

Arquitectura de Software

Mensajería, eventos y consistencia

Colas y logs · Garantías de entrega · Outbox y consumidores idempotentes · Event sourcing y CQRS · Consenso

Gerson Garrido

Semana 13

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A

1. Por qué poner un broker en medio
2. Cola y log
3. Garantías de entrega
4. El orden de los mensajes
5. La outbox
6. Consumidores idempotentes

Sesión B

7. Event sourcing
8. CQRS
9. Consistencia y el teorema CAP
10. Consenso

Laboratorio: outbox y consumidor idempotente con NATS JetStream

Lectura: *Designing Data-Intensive Applications* (2.ª ed.), cap. 12 (mensajería y logs particionados), cap. 3 (*Event Sourcing and CQRS*) y cap. 10 (*Linearizability*, *Consensus*); Kreps, Narkhede y Rao, *Kafka* (2011), secc. 1–3; Kleppmann, *Please stop calling databases CP or AP* (2015).

La cuenta pendiente de la semana 5

La versión con eventos ganó todas las comparaciones, y dejó cuatro partidas sin pagar:

1. Guardar y publicar son **dos operaciones**: un evento puede no salir nunca.
2. Un mensaje puede llegar **dos veces**.
3. Los mensajes pueden llegar **en otro orden**.
4. Nadie sabe si los efectos **ya ocurrieron**.

> **Hoy**
>
> Se paga la cuenta: outbox, entrega at-least-once y consumidores idempotentes.
> Todo se corre: se matan procesos a la mitad y se cuenta lo que quedó.

En Tinkuy, un evento es un concierto. Hoy, un *evento* es un mensaje que dice que algo ya pasó.

SECCIÓN 01

Por qué poner un broker en medio

01

Lo que el broker da y lo que cuesta

Da

- El consumidor no necesita **estar arriba**: el mensaje espera.
- Ni ser **rápido**: el broker absorbe los picos.
- Puede haber **varios** consumidores, y el productor no los conoce.

Cuesta

Un componente más que operar y del que depender.

Quien publica no sabe **cuándo** se procesó su mensaje, ni si se procesó.

La falla ocurre después, en otro proceso, sin nadie esperando.

Alternativa: la llamada síncrona. Respuesta definitiva en el momento, y la disponibilidad de quien llama atada a la de todos los que llama. Mensaje, cuando el efecto puede ocurrir *después*.

Dos preguntas para cualquier sistema de mensajería

1. **¿Qué pasa si el productor es más rápido que el consumidor?**
   - **Descartar** mensajes.
   - **Encolar**: que esperen.
   - **Frenar al productor** hasta que haya lugar: backpressure.
2. **¿Qué pasa si un nodo cae?**
   - ¿Los mensajes estaban solo en memoria, en disco, replicados?
   - Si el consumidor muere a la mitad de un mensaje, ¿se pierde o vuelve?

> **Encolar no es gratis**
>
> Una cola tiene un tamaño. Cuando se llena, hay que elegir una de las otras dos.

Un stream con lugar para cinco mensajes recibe ocho

```
make demo-limits

== max_msgs = 5, discard = old
...
mensaje 8: guardado con la secuencia 8
stream: 5 mensajes, de la secuencia 4 a la 8
== max_msgs = 5, discard = new
...
mensaje 5: guardado con la secuencia 5
mensaje 6: RECHAZADO (maximum messages exceeded)
stream: 5 mensajes, de la secuencia 1 a la 5
```

Error común: una cola sin límite

No elimina la sobrecarga: la esconde hasta que se acaba el disco, y la latencia crece sin tope.

SECCIÓN 02

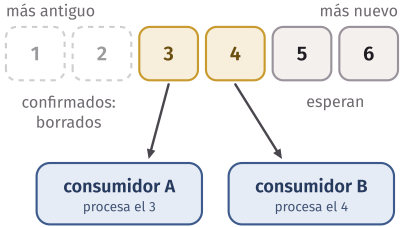
Cola y log

02

Los mismos seis mensajes, en una cola y en un log

COLA: EL BROKER REPARTE Y BORRA

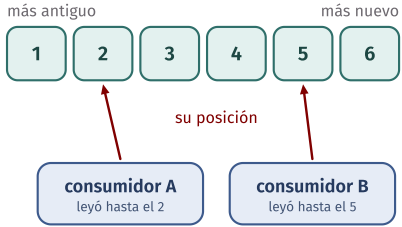
El broker lleva la cuenta, mensaje por mensaje.



Cada mensaje va a uno solo de los consumidores. Procesado y confirmado, deja de existir.

LOG: LOS MENSAJES SE QUEDAN

Cada mensaje tiene su offset; se agrega al final.



Leer no borra: cada consumidor lleva su posición. Puede volver atrás, y uno nuevo empieza desde el 1.

La cola: el broker reparte y borra

- El modelo de AMQP y de los brokers tradicionales.
- El broker lleva la cuenta **mensaje por mensaje**: cada mensaje va a **un** consumidor y, confirmado, se borra.
- Varios consumidores **compiten**: así se reparte el trabajo.
- Si uno no confirma, el mensaje va a otro, y el orden se pierde.

El log: los mensajes se quedan

- El modelo de Kafka: solo se agrega al final.
- **Leer no borra.** Cada lector lleva su posición, el **offset**.
- Un lector nuevo empieza **desde el principio**; uno que falló **vuelve atrás**.
- Para repartir: **particiones** por clave; cada una la lee un solo consumidor del consumer group.

Orden garantizado **dentro de una partición**, no entre particiones.

Cuál se elige

	Cola	Log
El mensaje, ya procesado	se borra	se conserva
Quién lleva la cuenta	el broker, por mensaje	cada lector, con un offset
Repartir trabajo	natural	por particiones
Orden	se pierde con varios consumidores	dentro de una partición
Lector nuevo	ve desde ahora	puede leer desde el principio
Reprocesar	no	sí

Cola

Tareas independientes y caras: enviar correos, generar un PDF.

Log

Orden por clave, varios sistemas con los mismos eventos, reprocesar.

La pregunta que decide: ¿qué pasa con el mensaje después de procesarlo? Una tarea se hace y desaparece; un hecho queda registrado.

En la industria: el artículo de Kafka (LinkedIn, 2011)

- Cuánto consumió cada consumidor **no lo guarda el broker**, sino el consumidor.
- Como el broker no sabe quién leyó qué, borra **por antigüedad** *(siete días, como valor típico)*.
- Beneficio lateral: un consumidor puede **retroceder** a un offset anterior y volver a consumir. Los autores dicen que eso viola el contrato habitual de una cola, y que resulta esencial.
- Entrega: **at-least-once**. Si a la aplicación le importan los duplicados, los descarta ella.
- Orden: garantizado dentro de una partición.

Kreps, Narkhede y Rao, *Kafka: a Distributed Messaging System for Log Processing* (2011), secciones 3.1 a 3.3.

NATS JetStream: stream y consumer

- **Stream**: guarda los mensajes de sus subjects, en orden, con un número de secuencia. Es el log.
- **Consumer**: la posición de un lector, guardada *en el servidor*.

```
nats consumer info W13_ORDERS first   (un mensaje confirmado, uno recibido sin confirmar)

Last Delivered Message: Consumer sequence: 2 Stream sequence: 2 Last delivery: 30ms ago
  Acknowledgment Floor: Consumer sequence: 1 Stream sequence: 1 Last Ack: 68ms ago
      Outstanding Acks: 1 out of maximum 1,000
  Redelivered Messages: 0
  Unprocessed Messages: 1
```

`Last Delivered`	hasta qué secuencia entregó
`Acknowledgment Floor`	hasta dónde está todo confirmado, sin huecos
`Outstanding Acks`	entregados que nadie confirmó: vuelven al vencer `Ack Wait`
`Unprocessed`	mensajes del stream que todavía no entregó

Como log: dos consumers, y cada uno recibe todo

`make worker C=reader-a` y después `make worker C=reader-b`

```
[proceso 1822, consumer reader-a] seq 1 (entrega 1): recibí la orden 1
...
[proceso 1822, consumer reader-a] seq 6 (entrega 1): recibí la orden 6

[proceso 1832, consumer reader-b] seq 1 (entrega 1): recibí la orden 1
...
[proceso 1832, consumer reader-b] seq 6 (entrega 1): recibí la orden 6
```

- `reader-b` empezó cuando `reader-a` ya había confirmado los seis. Los recibió igual.
- Leer no borra: cada consumer es una posición independiente.

Como cola: dos procesos con el mismo consumer

```
WORK_MS=200 make worker C=shared   (en dos terminales)

[proceso 1847, consumer shared] seq 1 (entrega 1): recibí la orden 1
[proceso 1847, consumer shared] seq 3 (entrega 1): recibí la orden 3
[proceso 1847, consumer shared] seq 5 (entrega 1): recibí la orden 5
[proceso 1846, consumer shared] seq 2 (entrega 1): recibí la orden 2
[proceso 1846, consumer shared] seq 4 (entrega 1): recibí la orden 4
[proceso 1846, consumer shared] seq 6 (entrega 1): recibí la orden 6
```

- Una sola posición y dos procesos que piden: cada mensaje va a **uno** solo.

> **En JetStream, log o cola lo decide cómo se lee**
>
> Un consumer por cada sistema que necesita todos los mensajes; varios procesos sobre un consumer para repartir el trabajo de ese sistema.

SECCIÓN 03

Garantías de entrega

03

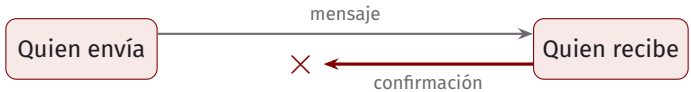
At-most-once y at-least-once

At-most-once

Se envía y no se reintenta. Llega una vez o ninguna. **Puede perderse.**

At-least-once

Se reintenta hasta recibir la confirmación. **Puede duplicarse.**



quien envía ve silencio: ¿no llegó el mensaje, o se perdió la confirmación?

En los dos casos reintenta. En el segundo, el receptor recibe el mensaje **otra vez**. Es el timeout de la semana 12: no se puede saber; solo diseñar para que repetir sea seguro.

Confirmar antes de procesar

make demo-ack (cinco mensajes; el proceso muere, con SIGKILL, mientras procesa el tercero)

```
== Confirmar ANTES de procesar (ACK_FIRST=1)
-- primer proceso (muere mientras procesa la secuencia 3)
...
[proceso 678, consumer ack-first] seq 3 (entrega 1): recibí la orden 3
-- segundo proceso, con el mismo consumer
[proceso 686, consumer ack-first] seq 4 (entrega 1): recibí la orden 4
...
[proceso 686, consumer ack-first] seq 5: orden 5 procesada
-- procesados: 1 2 4 5; perdidos: 3
```

At-most-once

El mensaje 3 se confirmó y el trabajo no se hizo. Nadie lo sabe.

Confirmar después de procesar

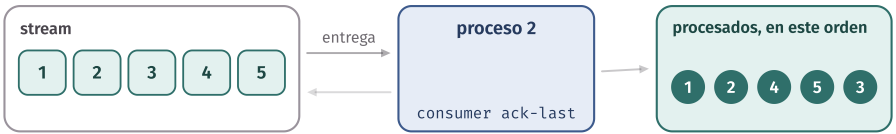
make demo-ack (segunda parte)

```
== Confirmar DESPUÉS de procesar
-- primer proceso (muere mientras procesa la secuencia 3)
...
[proceso 694, consumer ack-last] seq 3 (entrega 1): recibí la orden 3
-- segundo proceso, con el mismo consumer
[proceso 702, consumer ack-last] seq 4 (entrega 1): recibí la orden 4
...
[proceso 702, consumer ack-last] seq 5: orden 5 procesada
[proceso 702, consumer ack-last] seq 3 (entrega 2): recibí la orden 3
[proceso 702, consumer ack-last] seq 3: orden 3 procesada
-- procesados: 1 2 3 4 5; perdidos: ninguno
```

- **At-least-once**: venció `ack_wait` y el servidor lo entregó de nuevo *(entrega 2)*.
- El 3 llegó después del 4 y del 5. Y si el proceso hubiera muerto *después* del trabajo, se habría hecho dos veces.

La misma caída, con las dos políticas

1. confirmar antes de procesar | **2. confirmar después de procesar**



Vence ack_wait y el servidor entrega el 3 otra vez: entrega 2, después del 4 y del 5.

SALIDA DE MAKE DEMO-ACK

el consumidor confirma	procesados	perdidos	garantía
antes de procesar	1 2 4 5	3	**at-most-once**
después de procesar	1 2 3 4 5	ninguno	**at-least-once**

Exactly-once es un efecto, no una entrega

$$\text{efecto exactly-once} = \text{entrega at-least-once} + \text{consumidor idempotente}$$

- At-least-once garantiza que el efecto ocurra **por lo menos** una vez.
- La idempotencia *(semana 12)* garantiza que aplicarlo dos veces deje lo mismo que una.

Por qué el broker no puede darlo solo

No controla el efecto del consumidor. El efecto ocurre en otro sistema (una fila, un correo, un cobro), y el broker solo se entera por la confirmación, que puede perderse.

Error común: «en las pruebas siempre llegó una vez y en orden»

En las pruebas nada falla:

- ningún proceso muere entre el efecto y la confirmación;
- ninguna confirmación se pierde;
- ningún consumidor tarda más que `ack_wait`.

Un duplicado no es un caso raro

Es el funcionamiento normal de la entrega at-least-once cuando algo se reinicia, y en producción siempre hay algo reiniciándose: cada deploy lo hace.

Por eso las pruebas de la semana matan procesos y publican duplicados a propósito.

SECCIÓN 04

El orden de los mensajes

04

Ocho mensajes, leídos de cuatro maneras

```
make demo-order  (los impares tardan 60 ms; los pares, 5 ms)

1. Un lector
    orden en que terminaron: 1 2 3 4 5 6 7 8
2. Tres lectores del mismo consumer
    orden en que terminaron: 2 4 1 6 3 8 5 7
3. Un lector; la secuencia 3 falla una vez y se reintenta a los 300 ms
    orden en que terminaron: 1 2 4 5 6 7 8 3(entrega 2)
4. Lo mismo, con max_ack_pending = 1 (el servidor no entrega el siguiente sin confirmar el anterior)
    orden en que terminaron: 1 2 3(entrega 2) 4 5 6 7 8
```

- Repartir el trabajo y conservar el orden son objetivos opuestos *(2)*.
- Con un solo lector, basta un reintento para desordenar *(3)*.
- Orden estricto *(4)*: los mensajes 4 a 8 esperaron al reintento. Si el 3 no pasara nunca, no pasaría ninguno.

Diseñar para el desorden

1. **Efectos conmutativos**: sumar puntos da lo mismo en cualquier orden.
2. **Números de versión**: se ignora lo más antiguo que lo que ya se tiene.
3. **Clave de partición por entidad**: orden solo dentro de una orden. Lo más caro.

ordering/order-status.ts

```ts
export function applyIfNewer(view: StatusView, event: StatusEvent): void {
  const current = view.get(event.orderId);
  if (current && current.version >= event.version) return;
  view.set(event.orderId, { status: statusOf(event), version: event.version });
}
```

`OrderRefunded` (v2) llega antes que `OrderPaid` (v1): con «el último que llega gana», una orden devuelta queda pagada. Con versión, queda `refunded` en cualquier orden y con duplicados.

SECCIÓN 05

La outbox

05



La escritura doble, medida

errors/dual-write.ts

```ts
const order = await inTransaction((client) => insertPaidOrder(client, input));
crashPoint('CRASH_BETWEEN');          // SIGKILL, si la variable está definida
await publish(settings.subjects.paid, newOrderPaid(order));
```

CRASH_BETWEEN=1 make dual-write N=100 make counts

```
100 compras (dual-write): 0 procesos terminaron bien, 100 murieron a la mitad
órdenes guardadas                  100
mensajes en el stream                0
órdenes sin evento                 100
eventos de órdenes que no existen    0
```

Con `CRASH_BETWEEN=0.5`: 47 procesos murieron, y quedaron 100 órdenes, 53 eventos y 47 órdenes sin evento. Nada en el sistema lo sabe.

¿Y publicar primero?

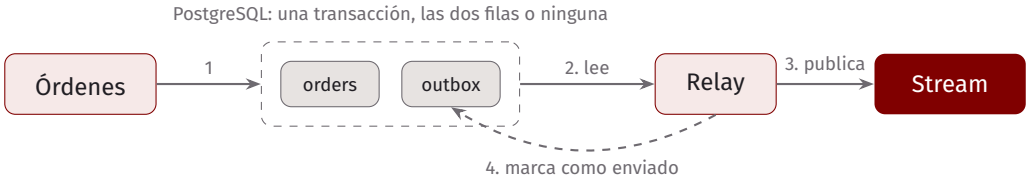
```
PUBLISH_FIRST=1 CRASH_BETWEEN=1 make dual-write N=100   make counts

100 compras (dual-write): 0 procesos terminaron bien, 100 murieron a la mitad
órdenes guardadas                    0
mensajes en el stream              100
órdenes sin evento                   0
eventos de órdenes que no existen  100
```

Cien correos de confirmación de compras que no ocurrieron.

> **No hay un orden correcto**
>
> Con una primero, el sistema calla algo que pasó; con la otra, anuncia algo que no pasó. El problema no es el orden: es que son **dos** escrituras en **dos** sistemas.

Outbox: una sola transacción, y un relay



- Publicar pasa a ser **una escritura más** de la misma base de datos *(atomicidad, semana 12)*.
- La compra **ya no habla con el broker**: con NATS caído, funciona y los eventos esperan.
- Alternativas: leer el WAL de la base de datos; two-phase commit.
- **Se paga**: latencia *(el relay pasa cada 200 ms)*, un proceso más y una tabla que limpiar.

La misma prueba, con outbox

```
CRASH_BETWEEN=0.5 make outbox N=100   make counts

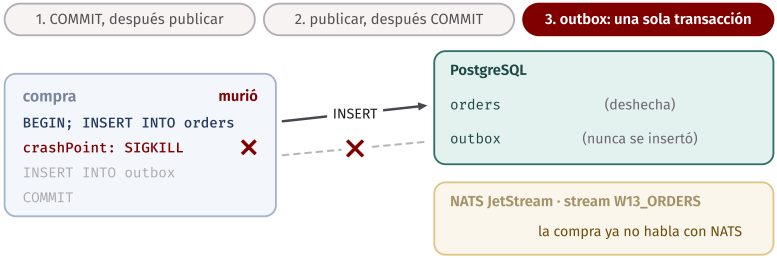
100 compras (outbox): 50 procesos terminaron bien, 50 murieron a la mitad
órdenes guardadas                    50
filas en la outbox                   50
mensajes en el stream                 0
```

```
make relay-once   make counts

[relay] 50 eventos publicados (solution/relay.ts)
mensajes en el stream                50
  eventos distintos                  50
órdenes sin evento                    0
eventos de órdenes que no existen     0
```

Los procesos que murieron dejaron la transacción abierta, y PostgreSQL la deshizo: ni orden ni evento.

La misma caída en las tres versiones de la compra



Nada a medias: el comprador ve un error y puede intentar de nuevo.

MAKE COUNTS TRAS CIEN COMPRAS CON CRASH_BETWEEN=1

	órdenes guardadas	mensajes en el stream	órdenes sin evento	eventos de órdenes que no existen
COMMIT, después publicar	100	0	100	0
publicar, después COMMIT	0	100	0	100
outbox	0	0	0	0

El relay: publicar y marcar son, otra vez, dos escrituras

El relay…	Si muere entre las dos	Resultado
marca y después publica	marcada, y el evento no salió	se pierde
publica y después marca	salió, y la fila sigue pendiente	se **repite**

```
tres compras   CRASH_AFTER_PUBLISH=1 make relay-once   make relay-once   make counts

make: *** [Makefile:90: relay-once] Killed
[relay] 3 eventos publicados (solution/relay.ts)
mensajes en el stream                4
  eventos distintos                  3
```

La outbox no elimina el duplicado: lo mueve a donde se puede manejar

Entrega at-least-once. Para la repetición hay remedio; para la pérdida, no.

Con `Nats-Msg-Id`, el stream descarta la repetición dentro de su ventana de duplicados (dos minutos): ayuda, y no cubre el tramo del stream al consumidor.

SECCIÓN 06

Consumidores idempotentes

06



Un consumidor que no es idempotente

```
make consumer C=points-naive   (10 puntos por orden; 4 mensajes de 3 órdenes)

[points-naive] seq 1 (entrega 1): orden 1 procesada
[points-naive] seq 2 (entrega 1): orden 1 procesada
[points-naive] seq 3 (entrega 1): orden 2 procesada
[points-naive] seq 4 (entrega 1): orden 3 procesada
```

buyer_email	orders	expected_points	points
rosa.quispe@correo.pe	3	30	40

- La orden 1 se contó dos veces.
- Las dos fueron **entrega 1**: para el servidor son dos mensajes distintos.
- Lo único que las une es el `id` del evento, que viaja dentro del mensaje.

El id del evento, en la misma transacción que el efecto

consumers/report.ts

```
1  await inTransaction(async (client) => {
2    const first = await client.query(
3      'INSERT INTO processed_events (consumer, event_id) VALUES ($1, $2) ON CONFLICT DO
         NOTHING', [...]);
4    if (first.rowCount === 0) return; // ya se procesó: no se suma de nuevo
5    await client.query(`INSERT INTO ${target.table} ... ON CONFLICT (concert_id) DO
       UPDATE SET ...`, [...]);
6  });
```

- **Misma transacción**: si no, es otra escritura doble.
- **Un `INSERT`**, no `SELECT` y después `INSERT`: comprobar y actuar vendía dos veces un asiento.
- **Un duplicado termina bien**: se confirma. Con puntos así, los 4 mensajes dan **30**.

Del relay que muere al consumidor que suma



Tres órdenes: 40 puntos sin idempotencia, 30 con el id anotado junto con la suma.

La tabla de eventos procesados: lo que cuesta

Tabla de ids procesados

General: sirve para cualquier efecto en la misma base de datos. Cuesta una escritura más por mensaje y una tabla que hay que limpiar.

Efecto idempotente por naturaleza

`SET status = 'paid'`; un `INSERT` con la orden como clave. Sin tabla aparte, y no siempre existe: sumar no lo es.

Cuando el efecto ocurre fuera de la base de datos

Un correo, un cobro: ninguna transacción une el id con el efecto. La clave de idempotencia se envía al otro sistema (semana 12), o se acepta la repetición.

Un mensaje que nunca se va a procesar

```
POISON_ORDER=3 make consumer C=points

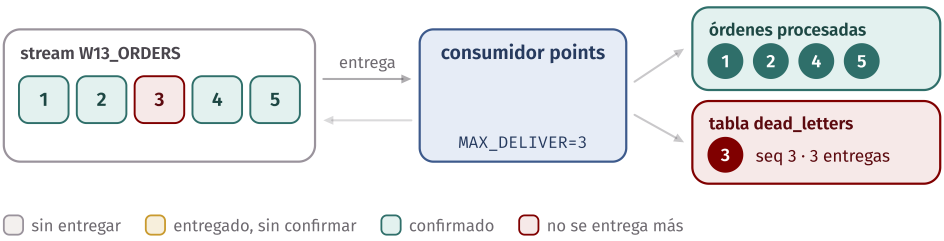
[points] seq 2 (entrega 1): orden 2 procesada
[points] seq 3: la entrega 1 falló (Error: no puedo procesar la orden 3)
[points] seq 4 (entrega 1): orden 4 procesada
[points] seq 5 (entrega 1): orden 5 procesada
[points] seq 3: la entrega 2 falló (Error: no puedo procesar la orden 3)
[points] seq 3: la entrega 3 falló (Error: no puedo procesar la orden 3)
...
```

- Sin límite: se reintenta para siempre. Con orden estricto, además, bloquea a los que vienen detrás.
- Señal: el `Acknowledgment Floor` no avanza mientras las entregas siguen.
- Salida: un **máximo de entregas** y una **dead letter queue**, donde alguien lo mira.

Con `MAX_DELIVER=3`: tres intentos, y una fila en `dead_letters` con `stream_seq 3`, `deliveries 3` y el motivo. Faltan 10 puntos, y ahora alguien lo sabe.

La dead letter, entrega por entrega

UN MENSAJE ENVENENADO, CON UN MÁXIMO DE TRES ENTREGAS



El consumer queda limpio y nada se perdió de vista: alguien tiene que resolver la orden 3.

Cómo avisa JetStream que se agotaron las entregas

```
make demo-max-deliveries   (max_deliver = 3; el consumidor solo hace ack o nak)

 310 ms  seq 2, deliveryCount 2: falla, nak
 632 ms  seq 2, deliveryCount 3: falla, nak
aviso del servidor en $JS.EVENT.ADVISORY.CONSUMER.MAX_DELIVERIES.W13_DEMO.poisoned
   {"type":"io.nats.jetstream.advisory.v1.max_deliver", ... ,"stream":"W13_DEMO","consumer":"poisoned","stream_seq":2,"deliveries":3}
```

Señal	Qué trae	Cuándo falla
`deliveryCount` en la entrega	el mensaje y el error	si el proceso no llega vivo a la última entrega
Aviso `MAX_DELIVERIES`	stream, consumer y secuencia	si nadie está suscrito: el aviso no se guarda

Error común: un límite de entregas sin dead letter, o una dead letter que nadie mira. El mensaje desaparece del radar.

Tiempos en una laptop con Apple Silicon, dentro de Docker; importa la secuencia.

Reprocesar: una tabla reconstruida desde el stream

```
make replay   (13 mensajes de 12 órdenes; un consumer nuevo lee desde la secuencia 1)

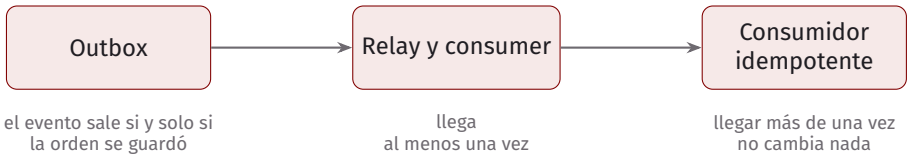
Reconstruido con el consumer "report-rebuild": 13 mensajes leídos desde la secuencia 1.
...
Las dos tablas son iguales.
```

Para que reprocesar sea seguro, el consumidor debe:

- ser **idempotente** *(el stream tiene duplicados)*;
- depender **solo del contenido** de los eventos, no del estado de hoy;
- **no tener efectos hacia afuera** *(reprocesar el correo reenvía toda la historia)*.

Si el consumidor original mandó una orden a la dead letter, la reconstrucción tiene una orden más: comparar una vista con su reconstrucción es una manera de auditar a un consumidor.

La cadena completa



Idea clave

Ninguno alcanza solo. Los tres juntos dan un efecto que ocurre exactamente una vez.

En Tinkuy (P4): `OrderPaid` sale de una outbox a un stream; notificaciones y reportes llevan cada uno su posición. Qué eventos, qué consumidores y qué se hace con una dead letter lo deciden ustedes.

SECCIÓN 07

Event sourcing

07



El estado como fold de los eventos

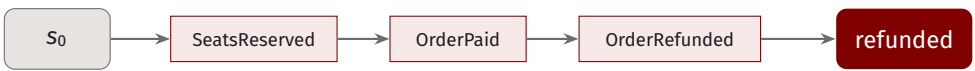
Lo habitual

Se guarda el **estado actual** y se sobrescribe. Cómo llegó ahí no está en ninguna parte.

Event sourcing

Se guarda la **secuencia de eventos**, que nunca se modifican. El estado es un dato derivado.

$$s_n = \text{apply}(\ldots \text{apply}(\text{apply}(s_0, e_1), e_2) \ldots, e_n)$$



aplicar solo hasta aquí:
el estado *antes* de la devolución

Una orden y su historia

position	aggregate_id	type	version	position	aggregate_id	type	version
1	order-1	SeatsReserved	1	4	order-3	SeatsReserved	1
2	order-1	OrderPaid	2	5	order-3	OrderPaid	2
3	order-2	SeatsReserved	1	6	order-1	OrderRefunded	3

```
make es   (arriba, el log es_events que imprime el mismo comando)

== La orden 1: su estado es el fold de su historia
  estado actual:     {"status":"refunded","concertId":1, ... ,"quantity":2,"totalCents":32000}
  estado en la v2:   {"status":"paid","concertId":1, ... ,"quantity":2,"totalCents":32000}
```

- No existe una tabla de órdenes: el estado se calcula leyendo los eventos.
- **Comando** (*`RefundOrder`*): un pedido, se puede rechazar. **Evento** (*`OrderRefunded`*): ya ocurrió.
- Ciclo de una escritura: leer la historia, `fold`, **decidir** *(aquí viven los invariantes)*, registrar.

Lo que da y lo que cuesta

Da

- Historia completa y auditable.
- El estado de cualquier momento.
- Vistas nuevas con eventos antiguos.

Cuesta

Leer exige una vista materializada.

Los eventos son un contrato **para siempre** (semana 9).

Corregir es agregar otro evento.

Borrar datos personales es difícil.

Se justifica donde la historia *es* el negocio: contabilidad, pagos, inventario. Alternativa barata: tabla de estado más tabla de auditoría.

Error común: aplicarlo a todo el sistema. Se adopta por agregado, donde la historia importa.

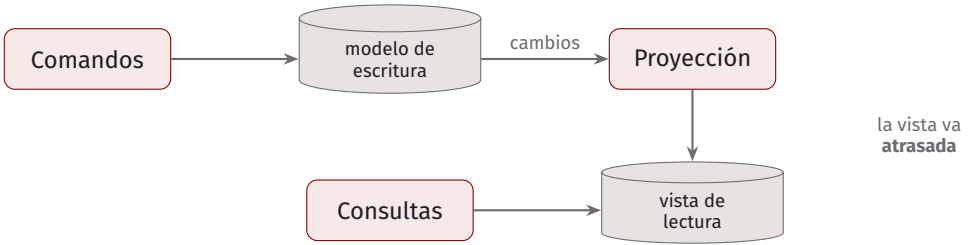
SECCIÓN 08

CQRS

08

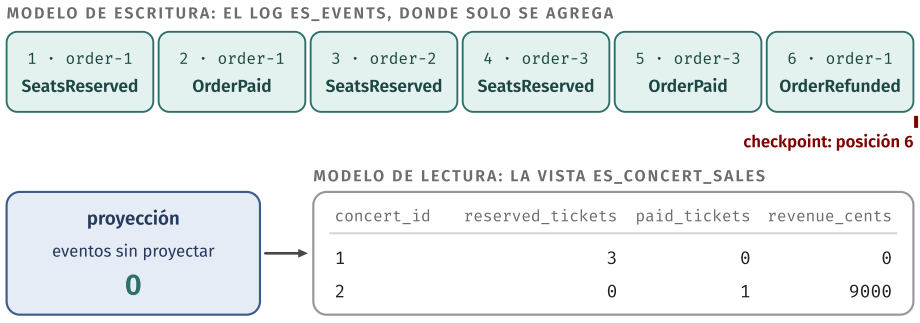


Dos modelos: uno para escribir, otro para leer



- Escribir pide un modelo normalizado que proteja los invariantes; leer, la respuesta casi hecha.
- Sirve cuando leer y escribir tienen **formas** o **cargas** muy distintas.
- **Se paga**: consistencia eventual entre escribir y leer, y una proyección detenida deja una vista que responde rápido, sin errores y con datos viejos.

La proyección, evento por evento



Al día: cero eventos sin proyectar. Mientras tanto, quien leyó la vista vio datos atrasados.

La vista va atrasada

```
make es   (segunda parte: seis eventos ya confirmados)

== La vista de lectura, antes de que corra la proyección
  eventos sin proyectar: 6
  (vacía)
== La proyección aplica cuatro eventos
  eventos sin proyectar: 2
== La proyección se pone al día
  eventos sin proyectar: 0
```

	tras cuatro eventos			al día		
concert_id	reserved	paid	revenue_cents	reserved	paid	revenue_cents
1	3	2	32000	3	0	0
2	1	0	0	0	1	9000

- A medio camino, la vista dice algo que *era* verdad hace dos eventos.
- Es el retraso de replicación de la semana 10. A cambio, la vista se puede **reconstruir**.

CQRS sin event sourcing

- Event sourcing obliga a tener vistas: casi siempre trae CQRS.
- Al revés no. **CQRS sin event sourcing es lo más frecuente**: tablas corrientes para escribir, y una vista de lectura alimentada por la outbox.
- Ya lo construyeron: `orders` es el modelo de escritura, `sales_report` es una vista, y el consumidor `report` es su proyección.

> **En Tinkuy**
>
> El reporte de ventas es hoy una consulta que agrupa órdenes, ítems, asientos y secciones cada vez que alguien lo pide. La alternativa: una vista que se actualiza con cada `OrderPaid`. ¿Es aceptable que vaya unos segundos atrasada? Es una pregunta de requisitos.

Fowler: para la mayoría de los sistemas, CQRS agrega una complejidad riesgosa (*CQRS*, martinfowler.com).

SECCIÓN 09

Consistencia y el teorema CAP

09

Una palabra, varios significados

- La **C de ACID** *(semana 12)*: los invariantes se mantienen. Depende de la aplicación.
- La **consistencia entre réplicas** *(semana 10)*: que las copias coincidan.

Hoy: **qué tan actual y coherente es lo que se lee** de un sistema con varias copias.

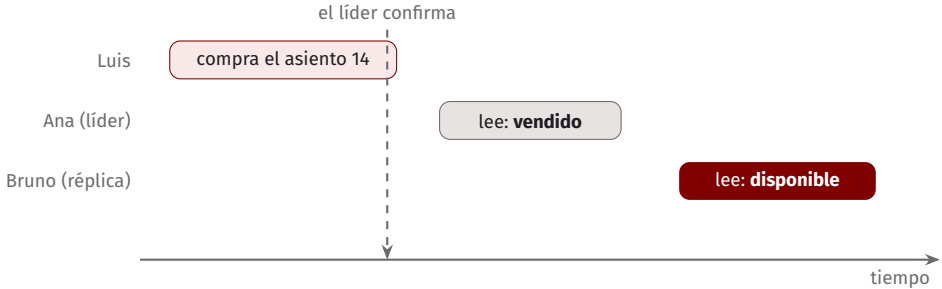
Linealizable

Como si hubiera **una sola copia**, y cada operación ocurriera en un instante entre su inicio y su fin.

Consistencia eventual

Si dejan de llegar escrituras, las copias **acabarán** coincidiendo. No dice cuándo.

Linealizabilidad: nadie lee después un valor más viejo



- Bruno leyó, *después* que Ana, un valor más viejo que el de Ana. Con una sola copia no podría pasar.
- Un líder con una réplica asíncrona **no es linealizable** *(lo midieron en la semana 10)*.
- No es serializabilidad: aquella habla de transacciones; esta, de operaciones sobre un dato y del tiempo real.

Dónde hace falta de verdad

Sí

Elegir un líder: dos a la vez es split brain.

Unicidad: un asiento, un nombre de usuario.

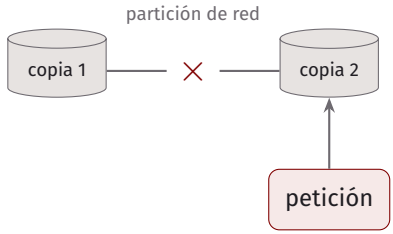
Un lock distribuido.

No: casi todo lo demás

- El catálogo de conciertos.
- El reporte de ventas.
- El saldo de puntos.
- «Quedan 12 entradas», como información.

La pregunta, dato por dato: ¿qué se rompe si alguien lee un valor de hace dos segundos?

Lo que cuesta, y lo que dice el teorema CAP



Latencia: coordinar las copias en cada operación, haya fallas o no.

Disponibilidad: la copia aislada tiene dos opciones.

- **Responder** con lo que tiene: puede ser un valor viejo. Deja de ser linealizable.
- **No responder**: sigue siendo linealizable y deja de estar disponible.

Eso, y nada más, dice el teorema CAP

No es «elijan dos de tres»: las particiones no se eligen, ocurren. Se elige qué hace cada parte del sistema **mientras dura** una partición.

Lo que el teorema CAP no dice

- **No clasifica bases de datos en «CP» y «AP».** Un líder con réplicas asíncronas: quien quedó separado del líder no puede escribir *(no es disponible)*, y leer de una réplica no es linealizable. Ni lo uno ni lo otro.
- **No habla de latencia.** En operación normal, el trade-off de todos los días es consistencia contra latencia.
- **Un solo tipo de falla** *(la partición)* y un modelo mínimo *(un dato que se lee y se escribe)*.
- **No obliga a una sola respuesta**: vender un asiento, no responder; mostrar el catálogo, responder.

Kleppmann, *Please stop calling databases CP or AP* (2015). Su ejemplo: ZooKeeper, que usa consenso y suele llamarse «CP», por omisión no da lecturas linealizables, y el lado minoritario de una partición no procesa escrituras.

SECCIÓN 10

Consenso

10

El problema, y las mayorías

Consenso: que varios nodos se pongan de acuerdo en un valor (en la práctica, en un **orden de operaciones**: un log replicado), aunque algunos fallen. Es lo que hay detrás de elegir un líder sin split brain.

Nodos n	Mayoría	Tolera
3	2	1
4	3	1
5	3	2
7	4	3

$$\text{mayoría} = \left\lfloor \frac{n}{2} \right\rfloor + 1$$

- Raft y Paxos necesitan una **mayoría** para avanzar.
- Dos mayorías del mismo grupo **siempre comparten un nodo**: una decisión tomada no se contradice.
- Si la red se parte en dos, **a lo más un lado** tiene mayoría. El otro se detiene.
- La minoría elige no responder: es el teorema CAP.

El consenso se usa, no se implementa

- Los algoritmos son famosos por lo fácil que es implementarlos mal.
- Se usa a través de un componente que ya lo resuelve: la base de datos, etcd o ZooKeeper, el propio broker *(JetStream en cluster replica cada stream con un grupo Raft)*.
- La decisión de arquitectura: **qué datos lo necesitan**, y mantenerlos pocos.

Error común: consistencia fuerte «por si acaso»

Cada dato que la exige ata la disponibilidad del sistema a la de un quórum, y paga latencia en cada operación, también cuando nada falla.

En Tinkuy: una sola verdad, la disponibilidad de un asiento **al venderlo**. El listado, el mapa de asientos y el reporte pueden venir de una réplica, una cache o una vista atrasada.

Diseñar con consistencia eventual

Elegir consistencia eventual traslada la decisión a la interfaz: **qué ve el usuario mientras tanto**.

Honesto

«Su compra está en proceso; le avisaremos por correo cuando sus entradas estén listas.»

No lo es

Un saldo de hace una hora, sin ninguna indicación: el usuario cree estar viendo el presente.

- **Estados intermedios explícitos**: la orden está `pending`.
- **Decir de cuándo es el dato**: «ventas hasta las 10:42».
- **Read-your-writes** para quien acaba de escribir.

SECCIÓN 11

Laboratorio

11

Laboratorio: outbox y consumidor idempotente

En parejas, en `code/week-13/`. Una fila por paso: qué esperaban, qué vieron, qué garantía lo explica.

1. **Primer contacto**: un stream, tres mensajes, un consumer; `nats stream info` y `nats consumer info`.
2. **Log o cola**: dos consumers; después, dos procesos con el mismo.
3. **Escritura doble**: `CRASH_BETWEEN=1 make dual-write N=100` y `make counts`.
4. **Outbox**: completar `lab/outbox.ts` y `lab/relay.ts` (*`make test-lab`*); repetir el paso 3.
5. **Duplicados**: `CRASH_AFTER_PUBLISH=1`. ¿Qué garantía tiene su outbox?
6. **Puntos**: contar con `points-naive`; hacer idempotente `lab/points.ts`.
7. **Dead letter**: `POISON_ORDER=3`, sin y con `MAX_DELIVER=3`.
8. **Reprocesar**: `make replay`. ¿Qué debe cumplir el consumidor?

Antes de escribir el relay: ¿qué queda si el proceso muere en cada línea?

Resumen

- **Cola**: reparte y borra. **Log**: conserva, y cada lector lleva su offset.
- Confirmar antes: at-most-once. Después: at-least-once. **Exactly-once** es at-least-once más idempotencia.
- El orden solo existe con un lector y sin reintentos que se adelanten: se diseña para el desorden.
- **Outbox**, **relay** y **consumidor idempotente**: tres eslabones, y ninguno alcanza solo.
- **Event sourcing**: el estado es el fold de los eventos. **CQRS**: vistas de lectura, atrasadas.
- **Linealizable**: como una sola copia; cuesta latencia siempre y disponibilidad en una partición *(CAP)*.
- **Consenso**: necesita una mayoría; se usa, no se implementa, y para pocos datos.

Para la próxima clase

- **P4 · Pagos como servicio** se entrega en la semana 14.
- Terminar la tabla del laboratorio, con sus ocho pasos.
- Semana 14, diseñar para las fallas. Leer de Nygard, *Release It!* (2.ª ed.): cap. 4 *Stability Antipatterns* y cap. 5 *Stability Patterns*.
- Lecturas libres: *Site Reliability Engineering*, cap. 21 *Handling Overload*, cap. 22 *Addressing Cascading Failures*, cap. 4 *Service Level Objectives* y cap. 6 *Monitoring Distributed Systems*; Brooker, *Exponential Backoff And Jitter* (2015).

Pregunta

El relay reintenta publicar cuando NATS no responde. ¿Cada cuánto debería reintentar, y qué pasa si mil relays reintentan todos al mismo tiempo?

Lecturas de hoy: *Designing Data-Intensive Applications* (2.ª ed.), cap. 12, cap. 3 (*Event Sourcing and CQRS*) y cap. 10; Kreps, Narkhede y Rao, *Kafka* (2011), secc. 1–3; Kleppmann, *Please stop calling databases CP or AP* (2015).