

UNIVERSIDAD NACIONAL DE INGENIERÍA

Arquitectura de Software

Replicación

Un líder: síncrona, asíncrona, retraso y failover · Varios líderes y sin líder: conflictos y quórums

Gerson Garrido

Semana 10

Agenda

Sesión A

1. Para qué se replica
2. Replicación con un líder
3. Síncrona y asíncrona
4. El retraso y sus anomalías
5. Failover

Sesión B

6. Varios líderes: conflictos
7. Sin líder: quórums
8. Dynamo y cómo se elige

Lectura

A: Kleppmann y Riccomini, *Designing Data-Intensive Applications* (2.ª ed.), cap. 6 *Replication*: la introducción, *Single-Leader Replication* y los problemas del retraso de replicación; Kleppmann, apuntes de sistemas distribuidos de Cambridge, sección 5.
B: cap. 6: *Multi-Leader Replication* y *Leaderless Replication*; DeCandia y otros, *Dynamo: Amazon's Highly Available Key-value Store* (2007), secciones 1, 2 y 4.

SECCIÓN 01

Para qué se replica

01

Tres razones, tres escenarios

Replicar: mantener los mismos datos en varias máquinas.

Razón	Atributo	Qué le pide a la copia
Seguir si una máquina falla	Disponibilidad	Estar al día en el momento de la falla
Atender más lecturas	Rendimiento	Muchas copias rápidas; tolera algo de atraso
Datos cerca de quien los usa	Latencia	Copias lejos entre sí, con una red lenta en medio

La primera pregunta

Ante «pongamos una réplica»: ¿para cuál de las tres? Un diseño que sirve para una puede ser malo para otra.

Toda la dificultad está en propagar los cambios

- Si los datos no cambiaran, replicar sería copiar una vez.
- Cada escritura viaja por una red que demora, pierde y se corta.
- Mientras viaja, hay copias que la tienen y copias que no, y alguien está leyendo.

Tres preguntas

1. ¿Quién acepta escrituras?
2. ¿Cuándo se responde «listo»?
3. ¿Qué ve quien lee de una copia atrasada?

Idea clave

Replicar no agrega una garantía: cambia unas por otras. Se paga con latencia al escribir, con disponibilidad cuando un nodo falla o con lecturas que muestran el pasado.

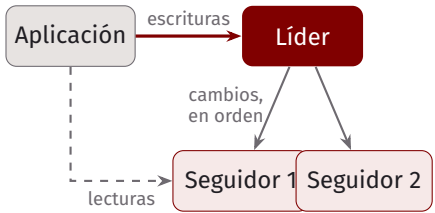
SECCIÓN 02

Replicación con un líder

02



La forma



- Un nodo, el **líder**, acepta las escrituras.
- Los **seguidores** las aplican en el mismo orden.
- Las lecturas van a cualquiera.

- **Un solo orden de las escrituras**: transacciones y restricciones como en una máquina.
- **Los seguidores van detrás**: tienen el estado del líder en algún momento del pasado.
- No da capacidad de escritura: todas siguen pasando por un nodo.

Cómo viaja el cambio: el WAL

- PostgreSQL ya escribe cada cambio en su **WAL** *(write-ahead log)* antes de tocar los datos.
- Un seguidor es un servidor que está permanentemente recuperándose: aplica el WAL de otro.
- Cada posición del WAL tiene un número, el **LSN**: `0/2E009C78`. Solo crece.
- **Replicación por streaming**: el WAL llega en continuo por una conexión abierta.

Copia física

Lo que viaja son cambios a páginas de disco. El seguidor tiene las mismas tablas e índices, byte por byte, y **solo acepta lecturas**.

Una réplica en el contenedor: solo lecturas

pg-replica create: el principal en el puerto 5432, la réplica en el 5433

```
$ psql -c "SELECT pg_is_in_recovery()"
 pg_is_in_recovery
-------------------
 f
(1 row)

$ psql -p 5433 -c "SELECT pg_is_in_recovery()"
 pg_is_in_recovery
-------------------
 t
(1 row)

$ psql -p 5433 -d arq_w10 \
    -c "INSERT INTO orders (buyer, seat) VALUES ('rosa.quispe@correo.pe', 'A-1')"
ERROR:  cannot execute INSERT in a read-only transaction
```

Agregar un seguidor sin detener al líder

1. Una copia consistente del líder (un **snapshot**), anotando su posición en el log.
2. Llevarla al nodo nuevo.
3. El nodo nuevo pide **todo el log desde esa posición** y sigue recibiendo.

env/pg-replica

```
1  gosu postgres pg_basebackup -D "$DIR" -R -X stream -C -S replica_1 -c fast
```

- `-R`: deja `standby.signal` y `primary_conninfo`. `-C -S`: crea un **slot** en el principal.

> **Trade-off**
>
> El slot impide que el líder borre WAL que el seguidor no recibió. Lo paga el disco del líder: un seguidor que desaparece sin que nadie borre su slot termina llenándolo.

El viaje de un cambio, en cuatro posiciones



- `pg_stat_replication`, en el principal: una fila por seguidor, con las cuatro posiciones.
- Hasta `flush_lsn`: sobrevive aunque el principal muera. Hasta `replay_lsn`: una lectura ya lo ve.
- `pg-replica lag`: los bytes que faltan aplicar y `replay_lag`.

Error común

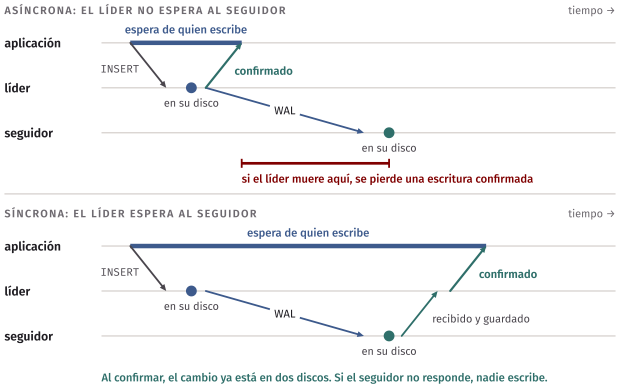
Medir el retraso solo con `replay_lag`: si la réplica deja de recibir, se queda mostrando el último valor. Se miran los bytes pendientes.

SECCIÓN 03

Síncrona y asíncrona

03

Qué espera el líder antes de responder



La pregunta que las separa: ¿qué pasa si el líder muere un instante después de responder «confirmado»?

Lo que garantiza cada una

	Asíncrona	**Síncrona**
El líder responde cuando	el cambio está en su disco	además está en el disco del seguidor
Si el líder muere después de confirmar	una escritura confirmada se puede perder	nada confirmado se pierde
Latencia de cada escritura	la del líder	más la ida y vuelta al seguidor
Si el seguidor no responde	el líder sigue	las escrituras se detienen

En PostgreSQL: `synchronous_standby_names = 'walreceiver'` en el principal (`make sync-on`).

Lo que cuesta la síncrona, medido

```
CLIENTS=1 WRITES=1000 make burst

réplica: async
1000 escrituras con 1 conexión en 1730,27 ms
latencia de escritura: p50 1,16 ms, p95 5,00 ms, p99 8,94 ms
...
réplica: sync
1000 escrituras con 1 conexión en 2950,87 ms
latencia de escritura: p50 2,30 ms, p95 6,72 ms, p99 11,45 ms
...
```

- Tres pares de corridas, alternando: p50 de 1,16 / 0,90 / 1,30 ms asíncrona y 2,30 / 1,26 / 1,85 ms síncrona.
- La síncrona tardó entre 1,4 y 2 veces en la mediana. Laptop con Apple Silicon, en Docker; varía entre corridas.
- Es la diferencia mínima: aquí no hay red. Entre ciudades, cada escritura paga decenas de milisegundos.

Cuando el seguidor síncrono no responde

`pg-replica stop`, y un `INSERT` en el principal: no da error, **espera**. Al cancelarlo desde otra sesión:

el INSERT, cancelado con pg_cancel_backend

```
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been replicated to the standby.
INSERT 0 1
```

- La fila está en el principal: cancelar la espera no deshace la escritura.

Error común

Síncrona con un solo seguidor no es «alta disponibilidad»: es durabilidad a cambio de *menos* disponibilidad de escritura ($0{,}999 \cdot 0{,}999 \approx 0{,}998$).

Semisíncrona, y el trade-off

- Varios seguidores, y el líder espera **a uno cualquiera**: `'ANY 1 (replica_a, replica_b)'`.
- Si un seguidor se cae, otro ocupa su lugar. Lo confirmado está siempre en dos máquinas.

	Se gana	**Se paga**
Asíncrona	Escrituras rápidas; el líder no depende de nadie	Un failover puede perder lo confirmado al final
Síncrona, un seguidor	Nada confirmado se pierde	Latencia; el seguidor caído detiene las escrituras
Semisíncrona	Las dos garantías	Otra máquina

Una red social puede perder un «me gusta». Un sistema que cobra no puede perder un pago.

SECCIÓN 04

El retraso y sus anomalías

04

El retraso de replicación: leer de un seguidor es leer el pasado

make delay MS=2000: la réplica aplica 2 s tarde

```
$ psql -d arq_w10 -c "INSERT INTO orders (buyer, seat) VALUES ('rosa.quispe@correo.pe', 'B-7')"
INSERT 0 1
$ psql -p 5433 -d arq_w10 -c "SELECT count(*) FROM orders WHERE seat = 'B-7'"
 count
-------
     0
(1 row)
```

Consistencia eventual

Si se deja de escribir, las copias convergen. No dice **cuándo** ni **qué se ve mientras tanto**.

No leer lo que uno escribió

read-your-writes.ts

```ts
const id = await createOrder(primary, `comprador-${run}@correo.pe`, `A-${run}`);
const confirmed = process.hrtime.bigint();
if (await orderExists(replica, id)) continue;

missing++;
```

make anomaly, sin ninguna demora artificial

```
100 órdenes creadas en el principal y leídas enseguida en la réplica
no encontradas en la primera lectura: 80 de 100
las que faltaban aparecieron después de: p50 2,52 ms, máximo 10,03 ms
```

- Error común: enviar todas las lecturas a la réplica «para descargar al principal».

Read-your-writes, de tres maneras

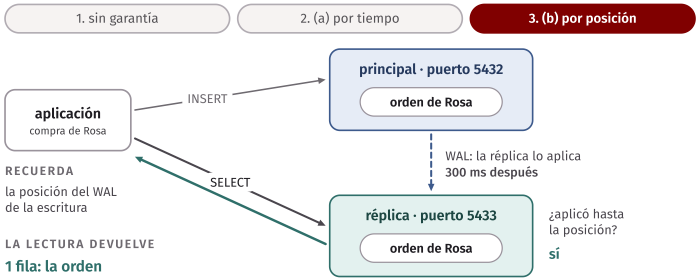
Read-your-writes

Después de escribir, el mismo usuario ve su escritura, o algo posterior. No promete nada sobre los demás.

- **Por tipo de dato**: leer del líder lo que el propio usuario pudo modificar (su perfil).
- **(a) Por tiempo**: leer del líder durante unos segundos después de una escritura del mismo usuario.
- **(b) Por posición**: guardar el LSN de la escritura y esperar a que el seguidor lo haya aplicado.

```
SELECT pg_last_wal_replay_lsn() >= $1::pg_lsn AS reached;
```

Sin garantía, (a) y (b), paso a paso



Mide el retraso en vez de suponerlo, y lo paga quien lee: unos 300 ms.

MAKE COMPARE: 100 COMPRAS, DEMORA DE 300 MS

estrategia	no encontradas	lecturas en el principal	lectura p50
ninguna	100	0	0,43 ms
(a) principal un rato	0	100	0,32 ms
(b) esperar a la réplica	0	0	301,63 ms

(a) y (b), medidas

make compare, con una demora de 300 ms en la réplica

```
100 compras por estrategia; ventana de (a): 5000 ms; pausa antes de leer: 0 ms
ninguna                 no encontradas: 100   lecturas en el principal:   0   lectura p50 0,43 ms, p95 1,51 ms
(a) principal un rato   no encontradas:   0   lecturas en el principal: 100   lectura p50 0,32 ms, p95 0,99 ms
(b) esperar a la réplica  no encontradas:   0   lecturas en el principal:   0   lectura p50 301,63 ms, p95 306,52 ms
```

THINK_MS=150 WINDOW_MS=100 RUNS=30 make compare

```
30 compras por estrategia; ventana de (a): 100 ms; pausa antes de leer: 150 ms
ninguna                 no encontradas:  30   lecturas en el principal:   0   lectura p50 0,97 ms, p95 3,36 ms
(a) principal un rato   no encontradas:  30   lecturas en el principal:   0   lectura p50 0,79 ms, p95 4,52 ms
(b) esperar a la réplica  no encontradas:   0   lecturas en el principal:   0   lectura p50 152,35 ms, p95 157,66 ms
```

- (a) supone que el retraso es menor que la ventana. (b) lo mide: la lectura espera, y necesita un timeout.

Un error que pasa las pruebas: el LSN antes del commit

La posición se pide **después** de confirmar la escritura, en una consulta aparte.

make wrong-lsn, con una demora de 2 s

```
posición tomada dentro del INSERT (0/3009C48): la réplica llegó ahí;
  ¿ve la orden? no
posición tomada después del commit (0/3009D38): la réplica llegó ahí;
  ¿ve la orden? sí
```

- `INSERT ... RETURNING pg_current_wal_insert_lsn()` da una posición anterior al registro de COMMIT.
- Una fila es visible en la réplica cuando se aplica el commit de su transacción.
- Con una réplica al día casi nunca se nota: el commit llega una fracción de milisegundo después.

Síncrona no es read-your-writes

La síncrona espera a que la réplica **guarde** el WAL (`flush_lsn`), no a que lo **aplique** (`replay_lsn`).

```
make sync-on; make anomaly

100 órdenes creadas en el principal y leídas enseguida en la réplica
no encontradas en la primera lectura: 2 de 100
las que faltaban aparecieron después de: p50 2,92 ms, máximo 3,21 ms
```

- Baja (de 80 a 2 o 3), pero nada lo garantiza: con una demora de aplicación vuelven a ser todas.
- `synchronous_commit = remote_apply`: 0 de 100. Ahora espera toda escritura, lean después o no.
- Síncrona responde «¿se pierde lo confirmado?». Read-your-writes, «¿ve su escritura quien escribió?».

Lecturas que retroceden

```
make monotonic, con la demora de 2 s

lectura 1 (principal): 1 orden(es) del asiento M-1794811781632
lectura 2 (réplica): 0 orden(es) del asiento M-1794811781632   <- retrocedió
lectura 3 (principal): 1 orden(es) del asiento M-1794811781632
```

Lecturas monótonas

Un mismo usuario nunca ve un estado anterior al que ya vio. Se da **fijando cada usuario a un seguidor**.

- **Prefijo consistente**: nadie ve un efecto antes que su causa (la respuesta antes que la pregunta). Con un líder y un solo log viene incluida; se rompe con varios logs: shards o varios líderes.

Qué lectura va a dónde

Lectura	Si ve datos de hace segundos	Va a
Lista de conciertos	No pasa nada	Réplica
Asientos libres, para el mapa	Ve libre uno recién vendido; la compra lo rechazará	Réplica
¿Está libre este asiento?, al comprarlo	Vendería un asiento dos veces	Líder, en la transacción
La orden que acabo de pagar	Cree que no compró	Con read-your-writes
Reporte del mes	No pasa nada	Réplica

En Tinkuy (P3)

El catálogo y el reporte pueden ir a una réplica; la confirmación de una compra, no sin más. La decisión es **por caso de uso**, y se escribe en un ADR.

Una réplica no es un respaldo

un DELETE sin WHERE, en el principal

```
$ psql -d arq_w10 -c "DELETE FROM orders"
DELETE 1
$ psql -p 5433 -d arq_w10 -c "SELECT count(*) FROM orders"
 count
-------
     0
(1 row)
```

- La réplica hizo su trabajo: parecerse al líder lo antes posible, errores incluidos.
- Una réplica protege de la **falla de una máquina**. Un respaldo, de **cambios que no debieron ocurrir**.
- Para lo segundo hay que volver a un momento anterior, y una réplica solo tiene el presente.

SECCIÓN 05

Failover

05



Tres pasos, un riesgo en cada uno

Hacer failover: promover a un seguidor cuando el líder falla.

1. **Detectar la falla.** Solo se observa que no responde: ¿murió, está lento o se cortó la red? Se decide con un timeout.
   - Corto: failovers innecesarios, justo bajo carga. Largo: alarga cada caída real.
2. **Elegir al nuevo líder.** El más actualizado, y todos de acuerdo en cuál: eso es consenso (semana 13).
3. **Redirigir a los clientes.** Y que el anterior, si vuelve, se entere de que ya no es líder.

Dos riesgos

Con replicación asíncrona, el nuevo líder puede no tener escrituras **ya confirmadas**. Si el anterior vuelve creyéndose líder, hay **dos nodos aceptando escrituras**.

Un failover a mano

make writer; en otra terminal, make kill-primary

```
Escribiendo órdenes numeradas en el principal (puerto 5432)...
...
El principal dejó de responder: read ECONNRESET
Última escritura confirmada al cliente: 6005
Escritura sin respuesta (no se sabe si llegó): 6006
```

promover y comparar

```
$ pg-replica promote
La réplica ahora es un principal independiente en el puerto 5433.
$ make compare-failover
confirmadas al cliente:        1..6005
existen en el nuevo principal: 1..6005 (6005 filas)
ninguna confirmada se perdió
```

Esta vez. Con replicación asíncrona, eso es suerte.

Escrituras confirmadas que se pierden

	Asíncrona, enlace sano	Asíncrona, enlace cortado 1 s	Síncrona, enlace cortado 1 s
Confirmadas que se perdieron	0, 1 o 3 (diez corridas)	830, 692 y 1145 (tres)	0
¿Escribía durante el corte?	—	Sí	No

```
make link-cut, un segundo, make kill-primary, pg-replica promote, make compare-failover

confirmadas al cliente:        1..2669
existen en el nuevo principal: 1..1839 (1839 filas)
confirmadas que se PERDIERON:  830 (de la 1840 a la 2669)
```

- Ningún error en ningún log. Enlace sano: ocho de diez corridas sin pérdida, una perdió 1 y otra 3.
- La última fila es el precio de la penúltima: el síncrono no perdió ventas porque no vendió.

Síncrona: lo confirmado existe; lo no confirmado, quién sabe

el mismo failover, con make sync-on

```
...
Última escritura confirmada al cliente: 654
Escritura sin respuesta (no se sabe si llegó): 655
$ make compare-failover
confirmadas al cliente:        1..654
existen en el nuevo principal: 1..655 (655 filas)
ninguna confirmada se perdió; además llegó la 655, que el cliente nunca supo si se guardó
```

- La orden 655 salió hacia la réplica; lo que no llegó fue la confirmación al cliente.
- Es la respuesta perdida de la semana 5. Un cliente que reintenta la escribiría dos veces, salvo que la escritura sea idempotente (semana 12).

Split brain

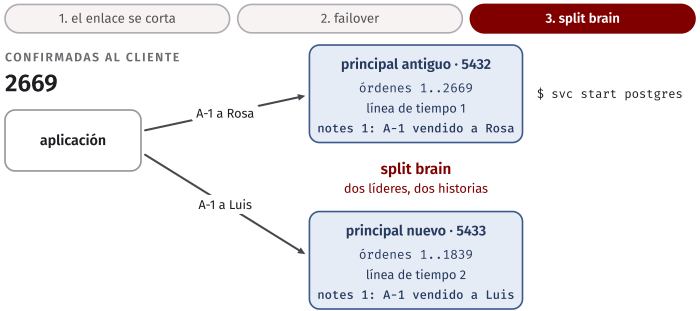
vuelve el principal antiguo (svc start postgres): nadie le avisó del failover

```
$ psql -p 5432 -d arq_w10 -c "SELECT id, written_on, body FROM notes"
 id | written_on |            body
----+------------+----------------------------
  1 | 5432       | asiento A-1 vendido a Rosa
(1 row)

$ psql -p 5433 -d arq_w10 -c "SELECT id, written_on, body FROM notes"
 id | written_on |            body
----+------------+----------------------------
  1 | 5433       | asiento A-1 vendido a Luis
(1 row)
```

- La misma clave primaria, dos contenidos. Cada servidor es consistente por dentro; el sistema, no.
- PostgreSQL no puede juntar las dos historias (líneas de tiempo 1 y 2): alguien elige cuál sobrevive.

Del enlace cortado al split brain



Los dos aceptaron: el asiento A-1 quedó vendido a dos personas, con el mismo id.

SALIDA DE MAKE COMPARE-FAILOVER

```
confirmadas al cliente:        1..2669
existen en el nuevo principal: 1..1839 (1839 filas)
confirmadas que se PERDIERON:  830 (de la 1840 a la 2669)
```

Qué tendría que haberlo impedido: fencing

Que el líder anterior **no pueda** aceptar escrituras una vez reemplazado.

- No volver a levantarlo como líder. Depende de que el procedimiento se siga.
- Quitarle a los clientes (un proxy, una dirección que el failover mueve). Depende de que nadie conserve la vieja.
- Un número de época que seguidores y almacenamiento comprueban. Necesita consenso.

En la industria: GitHub, 21 de octubre de 2018

Un corte de red de 43 segundos entre centros de datos; su herramienta promovió réplicas de la otra costa. Quedaron escrituras en los dos lados. 24 horas y 11 minutos de servicio degradado, y una reconciliación a mano. Fuente: GitHub, *October 21 post-incident analysis*.

SECCIÓN 06

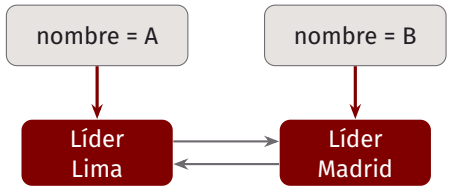
Varios líderes

06

Para qué, y a qué precio

Más de un nodo acepta escrituras; cada uno replica las suyas a los demás.

- Varios centros de datos: cada región escribe en el suyo.
- Aplicaciones sin conexión: cada dispositivo es un líder.
- Edición colaborativa.



El conflicto

Dos escrituras concurrentes sobre el mismo dato, las dos «confirmadas». **Nadie lo detectó al escribir**.

Qué hacer con un conflicto

- **Evitarlo**: todas las escrituras de un dato pasan por el mismo líder. Falla en los bordes (una región caída).
- **Una regla fija**: la última escritura gana *(LWW)*. Simple, converge, y pierde datos en silencio.
- **Conservar las dos versiones**: la aplicación las fusiona o el usuario elige. Toda lectura puede traer varias.
- **CRDT**: tipos de datos que convergen por construcción (contadores, conjuntos, texto). No todo dato se deja.

No hay una resolución correcta en general

Dos incrementos de un contador de visitas se suman. Dos ventas del mismo asiento no se pueden fusionar. Qué significa «resolver» depende de qué significa el dato.

La última escritura gana, y los relojes

```
make quorum: cada nodo sella sus escrituras con su reloj

== La última escritura gana, con el reloj de Lima adelantado 500 ms
t = 1000 ms, en lima:  "Rosa Quispe" con sello 1500
t = 1200 ms, en cusco: "Rosa Quispe Mamani" con sello 1200
después de replicar: lima = "Rosa Quispe", cusco = "Rosa Quispe"
```

- Convergieron, al valor anterior: la corrección, hecha 200 ms después, se descartó en los dos nodos.

Error común

LWW con el reloj de cada máquina: gana la escritura que ocurrió antes, y el sistema nunca avisa.

Versiones en vez de relojes

Registrar **qué conocía cada escritura** cuando se hizo: un contador por nodo, un **vector de versiones**.

Caso	Primera	Segunda	Comparación	El sistema
Cusco había recibido la de Lima	`{lima: 1}`	`{lima: 1, cusco: 1}`	anterior	se queda con la segunda
Ninguno conocía al otro	`{lima: 1}`	`{cusco: 1}`	concurrentes	conserva las dos

- En el primer caso gana la correcta, sin consultar ningún reloj.
- En el segundo, el sistema *sabe* que no puede elegir. El conflicto ahora se ve.
- Costo: un vector por valor, y lecturas que pueden devolver varias versiones.

SECCIÓN 07

Sin líder: quórums

07

La forma: *n*, *w* y *r*

- Cualquier réplica acepta escrituras y lecturas. No hay failover.
- *n*: réplicas que guardan cada dato.
- *w*: cuántas confirman una escritura.
- *r*: cuántas se consultan en una lectura.
- La lectura se queda con la **versión más alta**.

El simulador de la semana

`quorum/cluster.ts`: sin red ni azar. Quien lo usa dice a qué réplicas llega cada petición, y las pruebas recorren *todos* los casos.

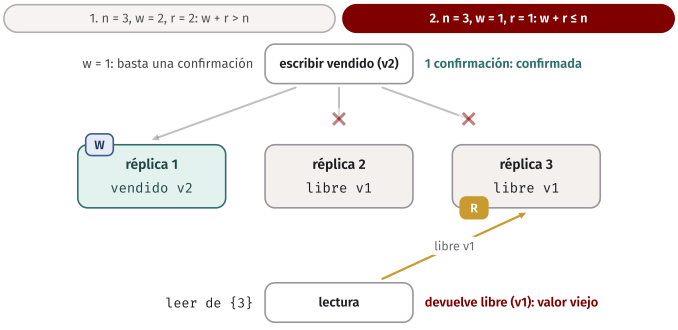
Con $w = 1$, $r = 1$, dos de tres lecturas ven un asiento vendido como libre

make quorum

```
== n = 3, w = 2, r = 2: la escritura llega a dos réplicas y se lee de otras dos
r1: vendido v2 | r2: vendido v2 | r3: libre v1
  leer de {1, 2} -> vendido (v2)
  leer de {1, 3} -> vendido (v2)
  leer de {2, 3} -> vendido (v2)

== n = 3, w = 1, r = 1: w + r <= n, y la lectura depende de a quién le toque
escritura confirmada: sí (con 1 réplica)
r1: vendido v2 | r2: libre v1 | r3: libre v1
  leer de {1} -> vendido (v2)
  leer de {2} -> libre (v1)
  leer de {3} -> libre (v1)
```

Los dos casos, réplica por réplica



Con w + r ≤ n, dos de las tres lecturas posibles devuelven el valor viejo.

SALIDA DE MAKE QUORUM

```
leer de {1} -> vendido (v2)
leer de {2} -> libre (v1)
leer de {3} -> libre (v1)
```

Caben W y R sin réplica en común: depende de a quién le toque.

El argumento del solapamiento

W: las réplicas que confirmaron la escritura. R: las que respondieron la lectura. Las dos, dentro de las mismas n.

$$|W \cap R| \;=\; |W| + |R| - |W \cup R| \;\geq\; w + r - n$$

- Si $w + r > n$: al menos **una réplica está en los dos conjuntos**. Confirmó la escritura y respondió la lectura.
- Si $w + r \leq n$: caben dos quórums sin nada en común. No pasa siempre: por eso pasa las pruebas.

Idea clave

$w + r > n$ no dice que todas las réplicas estén al día. Dice que toda lectura *toca* al menos una que sí lo está.

Mover el punto

Una escritura tolera $n - w$ réplicas caídas; una lectura, $n - r$. Con $n = 3$:

	Escribe con	Lee con	¿Solapan?	
$w = 2, r = 2$	1 caída	1 caída	sí	El punto medio: una mayoría, $\lfloor n/2 \rfloor + 1$
$w = 3, r = 1$	0 caídas	2 caídas	sí	Lee rápido; una caída detiene las escrituras
$w = 1, r = 3$	2 caídas	0 caídas	sí	Al revés
$w = 1, r = 1$	2 caídas	2 caídas	no	Rápido, disponible, y puede leer lo viejo

- Con $n = 5$: solo $w = 3$, $r = 3$ tolera dos caídas en las dos operaciones.
- Subir w compra durabilidad y cobra latencia y disponibilidad de escritura; subir r, lo mismo al leer.

Ponerse al día: read repair y anti-entropía

```
make quorum

== Read repair: la réplica 3 estuvo caída durante una escritura
antes:   r1: vendido v2 | r2: vendido v2 | r3: libre v1
leer de {2, 3} -> vendido; reparadas: {3}
después: r1: vendido v2 | r2: vendido v2 | r3: vendido v2
```

- **Read repair**: la lectura ya tiene las respuestas delante; le escribe el valor nuevo a la atrasada. Barato, y solo repara lo que alguien lee.
- **Anti-entropía**: un proceso de fondo compara réplicas y copia lo que falte. Repara todo; gasta red y disco.

Lo que los quórums no garantizan

La cuenta supone *una* escritura que terminó y, después, *una* lectura. Fuera de eso:

- **Una escritura que falló a medias**: llegó a una réplica, no alcanzó w. Nadie la deshace; una lectura posterior puede devolverla, y read repair la propaga.
- **Una réplica restaurada con datos viejos**: el valor nuevo queda en menos de w réplicas sin que nada falle.
- **Escrituras concurrentes**: ¿cuál es «la más nueva»? Hacen falta versiones, no relojes de pared.
- **Una lectura mientras la escritura está en curso**: puede ver el valor nuevo, y la siguiente, el viejo.

Error común

Leer «$w + r > n$» como «consistencia fuerte». No da transacciones ni ordena escrituras concurrentes.

Borrar: el dato resucita

Rosa quita la entrada de su carrito mientras la réplica 3 está caída. El borrado se confirma con dos.

```
make quorum

== Borrar con la réplica 3 caída, sin tombstones
borrado confirmado: sí
  r1: - | r2: - | r3 (caída): entrada VIP v1
vuelve la réplica 3; leer de {2, 3} -> entrada VIP; reparadas: {2}
  r1: - | r2: entrada VIP v1 | r3: entrada VIP v1
```

- Desde afuera es igual a «la réplica 2 no recibió la escritura»: una tiene el dato, la otra no.
- Read repair «arregla» a la réplica que estaba bien. La ausencia de un dato no lleva versión.

Tombstones: borrar es una escritura

make quorum

```
== Borrar con la réplica 3 caída, con tombstones
borrado confirmado: sí
  r1: tombstone v2 | r2: tombstone v2 | r3 (caída): entrada VIP v1
vuelve la réplica 3; leer de {2, 3} -> no existe; reparadas: {3}
  r1: tombstone v2 | r2: tombstone v2 | r3: tombstone v2
```

- Un **tombstone**: una marca de borrado con su versión, que se replica como cualquier valor.
- La versión 2 le gana a la 1, y read repair corrige a la réplica correcta.
- Se purgan solo cuando todas las réplicas los tienen; si no, el dato vuelve a resucitar.

SECCIÓN 08

Dynamo y cómo se elige

08

Dynamo: una arquitectura derivada de un escenario

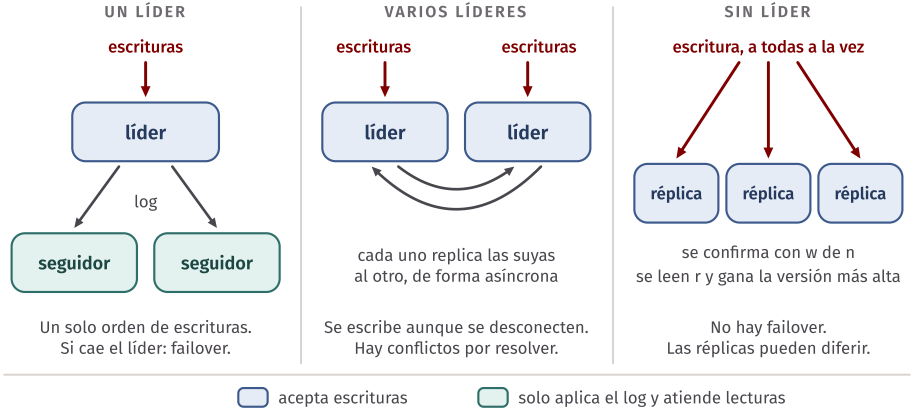
El escenario: un carrito de compras **siempre escribible**, aun con fallas de servidores y de red.

Requisito	Decisión	Lo que cuesta
Aceptar escrituras aunque fallen nodos	Sin líder	Las réplicas divergen
Que cada servicio elija su punto	n, r y w configurables	Elegirlos mal es fácil
No rechazar por un nodo caído	La escritura va a otro nodo, con una nota	El solapamiento ya no es seguro
No perder un «agregar al carrito»	Versiones en conflicto que la aplicación fusiona	Lo borrado puede reaparecer

Lo que hay que llevarse

Escrituras que no se pueden rechazar, sobre datos que se pueden fusionar. Sin una de las dos, el diseño deja de tener sentido. DeCandia y otros (2007), secciones 2 y 4.

Tres formas de replicar



Cómo se elige

	Cuando	Lo más difícil
Un líder	Hace falta un orden único de las escrituras y se tolera un failover	El failover
Varios líderes	Hay que escribir en varios lugares aunque se desconecten, y los conflictos son raros o tienen resolución natural	Resolver los conflictos
Sin líder	La disponibilidad de escritura importa más que leer siempre lo último	Elegir n, w, r y vivir con las versiones

En Tinkuy

Los asientos necesitan un orden único: un líder. Un contador de visitas toleraría las otras dos. La elección es **por dato**.

Laboratorio: una réplica de PostgreSQL

En parejas, con `code/week-10/` y dos terminales en el contenedor.

1. `make`; ¿quién es quién? (`pg_is_in_recovery()`); un `INSERT` en la réplica.
2. Escribir en el principal, leer en la réplica; `make lag` y `pg_stat_replication`.
3. `make burst`, sin demora y con `make delay MS=2000`.
4. `make anomaly`, tres veces: ¿cuántas de cien no encuentra?
5. Completar `lab/routing.ts` (`make test-lab`) y comparar con `make compare`.
6. `make sync-on`: latencia antes y después; detener la réplica y escribir.
7. Failover: `make writer`, `make kill-primary`, `pg-replica promote`, `make compare-failover`.
8. `svc start postgres`: escribir una fila distinta en cada servidor. ¿Cómo se llama esto?

Para matar al principal sirven `make kill-primary` o `svc kill postgres`: los dos dejan viva a la réplica.

Para la próxima clase

- Sin entregas esta semana. **P3 · Lecturas que escalan** se entrega en la semana 11.
- Terminar la tabla del paso 5 y juntar los resultados del failover con los de otras parejas.
- Semana 11, sharding y cache. Leer de *Designing Data-Intensive Applications* (2.ª ed.) el cap. 7 *Sharding*, y de Azure Architecture Center los patrones *Sharding* y *Cache-Aside*.

Pregunta

Una réplica reparte las lecturas. ¿Qué se hace cuando lo que ya no cabe en una máquina son las escrituras?

Lecturas de hoy: *Designing Data-Intensive Applications* (2.ª ed.), cap. 6; Kleppmann, apuntes de sistemas distribuidos de Cambridge, sección 5; DeCandia y otros, *Dynamo: Amazon's Highly Available Key-value Store* (2007), secciones 1, 2 y 4.