Procesamiento de Grandes Volumenes de Datos

Conferencia 3: Fundamentos del procesamiento distribuido Part-1

Deborah Famadas Rodríguez

Universidad de la Habana

September 15, 2025

Estrategias de Particionamiento

Existen varias maneras de decidir **dónde** va cada dato.

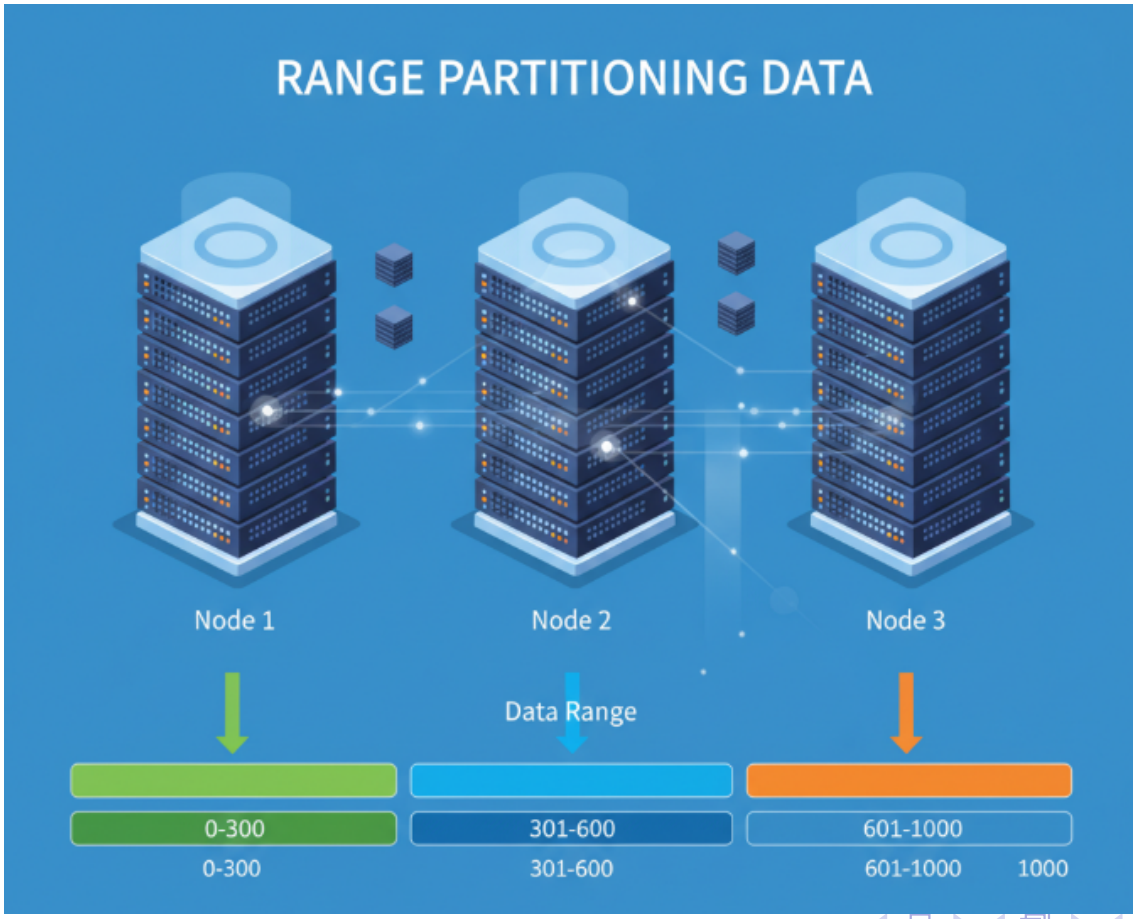
- Por Rango de Clave
- Por Hash de la Clave
- Por Directorio
- Geográfico

La elección correcta depende del patrón de acceso a los datos.

Estrategia 1: Particionamiento por Rango

Idea Principal: Se asignan rangos contiguos de la clave de particionamiento a cada nodo.

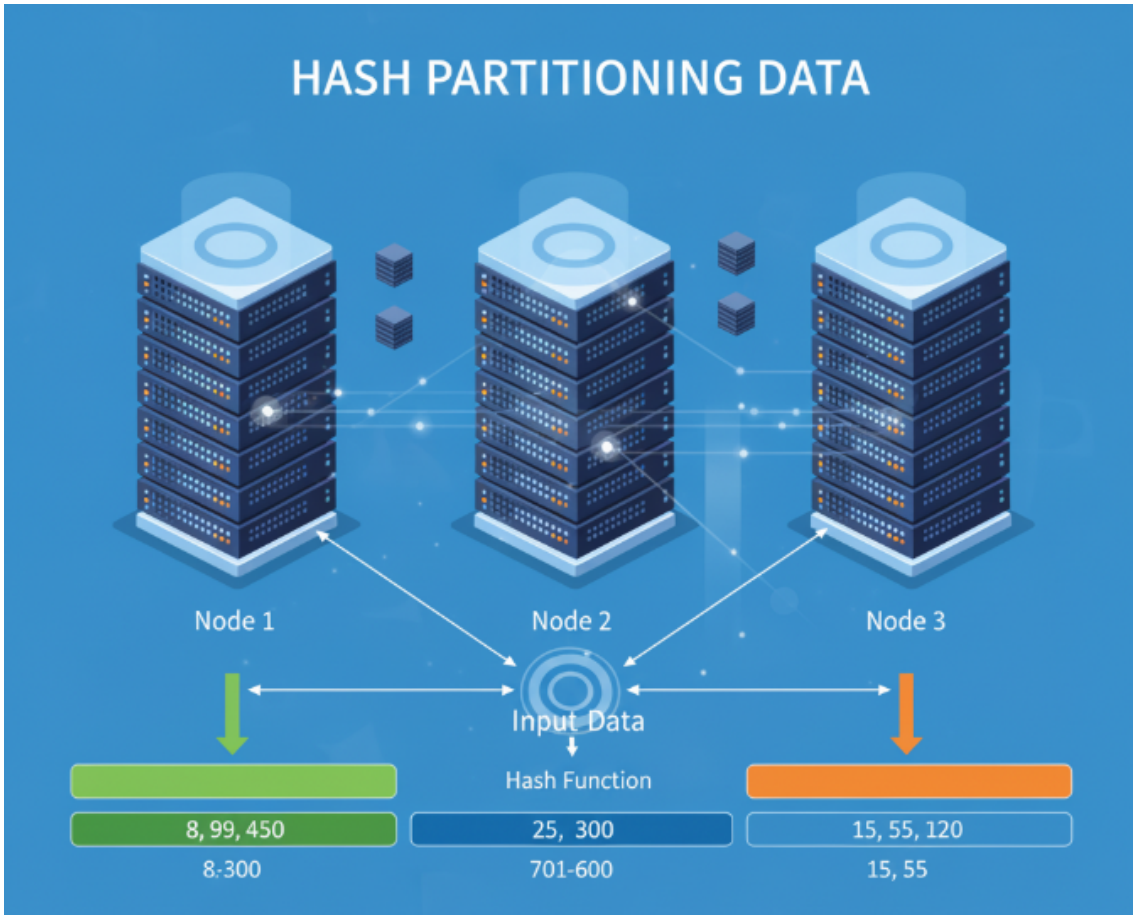
- **Ejemplo:** Usuarios con IDs de 1-1000 en el nodo A, 1001-2000 en el nodo B, etc.



Estrategia 2: Particionamiento por Hash

Idea Principal: Se aplica una función de hash a la clave. El resultado determina el nodo de destino.

- **Ejemplo:** $nodo = \text{hash}(\text{user\_id}) mod N_{\text{nodos}}$



Estrategia 3: Particionamiento por Directorio

Idea Principal: Una tabla de búsqueda (lookup table) centralizada mapea cada clave a su nodo correspondiente.

- **¿Cómo funciona?** Antes de acceder a un dato, se consulta al directorio para saber a qué nodo ir.

  Ventaja: Máxima flexibilidad. Mover datos entre nodos solo requiere actualizar el directorio, lo que simplifica el rebalanceo de carga.

  Desventajas:

  - El directorio es un **cuello de botella** y un **punto único de fallo**.
  - Añade una latencia extra a cada operación (la consulta al directorio).

Estrategia 4: Particionamiento Geográfico (Geo-Partitioning)

Idea Principal: Los datos se almacenan en centros de datos físicamente cercanos a los usuarios.

- **Ejemplo:** Los datos de usuarios europeos se guardan en un servidor en Frankfurt, los de EE.UU. en Virginia.

El Desafío: Índices Secundarios

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`?

Buscar por email requiere una estrategia, porque el hash(email) no nos dirá dónde está el usuario particionado por user_id.

Solución 1: Índice Local por Partición

La Estrategia de "Preguntar a Todos"

Cada partición (nodo) gestiona su propio "mini-índice" solo para los datos que almacena.

Uso y Relevancia Actual

¿Cuándo se usa? Principalmente en sistemas donde las **escrituras rápidas son la máxima prioridad**. La escritura de un dato y su índice ocurre en un solo lugar, lo que es muy eficiente.

- **Trade-Off:** Se sacrifica la flexibilidad de las lecturas. Para que la consulta sea eficiente, se obliga a incluir la clave de partición. (Ej: 'WHERE ciudad='Madrid' AND email='...'').

Solución 2: Índice Global Particionado

La Estrategia del "Catálogo Central"

Se crea un índice totalmente separado, que funciona como una tabla distribuida independiente. Este "catálogo" está particionado por el campo secundario (ej: `email`) y su único trabajo es mapear ese campo a la clave principal (`user_id`).

Uso y Relevancia Actual

¿Cuándo se usa? Es la **estrategia dominante hoy en día** para la mayoría de las bases de datos que necesitan ofrecer consultas flexibles y rápidas por cualquier atributo.

- **Trade-Off:** Las escrituras son más lentas. Cada vez que se inserta o actualiza un dato, se deben actualizar dos tablas distribuidas (la de datos y la del índice), lo que puede requerir transacciones más complejas.

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.

El Problema del Enrutamiento

La Pregunta Fundamental

En un clúster con N nodos, un cliente tiene una clave de partición (ej: `user_id=123`). **¿Cómo descubre la dirección IP del nodo exacto que almacena ese dato?**

La Solución: La Tabla de Enrutamiento

- En algún lugar del sistema, debe existir un **mapa** que asocia cada partición con el nodo físico que la aloja.
- *Partición 1-100 → Nodo A (10.0.1.5)*
- *Partición 101-200 → Nodo B (10.0.1.6)*
- Este mapa es dinámico y debe estar siempre actualizado.

El Desafío

¿Quién consulta este mapa y cómo se dirige la petición al nodo correcto? Existen varias arquitecturas para resolver esto.

Enfoque 1: Gateway de Enrutamiento (Proxy)

Idea: El cliente no sabe nada sobre la topología del clúster. Envía todas sus peticiones a un único punto de entrada: un gateway o proxy.

- El **gateway** contiene la lógica de enrutamiento.
- Consulta el mapa de particiones (que puede obtener de Zookeeper/etcd).
- Reenvía la petición del cliente al nodo de datos correcto.
- Espera la respuesta del nodo y se la devuelve al cliente.

$$\texttt{Cliente} \rightarrow \texttt{Gateway} \rightarrow \texttt{Nodo de Datos}$$

Enfoque 2: Enrutamiento en el Cliente (Smart Client)

Idea: El cliente es "inteligente". Descarga y cachea el mapa de particiones para saber exactamente a qué nodo conectarse.

- Al iniciar, el cliente se conecta a un nodo cualquiera del clúster para obtener la tabla de enrutamiento completa.
- Para cada petición, el cliente calcula el hash de la clave, consulta su mapa local y envía la petición **directamente** al nodo correcto.
- El cliente se suscribe a actualizaciones para mantener su mapa fresco.

$$\texttt{Cliente} \xrightarrow{\texttt{Consulta su mapa local}} \texttt{Nodo de Datos}$$

Ejemplos: Clientes de Cassandra, Kafka.

Enfoque 3: Coordinador + Redirección

Idea: Un nodo cualquiera puede recibir la petición, pero no la reenvía él mismo. En su lugar, actúa como coordinador y le indica al cliente a dónde ir. `Cliente` $\rightarrow$ `Nodo Aleatorio` $\rightarrow$ `Cliente (Respuesta: "Ve al Nodo B")` $\rightarrow$ `Nodo B (Correcto)`

Ejemplos: Redis Cluster sigue este modelo.

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.

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.

El Problema del Consenso

- En un sistema distribuido, un conjunto de nodos necesita ponerse de acuerdo sobre un valor o el estado del sistema.
- **Retos:** los nodos pueden fallar, los mensajes pueden perderse o llegar con retraso.
- **Objetivo:** asegurar que, una vez que se toma una decisión, esta sea final y todos los nodos la conozcan.
- Paxos es un protocolo diseñado para resolver este problema de manera tolerante a fallos.

Roles en Paxos

En Paxos, los nodos pueden desempeñar tres roles distintos:

- **Proponentes (Proposers):** Proponen valores. Son los iniciadores del consenso.
- **Aceptadores (Acceptors):** Aceptan o rechazan los valores propuestos. Forman la "memoria" del sistema y son la clave para la toma de decisiones. Un quórum (mayoría) de aceptadores es necesario para llegar a un acuerdo.
- **Aprendices (Learners):** Una vez que un valor ha sido aceptado por un quórum, los aprendices se enteran del valor decidido.

El Proceso de Paxos: Dos Fases

Paxos funciona en dos fases principales para garantizar que no se tomen decisiones conflictivas.

Fase 1: Preparación (Prepare)

1. Un **Proponente** elige un número de propuesta N (único y mayor que cualquiera usado antes por él).
2. Envía un mensaje `prepare(N)` a una mayoría de **Aceptadores**.
3. Un **Aceptador** responde con una "promesa" (`promise`) de no aceptar ninguna propuesta con un número menor que N. Si ya había aceptado un valor, informa al proponente de cuál era y con qué número de propuesta.

Fase 2: Aceptación (Accept)

1. Si el **Proponente** recibe promesas de una mayoría de Aceptadores:
   - Elige un valor V. Si los aceptadores ya habían aceptado algún valor, debe proponer el valor asociado con el número de propuesta más alto que recibió. Si no, puede proponer su propio valor.
   - Envía un mensaje `accept(N, V)` a esos Aceptadores.
2. Un **Aceptador** recibe el mensaje `accept(N, V)`:
   - Si no ha prometido atender a una propuesta mayor que N, acepta el valor V.
   - Notifica a los **Aprendices** que el valor V ha sido elegido.

¡Consenso alcanzado! Cuando una mayoría de aceptadores ha aceptado el mismo valor, ese valor es el elegido por el sistema.

Conclusiones sobre Paxos

- **Fortalezas:**
  - Garantiza la seguridad (nunca se elegirá más de un valor).
  - Es tolerante a fallos de nodos y de red (mientras una mayoría de aceptadores esté activa).

- **Debilidades:**
  - Es conocido por ser difícil de entender e implementar correctamente.
  - El protocolo básico es para decidir un único valor. Para una secuencia de valores (como un log), se necesita una extensión llamada Multi-Paxos.
  - Puede haber "duelos" de proponentes que impidan el progreso si no se gestiona bien la elección de números de propuesta.

¿Por qué Raft?

- Paxos ha sido el estándar teórico durante años, pero es muy difícil de implementar correctamente.
- **Raft** fue creado con un objetivo principal: **ser más fácil de entender que Paxos**.
- Para lograrlo, descompone el problema del consenso en tres partes más manejables.
- Hoy en día es muy popular y se usa en sistemas como **etcd** (el cerebro de Kubernetes) y **CockroachDB**.

Los Tres Subproblemas de Raft

Raft aborda el consenso dividiéndolo en tres partes clave:

1. Elección de Líder

Un único nodo es elegido como **líder** para gestionar el clúster. Si falla, se elige uno nuevo.

2. Replicación de Logs

El líder acepta comandos de los clientes, los añade a su log y se asegura de que los otros nodos (seguidores) repliquen ese log de forma idéntica.

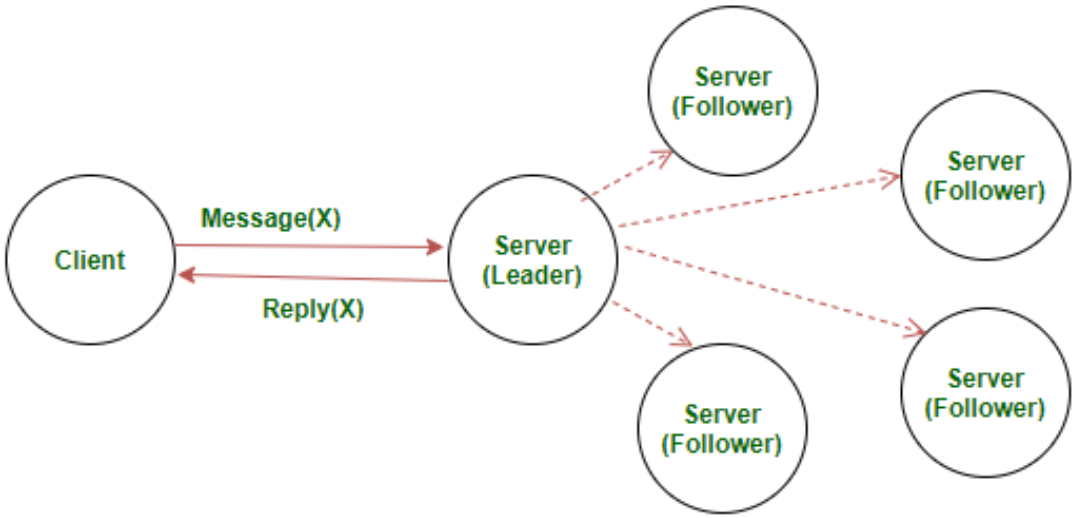
3. Seguridad

Un conjunto de reglas que garantizan la consistencia del sistema. Por ejemplo, si un nodo ha aplicado una entrada de log, ningún otro nodo puede aplicar una entrada diferente en ese mismo índice.

Estados de los Nodos y Elección de Líder

En Raft, un nodo siempre está en uno de estos tres estados:

- **Líder (Leader):** Gestiona todas las peticiones de los clientes y la replicación del log. Solo puede haber un líder a la vez. Envía "heartbeats" (latidos) a los seguidores para mantener su autoridad.
- **Seguidor (Follower):** Es pasivo. Responde a las peticiones del líder y de los candidatos. Si no recibe un "heartbeat" del líder durante un tiempo (election timeout), se convierte en candidato.
- **Candidato (Candidate):** Inicia una elección para convertirse en el nuevo líder. Pide votos a los demás nodos.



El Proceso de Elección

1. Un seguidor agota su *election timeout* y se convierte en candidato. Incrementa su "término" (un contador de épocas).
2. El candidato se vota a sí mismo y envía peticiones de voto (`RequestVote`) a todos los demás nodos.
3. Los otros nodos le conceden su voto si no han votado ya en ese término.
4. Si el candidato recibe votos de una **mayoría** del clúster, se convierte en el nuevo **Líder**.
5. Si otro candidato gana la elección o aparece un líder legítimo, vuelve a ser seguidor.
6. Si la votación se divide y nadie gana, se agota el tiempo y comienza una nueva elección con un nuevo término.

Replicación de Logs

Una vez hay un líder, el sistema puede procesar peticiones:

- **1. Petición del cliente:** Un cliente envía un comando al líder.
- **2. Añadir al log:** El líder añade el comando a su propio log como una nueva entrada.
- **3. Replicación:** El líder envía la entrada a todos los seguidores mediante mensajes `AppendEntries`.
- **4. Confirmación (Commit):** Cuando una **mayoría** de seguidores ha guardado la entrada en su log, el líder considera la entrada como "comprometida" (committed).
- **5. Aplicación y respuesta:** El líder aplica el comando a su máquina de estados y devuelve el resultado al cliente. Los seguidores aplicarán la entrada cuando sean notificados por el líder.

Raft garantiza que los logs de todos los nodos se mantengan consistentes.

Conclusiones sobre Raft

- **Comprensibilidad:** Su principal ventaja. Descompone el problema de forma clara, facilitando su implementación y depuración.
- **Elección de líder fuerte:** El rol del líder simplifica enormemente la gestión del sistema. Todas las decisiones pasan por él.
- **Seguridad demostrada:** Raft tiene una prueba formal de su correctitud.
- Es la base de muchos sistemas distribuidos modernos y fiables, demostrando que un diseño más simple puede ser igual de potente y más práctico que alternativas más complejas como Paxos.

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 3: Fundamentos del procesamiento distribuido Part-1

Deborah Famadas Rodríguez

Universidad de la Habana

September 15, 2025