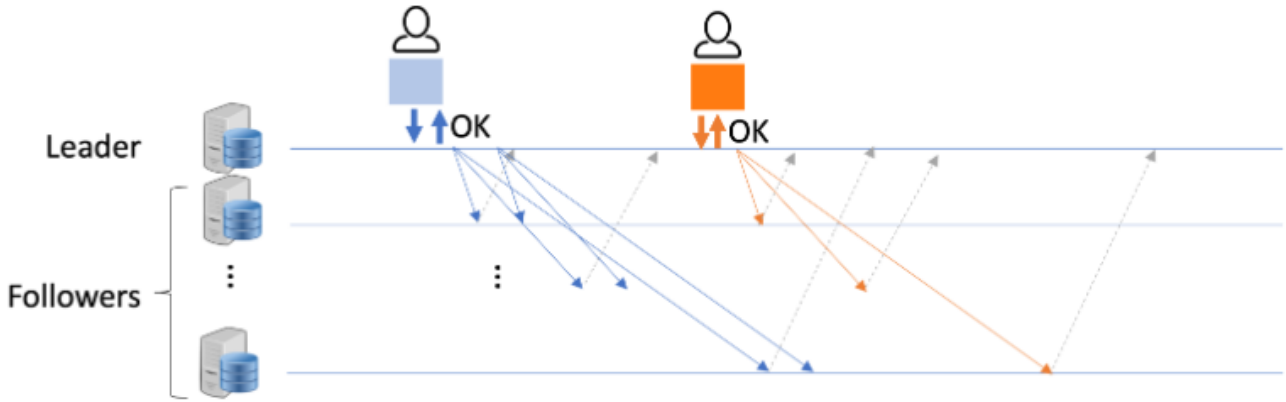


- **Pro:** Garantiza que los datos no se pierdan si el Líder falla.
- **Contra:** Aumenta la latencia de escritura.

Replicación Síncrona vs. Asíncrona

Replicación Asíncrona

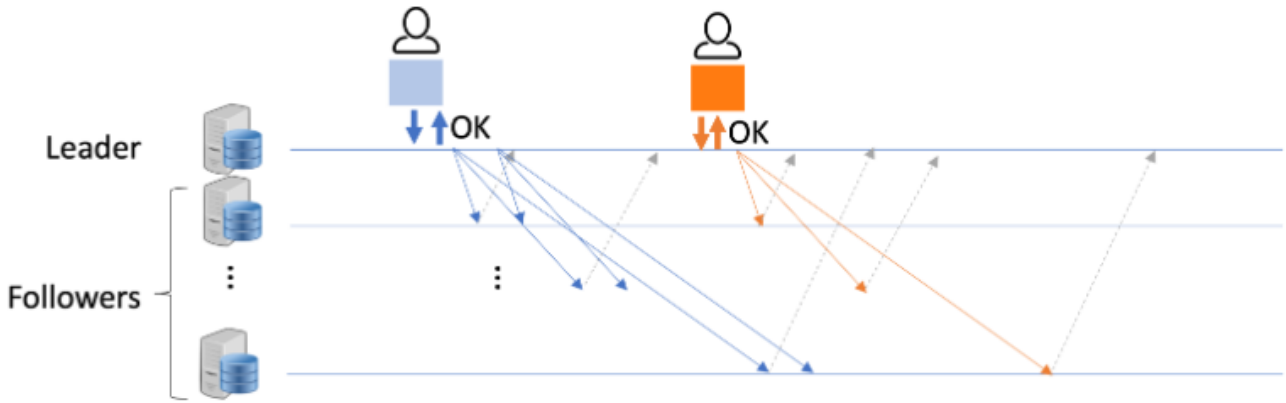
- El Líder responde al cliente inmediatamente y envía los cambios a los Seguidores en segundo plano.



Replicación Síncrona vs. Asíncrona

Replicación Asíncrona

- El Líder responde al cliente inmediatamente y envía los cambios a los Seguidores en segundo plano.



- **Pro:** Escrituras muy rápidas.
- **Contra:** Riesgo de pérdida de datos si el Líder falla antes de replicar.

Manejo de Fallos en Líder-Seguidor

Fallo de un Seguidor:

- El Seguidor se recupera a partir de los logs del Líder. El sistema sigue operativo.

Fallo del Líder:

- **Failover:** Se debe elegir un nuevo Líder entre los Seguidores.
- **Proceso de Elección**
- Riesgo de inconsistencias (cerebro dividido o split-brain).

Implementación del Log de Replicación

1. **Replicación Basada en Sentencias (Statement-Based):**
   - El Líder envía las sentencias SQL (`INSERT`, `UPDATE`) a los seguidores.
   - **Problema:** Funciones no determinísticas como `NOW()` o `RAND()` pueden dar resultados diferentes.
2. **Replicación Basada en WAL (Write-Ahead Log):**
   - Se envía el log de cambios a nivel de bytes/bloques. Acoplado al motor de almacenamiento.
3. **Replicación Lógica (Row-Based):**
   - Se envía un log de los cambios a nivel de fila (datos antiguos vs. nuevos).
   - Es el método más robusto y desacoplado.

Problema: Retraso en la Replicación (Replication Lag)

- En la replicación asíncrona, los seguidores siempre van un poco por detrás del líder.

Problema: Retraso en la Replicación (Replication Lag)

- En la replicación asíncrona, los seguidores siempre van un poco por detrás del líder.
- **Consecuencia:** Un usuario escribe un dato, lo envía al líder y luego intenta leerlo desde un seguidor. ¡Es posible que aún no vea su propio cambio!
- Esto viola la garantía de "Leer tus propias escrituras" (*Read-your-writes consistency*).

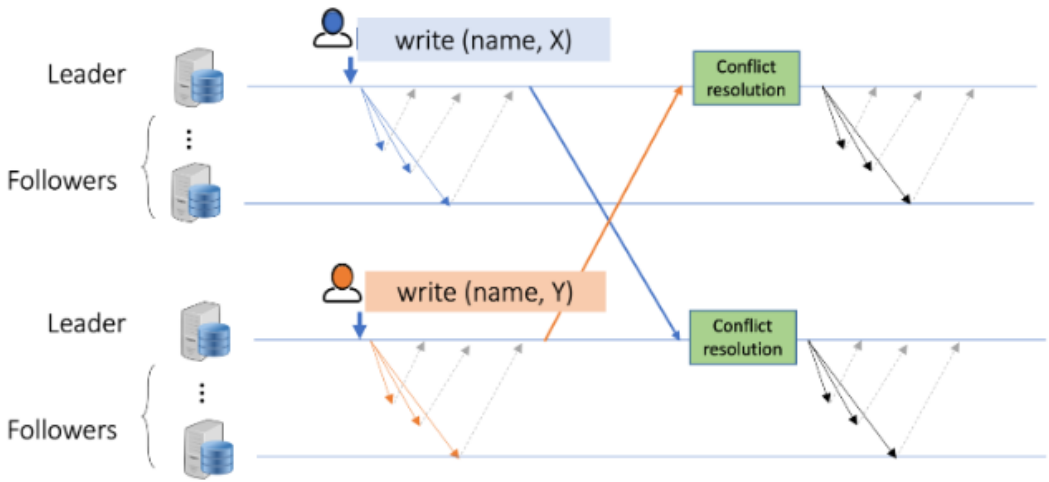
Soluciones:

- Leer desde el líder si el usuario ha escrito algo recientemente.
- Monitorear el retraso y no dirigir lecturas a seguidores muy desactualizados.

Arquitectura 2: Replicación Multi-Líder

- Más de un nodo puede aceptar escrituras, cada centro tiene un líder local para baja latencia, los líderes se sincronizan entre sí.

¿Qué pasa si el mismo dato es modificado simultáneamente en dos centros de datos?



Frame Title

- LWW (Last Write Wins).
- Lógica de Fusión Personalizada (Custom Merge Logic)
- CRDTs (Conflict-Free Replicated Data Types):

Ejemplo de Conflicto en un Sistema Multi-Líder (como Git)

Línea de Tiempo de un Conflicto

```
# t   |   User 1                        |   User 2
#-----|---------------------------------|--------------------------
# t+1| git clone git://....             |
# t+2| git add foo.c                    | git clone git://....
# t+3|                                  |
# t+4| git commit -m 'Hacked v1'|
# t+5| git push                         | git add foo.c
# t+6|                                  | git commit -m 'Hacked new'
# t+7|                                  | git push # ¡FALLA!
# t+8|                                  | git pull # ¡CONFLICTO!
```

Ejemplo de Conflicto en un Sistema Multi-Líder (como Git)

Línea de Tiempo de un Conflicto

```
# t   |   User 1                          |   User 2
#----|-----------------------------|-------------------------
# t+1| git clone git://....      |
# t+2| git add foo.c             | git clone git://....
# t+3|                           |
# t+4| git commit -m 'Hacked v1'|
# t+5| git push                  | git add foo.c
# t+6|                           | git commit -m 'Hacked new'
# t+7|                           | git push # ¡FALLA!
# t+8|                           | git pull # ¡CONFLICTO!
```

Ambos "líderes" (usuarios) modificaron la misma base (el repositorio) de forma concurrente, lo que requiere una resolución manual del conflicto.

Arquitectura 3: Replicación Sin Líder (Leaderless)

- También conocida como modelo Dynamo (Amazon).
- Cualquier réplica puede aceptar escrituras directamente desde el cliente.
- **Escritura:** El cliente envía la escritura a 'W' nodos. Se considera exitosa si 'W' nodos responden.
- **Lectura:** El cliente solicita el dato a 'R' nodos.

Quorums para Consistencia:

- Para garantizar lecturas consistentes, se debe cumplir que **W + R mayor que N** (donde N es el número total de réplicas).
- Esto asegura que el conjunto de nodos de lectura y escritura siempre se solapen en al menos un nodo.

Manejo de Inconsistencias en Modelos Sin Líder

- A pesar de los quorums, pueden surgir conflictos.
- **Read Repair:** Durante una lectura, si el cliente detecta versiones diferentes en los nodos, escribe la versión más reciente en los nodos desactualizados.
- **Anti-Entropy:** Un proceso en segundo plano busca diferencias entre réplicas y las sincroniza.
- **Relojes Vectoriales (Vector Clocks):** Mecanismo para detectar conflictos de escritura concurrentes sin una autoridad central.

Particionamiento: ¿Qué es y por qué?

Definición

Dividir una base de datos muy grande en partes más pequeñas, llamadas particiones o *shards*, y distribuir estas particiones entre diferentes nodos.

Motivación Principal:

- **Escalabilidad de Datos:** Permite almacenar conjuntos de datos que superan la capacidad de un solo servidor.
- **Escalabilidad de Escritura:** Distribuye la carga de escritura entre múltiples nodos, mejorando el rendimiento.

Estrategias de Particionamiento

1. **Particionamiento por Rango de Clave (Key Range):**
   - Se asignan rangos contiguos de la clave de particionamiento a cada partición.
   - **Ejemplo:** Usuarios con IDs de 1-1000 en el nodo A, 1001-2000 en el nodo B, etc.
   - **Ventaja:** Búsquedas por rango son eficientes.

Estrategias de Particionamiento

1. **Particionamiento por Rango de Clave (Key Range):**
   - Se asignan rangos contiguos de la clave de particionamiento a cada partición.
   - **Ejemplo:** Usuarios con IDs de 1-1000 en el nodo A, 1001-2000 en el nodo B, etc.
   - **Ventaja:** Búsquedas por rango son eficientes.
   - **Desventaja:** Riesgo de "puntos calientes" (hot spots). Si las escrituras se concentran en un rango, un nodo se sobrecarga.

Estrategias de Particionamiento (Cont.)

2. Particionamiento por Hash de la Clave (Hash-based):

- Se aplica una función de hash a la clave de particionamiento. El resultado del hash determina en qué partición va el dato.
- **Ejemplo:** $nodo = \text{hash}(\text{user\_id}) \mod N_{\text{nodos}}$
- **Ventaja:** Distribuye la carga de manera muy uniforme, evitando *hot spots*.
- **Desventaja:** Las búsquedas por rango se vuelven ineficientes (hay que consultar todos los nodos).

Estrategias de Particionamiento

El Problema Fundamental

El particionamiento se optimiza para una **clave primaria** (ej: `user_id`). ¿Cómo realizamos búsquedas eficientes por otros campos índices

secundarios), como por ejemplo un `email`?

Estrategias de Particionamiento

Estrategias de Indexación

- **Índices Locales (por partición)**
  - Cada partición (shard) mantiene su propio índice local.
  - Para encontrar un email, hay que consultar el índice de **TODAS** las particiones. Esta operación (*scatter-gather*) es muy ineficiente.

- **Índices Globales (particionados)**
  - Se crea un índice global separado que mapea el campo secundario (ej: `email`) a la partición de datos correcta.
  - La búsqueda es un proceso de dos pasos: consultar el índice global y luego ir directamente al shard correcto. Es mucho más eficiente.

Rebalanceo de Particiones

- ¿Qué pasa cuando necesitamos añadir o quitar nodos del clúster?
- Necesitamos mover datos para redistribuir la carga. ¡Este es un proceso muy costoso y delicado!

Estrategias:

- **Número fijo de particiones:** Crear muchas más particiones que nodos y asignar varias a cada nodo. Mover particiones es más fácil que recalcular todo.
- **Particionamiento dinámico:** El sistema divide o fusiona particiones automáticamente según la carga.

Enrutamiento de Peticiones (Request Routing)

- Cuando un cliente quiere acceder a un dato, ¿cómo sabe a qué nodo conectarse?
- **Enrutador (Coordination Service):** Un componente (o conjunto de ellos) sabe qué partición vive en qué nodo.
- El cliente se conecta al enrutador, este le indica el nodo correcto, y el cliente se conecta a dicho nodo.
- Ejemplos: Zookeeper, etcd.

Recordando las Propiedades ACID

ACID

Un conjunto de propiedades que garantizan la fiabilidad de las transacciones en una base de datos.

- **Atomicidad:** Todo o nada. La transacción se completa enteramente o no tiene ningún efecto.
- **Consistencia:** La transacción lleva a la base de datos de un estado válido a otro.
- **Aislamiento (Isolation):** Las transacciones concurrentes no interfieren entre sí. Parecen ejecutarse en serie.
- **Durabilidad:** Una vez que una transacción es confirmada (commit), sus cambios persisten incluso ante fallos.

El Problema de las Transacciones Distribuidas

- Una sola transacción puede necesitar leer y escribir en múltiples nodos.
- **Reto 1 (Atomicidad):** ¿Cómo aseguramos "todo o nada"? Algunos nodos pueden confirmar la escritura mientras otros fallan.
- **Reto 2 (Aislamiento):** ¿Cómo manejamos los bloqueos y el control de concurrencia a través de la red? La latencia de red lo hace muy lento.

Protocolo de Compromiso en Dos Fases (2PC)

Two-Phase Commit (2PC)

Un algoritmo para garantizar la atomicidad en transacciones distribuidas.

- Hay un **Coordinador** y múltiples **Participantes** (los nodos de la BD).
- **Fase 1: Votación (Prepare Phase)**
  1. El Coordinador envía un mensaje 'prepare' a todos los participantes.
  2. Cada participante responde 'vote commit' (si puede garantizar la escritura) o 'vote abort'.
- **Fase 2: Decisión (Commit Phase)**
  1. Si **todos** votaron 'commit', el Coordinador envía 'global commit'.
  2. Si **alguno** votó 'abort' (o no respondió), envía 'global abort'.

Problemas del Protocolo 2PC

- Es un protocolo de **bloqueo**: los participantes deben esperar la decisión del coordinador.
- **¿Qué pasa si el Coordinador falla?**
  - Si falla **antes** de la Fase 2, los participantes quedan bloqueados y no saben si hacer commit o abort.
  - No pueden liberar los bloqueos hasta que el coordinador se recupere.
  - Esto puede detener una parte del sistema por un tiempo indefinido.
- Debido a esto, 2PC no se usa con frecuencia en sistemas que requieren altísima disponibilidad.

Control de Concurrencia Distribuido

- **Bloqueo Pesimista (Pessimistic Locking):**
  - Se asume que los conflictos son frecuentes.
  - Una transacción bloquea los datos que va a modificar para que otras no puedan acceder.
  - Ejemplo: Bloqueo en Dos Fases (2PL - No confundir con 2PC).
  - **Problema:** Muy lento en un entorno distribuido debido a la latencia de red para adquirir y liberar bloqueos.
- **Control Optimista (Optimistic Concurrency Control):**
  - Se asume que los conflictos son raros.
  - Las transacciones no bloquean. Al momento de hacer commit, el sistema verifica si hubo algún conflicto. Si lo hubo, la transacción se aborta y se reintenta.

El Teorema CAP (Teorema de Brewer)

Definición

En un sistema de almacenamiento de datos distribuido, es imposible garantizar simultáneamente más de dos de las siguientes tres propiedades:

- **C**onsistency (Consistencia Fuerte): Cada lectura recibe la escritura más reciente o un error.
- **A**vailability (Disponibilidad): Cada petición recibe una respuesta (no un error), sin garantizar que contenga la escritura más reciente.
- **P**artition Tolerance (Tolerancia a Particiones): El sistema continúa operando a pesar de que un número arbitrario de mensajes se pierdan (o retrasen) por la red entre nodos.

CAP en la Práctica

- En un sistema distribuido, la **tolerancia a particiones (P) no es opcional**. Las fallas de red ocurren.
- Por lo tanto, el verdadero dilema es entre **Consistencia** y **Disponibilidad**.
- **Sistema CP (Consistencia ¿ Disponibilidad):**
  - Ante una partición de red, el sistema elige dejar de responder (no estar disponible) para evitar devolver datos inconsistentes.
  - Ejemplo: Sistemas bancarios.
- **Sistema AP (Disponibilidad ¿ Consistencia):**
  - Ante una partición de red, el sistema sigue respondiendo, aunque sea con datos que podrían estar desactualizados.
  - Ejemplo: Redes sociales (es mejor mostrar un feed un poco antiguo a no mostrar nada).

Más allá de Fuerte vs. Débil: Consistencia Eventual

Consistencia Eventual (Eventual Consistency)

Si no se realizan nuevas actualizaciones a un dato, eventualmente todas las réplicas convergerán al mismo valor.

- Es el modelo de los sistemas AP.
- Acepta que habrá un período de inconsistencia, pero garantiza que el sistema se "curará" a sí mismo con el tiempo.
- Es un compromiso práctico que permite alta disponibilidad y escalabilidad.

Modelos de Consistencia más Débiles

Existen muchos modelos intermedios. Algunos importantes:

- **Read-your-writes Consistency:** Un usuario siempre ve sus propias actualizaciones.
- **Monotonic Reads:** Un usuario nunca ve que el tiempo "retrocede". Si lee un valor, en lecturas posteriores nunca verá un valor anterior.
- **Causal Consistency:** Si la operación A "causa" la operación B (p.ej., una respuesta a un comentario), todos los nodos deben ver A antes de ver B.

El Ecosistema Moderno: SQL y NoSQL

Bases de Datos Relacionales (SQL) Distribuidas:

- Han adoptado técnicas de replicación y particionamiento.
- Generalmente favorecen la consistencia (CP).
- Ejemplos: Google Spanner, CockroachDB, Vitess.

Bases de Datos NoSQL:

- Nacieron para la escala web. Generalmente favorecen la disponibilidad (AP).
- Key-Value (DynamoDB, Riak), Document (MongoDB), Column-Family (Cassandra), Graph (Neo4j).

NewSQL: Lo mejor de ambos mundos?

NewSQL

Una nueva generación de bases de datos que busca combinar la escalabilidad horizontal de NoSQL con las garantías transaccionales ACID de los sistemas SQL tradicionales.

- Ofrecen interfaces SQL.
- Diseñadas desde cero para ser distribuidas.
- Utilizan algoritmos de consenso como Paxos o Raft en lugar de 2PC para evitar sus problemas de bloqueo.

Algoritmos de Consenso: Paxos y Raft

- Son protocolos para lograr que un grupo de nodos se ponga de acuerdo sobre un valor o estado.
- Son la base de muchos sistemas distribuidos modernos para tareas como la elección de líder o la replicación de logs de estado.
- **Raft** fue diseñado para ser más fácil de entender que Paxos, y es muy popular hoy en día (usado en etcd, CockroachDB).

Resumen Final de Conceptos

- No existe una solución única. La elección del sistema depende del caso de uso.
- **Replicación** nos da disponibilidad y escalabilidad de lectura.
- **Particionamiento** nos da escalabilidad de datos y escritura.
- **Transacciones** en sistemas distribuidos son un compromiso entre consistencia y rendimiento/disponibilidad.
- El **Teorema CAP** nos fuerza a elegir entre Consistencia y Disponibilidad ante fallos de red.

¡No hay magia!

Recordatorio

Todo en sistemas distribuidos es un *trade-off*. Entender estos compromisos es clave para diseñar sistemas robustos y escalables.

- Rendimiento vs. Consistencia
- Latencia vs. Durabilidad
- Simplicidad vs. Flexibilidad

¿Preguntas?

Procesamiento de Grandes Volumenes de Datos

Conferencia 2: Fundamentos del procesamiento distribuido

Deborah Famadas Rodríguez

Universidad de la Habana

September 8, 2025