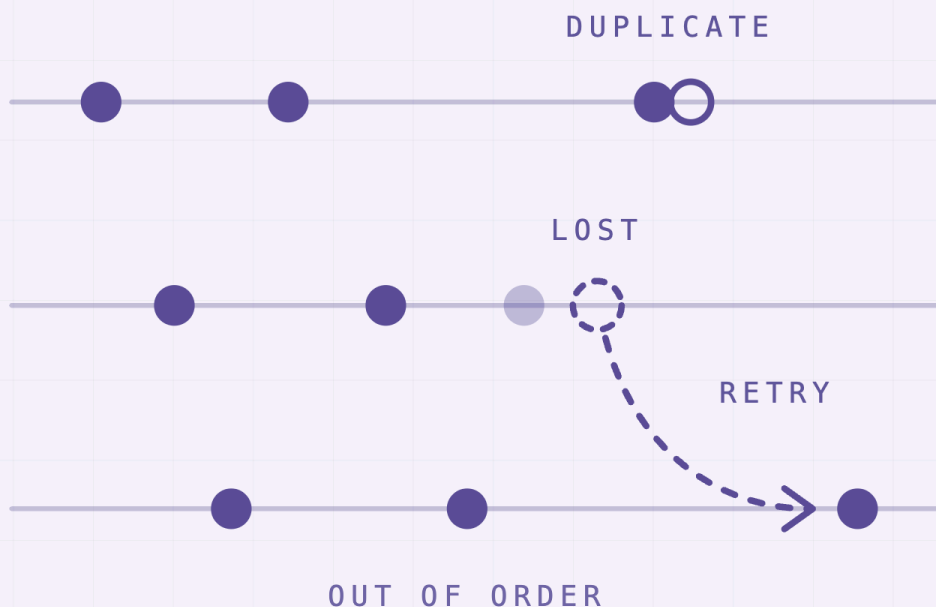




Diseño de sistemas confiables orientados a eventos

Diseño de sistemas confiables orientados a eventos

Mensajería, retries, idempotencia, outbox, orden y evolución de esquemas

Muhammet Şafak

Índice

Prefacio	6
PARTE I — FUNDAMENTOS	13
Capítulo 1 — El engaño del happy path	14
Capítulo 2 — Semántica de la mensajería	26
Capítulo 3 — Un modelo de fallos práctico	43
PARTE II — PUBLICAR CON CONFIABILIDAD	62
Capítulo 4 — El problema del dual write y el transactional outbox	63
Capítulo 5 — Relays, polling y change data capture	83
PARTE III — CONSUMIR CON CONFIABILIDAD	104
Capítulo 6 — Retries bien hechos	105
Capítulo 7 — Idempotencia	130
Capítulo 8 — Poison messages y dead letters	153
Capítulo 9 — Orden	176
PARTE IV — FLUJOS DE TRABAJO Y CONTRATOS	198
Capítulo 10 — Sagas y flujos de trabajo de larga duración	199
Capítulo 11 — Diseño de eventos	221
Capítulo 12 — Evolución de esquemas	241
PARTE V — OPERAR Y DEMOSTRAR	263
Capítulo 13 — Observabilidad y conciliación	264
Capítulo 14 — Recuperación, desastres y migración de brokers	287
Capítulo 15 — Cómo probar sistemas orientados a eventos	313
Capítulo 16 — Una lista de verificación de confiabilidad y reflexiones finales	331

Prefacio

Imagina una tarde de martes en Parcelly, un marketplace que no existe. El flujo de checkout lleva meses funcionando bien. Una persona hace un pedido, el servicio Orders lo escribe en su base de datos y un evento `OrderPlaced` sale hacia el message broker. Payments autoriza la tarjeta. Inventory reserva el stock. Shipping programa el paquete. Todos los dashboards están en verde.

Entonces un deploy atraviesa Orders. Cada instancia trata un pedido de la misma manera: lo confirma en su base de datos y después publica `OrderPlaced` en el message broker. Durante unos segundos, unas instancias se apagan mientras otras arrancan. Una instancia confirma un pedido y se detiene antes de la llamada de publicación. El pedido queda guardado de forma segura, y el evento nunca sale. Nada se cae con estruendo y ninguna alerta se dispara. A la mañana siguiente, quien compró ve un pedido marcado como realizado, pero Payments nunca autorizó una tarjeta e Inventory nunca reservó ni un solo artículo. Orders cree que el trabajo está en marcha, y el resto del sistema nunca ha oído hablar de él.

Nadie escribió un bug en el sentido habitual. Cada línea de código hizo lo que decía. El sistema falló en el espacio entre las líneas: entre un commit en la base de datos y una llamada de red, entre que un mensaje se entrega y que se entiende, entre “lo enviamos” y “lo hicieron”.

De ese espacio trata este libro.

Por qué existe este libro

La arquitectura orientada a eventos tiene un problema de reputación, y se lo ha ganado desde ambos lados. Quienes la defienden describen servicios débilmente acoplados que escalan de forma independiente y evolucionan sin coordinación. Quienes la critican describen espagueti distribuido, donde nadie sabe decir qué ocurre cuando se hace un pedido. Ambos describen el mismo sistema en días distintos.

Lo que separa un día del otro rara vez es la elección del broker, del framework o ni siquiera del diagrama de arquitectura. Es una lista corta de preguntas poco vistosas. ¿Qué pasa si este mensaje se entrega dos veces? ¿Qué pasa si nunca llega? ¿Qué pasa si llega después del mensaje que debía seguirlo? ¿Qué pasa si el consumidor falla absolutamente todas las veces? ¿Qué pasa cuando el productor necesita cambiar un campo y cien consumidores siguen ejecutando el código antiguo?

Esas preguntas son la tesis de este libro. Los sistemas confiables orientados a eventos se construyen en torno a la **semántica de fallos**, no a los message brokers: para una unión dada, qué puede fallar, cómo lo observarías y qué garantía sobrevive. Una unión es cualquier lugar donde un componente entrega trabajo a otro, como un commit en la base de datos seguido de una publicación, o una entrega seguida de un efecto. Los patrones de este libro existen por la red y por la base de datos, no por ningún producto concreto, y un broker es un componente más con sus propias uniones.

Los equipos suelen toparse con estas preguntas de una en una, en producción, en el orden en que al sistema le da por fallar. Cada respuesta se aprende por separado, y las lecciones rara vez acaban en un solo lugar. La documentación de un broker te explica cómo publicar y consumir. Dice mucho menos sobre cómo es un servicio correcto alrededor de esos dos verbos.

Este libro intenta ser ese lugar. Es una guía de los patrones que hacen confiables los sistemas orientados a eventos: el transactional outbox, los consumidores idempotentes, los retries con backoff y jitter, el backpressure, las dead-letter queues, las estrategias de orden, las sagas, el diseño cuidadoso de eventos, la evolución de esquemas, la observabilidad, la recuperación ante datos perdidos y ante un cambio de broker, y las pruebas. He procurado tratar cada uno como una decisión con precio, no como una regla que seguir. Un patrón que es adecuado para los pagos de Parcelly puede ser un desperdicio para sus notificaciones, y el libro lo dirá cuando importe.

También quiero dejar claro lo que el libro no es. No es un tutorial de ningún broker en particular. No es un catálogo de todos los patrones de integración empresarial que alguna vez recibieron nombre. No promete que seguirlo vuelva correcto tu sistema. La confiabilidad no es una propiedad

que se añade; es un conjunto de fallos para los que has decidido, de forma deliberada, cómo manejarlos, y otro conjunto que has decidido, también de forma deliberada, aceptar. Mi objetivo es hacer visibles esas decisiones.

A quién va dirigido este libro

El libro está escrito para **quienes hacen ingeniería de backend y de plataforma** y construyen u operan servicios que se comunican mediante eventos, y para los **tech leads** que tienen que decidir cómo deben comportarse esos servicios cuando algo sale mal. Si alguna vez has mirado fijamente una dead-letter queue que creció durante la noche, has discutido si un consumidor necesita ser idempotente o has heredado un topic cuyos eventos nadie sabe explicar del todo, eres a quien tengo en mente.

No necesitas ser experto en mensajería. Conviene que tengas soltura con lo siguiente:

- Las bases de datos relacionales y las transacciones: qué es un commit, qué hace una restricción única y qué significa el aislamiento en términos generales.
- Los servicios que se hablan por red, y el vocabulario básico de los timeouts y de la solicitud y respuesta al estilo HTTP
- La idea general de un message broker: algo que acepta mensajes de los productores y se los entrega a los consumidores.

No necesitas experiencia con ningún producto en particular. Cuando un capítulo nombra Kafka, RabbitMQ o Amazon SQS, lo hace como ejemplo de una familia de sistemas, y describe un comportamiento que vale en general y no en una versión concreta. Si no has usado ninguno, el libro seguirá teniendo sentido. Si has usado uno a fondo, descubrirás que parte de lo que sabes es específico de él, y el libro intentará señalar qué partes.

Si eres tech lead y no escribes código todos los días, puedes leer el escenario de apertura, la sección de trade-offs y las ideas clave de cada capítulo y obtener la mayor parte del valor para tomar decisiones. El pseudocódigo está ahí para quienes necesitan ver dónde queda exactamente el límite de la transacción.

Cómo está organizado el libro

El libro tiene cinco partes, y están pensadas para leerse en orden la primera vez y para usarse en cualquier orden después.

Parte I, Fundamentos (capítulos 1 a 3). Empezamos desarmando el happy path. El Capítulo 1 muestra cómo un flujo que parece simple esconde una docena de decisiones. El Capítulo 2 precisa lo que realmente significan las garantías de la mensajería, y por qué la entrega at-least-once es la base realista. El Capítulo 3 construye un modelo de fallos práctico, un pequeño vocabulario para las formas en que las cosas se rompen, desde los timeouts y los crashes hasta los mensajes que siempre fallan y la pérdida de un almacén entero. El resto del libro lo usa como lenguaje común.

Parte II, Publicar con confiabilidad (capítulos 4 y 5). El Capítulo 4 aborda el problema del dual write, el hueco exacto de la escena de apertura, e introduce el transactional outbox. El Capítulo 5 cubre la maquinaria que mueve las filas del outbox al broker, compara los relays con polling con el change data capture, y explica qué te cuesta cada uno.

Parte III, Consumir con confiabilidad (capítulos 6 a 9). Es la parte más larga, porque en los consumidores es donde ocurren la mayoría de los efectos duplicados, las colas atascadas y las actualizaciones perdidas. El Capítulo 6 trata los retries: cuándo reintentar, cuándo no, por qué el backoff necesita jitter y cómo evitar que un consumidor se sobrecargue con sus propios retries. El Capítulo 7 trata la idempotencia y el inbox. El Capítulo 8 se ocupa de los poison messages y las dead letters. El Capítulo 9 se vuelca en el orden, y en lo poco que suele poder exigirse.

Parte IV, Flujos de trabajo y contratos (capítulos 10 a 12). El Capítulo 10 introduce las sagas para los flujos de trabajo que abarcan varios servicios y no pueden compartir una transacción. El Capítulo 11 trata el diseño de los eventos mismos: qué va en ellos, qué se queda fuera, cómo se nombran y cómo sobrevive un envelope a ser escrito y leído en varios lenguajes. El Capítulo 12 cubre la evolución de esquemas, la disciplina silenciosa que permite que productores y consumidores cambien a velocidades distintas.

Parte V, Operar y demostrar (capítulos 13 a 16). El Capítulo 13 trata de ver lo que hace tu sistema, y de la conciliación, el hábito de contrastar la visión de un sistema con la de otro. El Capítulo 14 trata de recuperarse de

datos perdidos, de una región perdida y de un cambio de broker. El Capítulo 15 explica cómo probar código cuyos peores bugs solo aparecen bajo duplicación, retraso y reordenamiento. El Capítulo 16 reúne todo en una lista de verificación de confiabilidad y cierra el libro.

Un apéndice contiene un glosario y una referencia rápida de los patrones, para que puedas consultar algo sin releer un capítulo.

Cada capítulo del 1 al 15 sigue la misma forma. Abre con un escenario breve, explica por qué el problema es difícil, presenta el patrón y pasa después a dos secciones que considero las más valiosas: **Trade-offs y costos**, que dice cuándo no usar el patrón, y **Errores comunes**. Cada capítulo termina con cinco a siete ideas clave. Si tienes poco tiempo, esas tres partes te llevarán más lejos de lo que imaginas.

Puedes leer los capítulos de forma selectiva. Si tu dolor de hoy es una dead-letter queue que no deja de crecer, ve directo al Capítulo 8 y sigue las referencias hacia los capítulos 6 y 7 según haga falta. Si estás diseñando un servicio nuevo, yo leería la Parte II antes de escribir la primera línea de código de publicación, porque es mucho más fácil empezar con el out-box que incorporarlo después.

Conoce Parcelly

A lo largo del libro, un sistema ficticio carga con los ejemplos: **Parcelly**, un marketplace en línea donde quienes venden de forma independiente publican productos y quienes compran los piden. Tiene seis servicios.

- **Orders** es responsable del ciclo de vida del pedido y es donde empiezan la mayoría de los flujos.
- **Payments** autoriza y captura el dinero, y habla con un proveedor de pagos externo.
- **Inventory** lleva el control del stock y mantiene reservas temporales.
- **Shipping** programa los envíos y entrega los paquetes a los transportistas.
- **Notifications** envía correo electrónico, mensajes push y SMS, y nunca debe bloquear nada más.
- **Ledger** mantiene un registro de partida doble, de solo adición, de los movimientos de dinero.

Cada servicio es propietario de su propia base de datos relacional y no lee la de nadie más. Se comunican a través de un message broker intercambiando eventos con nombres como `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` y `ShipmentScheduled`. Los eventos siguen una forma consistente, un envelope que lleva un identificador, un tipo, una versión de esquema, campos de correlación y un payload, de modo que los capítulos puedan hablar de ellos con una sola voz.

Elegí un marketplace a propósito. Cobrar dinero, reservar mercancía y mover paquetes es un dominio donde el costo de un mensaje perdido o de un efecto duplicado se siente enseguida. Un correo duplicado es una molestia. Una captura duplicada es un reembolso, un ticket de soporte y una disculpa. Una reserva perdida es un artículo vendido de más. Ese gradiente, de lo inofensivo a lo caro, es exactamente lo que necesitas para decidir cuánta maquinaria de confiabilidad merece cada flujo.

Parcellly es ficción, y también lo son sus incidentes. Los escenarios que abren cada capítulo están inventados para ilustrar la forma de un fallo, y siempre se presentan como ficción. Usaré la primera persona para los juicios, como “yo añadiría aquí una restricción única” o “en mi opinión esto no vale la complejidad”, pero no disfrazaré razonamientos de historias de guerra. Debes poder confiar en que, cuando el libro dice que algo ocurre, es por cómo funcionan estos sistemas y no por una anécdota.

El libro es también deliberadamente **agnóstico respecto al broker**. Cuando un punto depende de si tu broker está basado en logs, con consumidores que llevan un offset en una partición ordenada, o basado en colas, con mensajes que se confirman y se reentregan uno a uno, diré cuál es el caso. Cuando no importa, no fingiré lo contrario.

Cómo leer el código

El código de este libro es **pseudocódigo**. No pertenece a ningún lenguaje, no compilará y nada de él llama a una biblioteca real. Es intencional. El objetivo de cada fragmento es mostrar dónde empieza y termina una transacción, dónde se confirma un mensaje y qué pasa cuando un paso falla a medias. Esos hechos son los mismos en todos los lenguajes, y la sintaxis solo estorbaría.

Los bloques son cortos, de 30 líneas como máximo. Cuando un esquema de tabla ayuda, se escribe en una notación parecida a SQL. Los diagramas usan Mermaid, y cada uno se presenta con una frase que te dice qué mirar. Si traduces un patrón a tu propio stack, trata el pseudocódigo como una especificación de orden y atomicidad, no como una implementación.

Una nota sobre la certeza

En este libro haré recomendaciones, y sostendré algunas con firmeza. No publicaría eventos con un dual write simple, por ejemplo, cuando la pérdida de un evento importa para el negocio. Otras recomendaciones son cuestión de gusto o de contexto, e intentaré decir cuál es cuál.

También habrá lugares donde la respuesta honesta es “depende, y aquí te digo de qué”. Los retry budgets, la retención de las dead letters, el número de particiones y el nivel adecuado de orden dependen de hechos sobre tu negocio y tu equipo que no puedo ver. En esos lugares nombraré las variables en lugar de inventar un número. Si terminas un capítulo sabiendo qué preguntas hacer en tu propio sistema, lo contaré como un éxito.

El trabajo de confiabilidad tiene una propiedad desafortunada: cuando tiene éxito, no ocurre nada visible. Los pedidos se cumplen, los libros contables cuadran y nadie te escribe para darte las gracias. Espero que este libro te ayude a construir esa clase de quietud.

Empecemos por el happy path y por las razones para no confiar en él.

PARTE I

Fundamentos

Capítulo 1 — El engaño del happy path

Imagina Parcelly en su primer año. Orders llama a Payments por HTTP, espera la respuesta, luego llama a Inventory, espera otra vez y después llama a Shipping. El código se lee de arriba abajo, un fallo es una excepción, y quien ve una página de error puede volver a intentarlo. Es lento cuando el proveedor de pagos es lento, y se cae cuando Inventory está caído, pero nadie está confundido sobre lo que pasó.

Entonces el equipo decide pasar a eventos. Orders publicará `OrderPlaced` en un message broker, y los demás servicios reaccionarán a su propio ritmo. La demo es preciosa. Alguien hace clic en “Hacer pedido” y, en un segundo, el stock queda reservado, la tarjeta se captura, el pedido se confirma y se programa un envío. Todo el mundo aplaude.

Ahora imagina el primer mes en producción. A una persona se le cobra, pero su pedido se queda en “pending” para siempre. Otro pedido se envía dos veces. Un tercero tiene un asiento en el libro mayor sin el pago correspondiente, y nadie sabe decir qué servicio tiene razón. Soporte pregunta: “¿qué pasó con el pedido 81724?”, y la respuesta honesta es que cuatro servicios conocen cada uno un pedazo y ninguno lo conoce entero.

Nada en la demo estaba mal. Estaba incompleta, porque una demo es una historia en la que todos los pasos salen bien. La pregunta que responde este capítulo es qué cedió el equipo al abandonar la solicitud y respuesta, y qué tiene que significar “confiable” antes de que alguien empiece a reconstruirlo.

El problema y por qué es difícil

El happy path es la secuencia de eventos que ocurre cuando todos los componentes están activos, todas las llamadas de red llegan, todos los mensajes se entregan una sola vez y en orden, y todos los consumidores

terminan lo que empiezan. Es el camino que dibujas en una pizarra y el que ejercita una demo. Es también el camino más raro en un sistema que lleva meses funcionando.

Los fallos no son exóticos. Los deploys reinician procesos. Las conexiones se caen en mitad de una escritura. Un broker conmuta a una réplica. Un consumidor tarda más de lo esperado y su mensaje se entrega a otro. Cada una de estas cosas ocurre en algún lugar de un sistema ocupado todos los días. La forma habitual de un mal mes en producción es un puñado de eventos corrientes que se encuentran con código escrito para el happy path.

Lo que lo hace difícil es que los sistemas orientados a eventos eliminan lo que hacía visibles los fallos. En una llamada síncrona, quien llama espera, y el resultado vuelve por la misma conexión: un resultado o un error, junto al código que lo causó. En un diseño orientado a eventos, el productor entrega un mensaje al broker y sigue adelante. Si algo sale mal tres servicios y diez segundos después, el productor ya no está, y la evidencia queda repartida en logs que nadie ha correlacionado todavía.

La respuesta obvia es volver confiable cada pieza: un buen broker, servicios bien probados, retries en cada llamada. Cada medida ayuda y ninguna basta, porque los fallos que duelen viven en los huecos entre los componentes. Un mensaje escrito en la base de datos pero nunca publicado, o procesado pero nunca confirmado, no es un bug de ninguno de los dos componentes. Es una propiedad de la unión. Gran parte de este libro trata de uniones.

Para cada unión, el libro hace las mismas tres preguntas: qué puede fallar, cómo lo observarías y qué garantía sobrevive. Llamaré **semántica de fallos** de la unión a las respuestas. La documentación de un broker te da la semántica de las uniones del propio broker. Las uniones que lo rodean, entre tu base de datos y tu llamada de publicación o entre una entrega y su efecto, te toca resolverlas a ti, y el Capítulo 3 muestra cómo.

Acoplamiento síncrono y asíncrono

El paso a eventos suele describirse como “desacoplamiento”, una palabra que promete más de lo que cumple. Es más exacto decir que el acoplamiento cambia de forma.

Una llamada síncrona acopla dos servicios en el **tiempo**, en la **disponibilidad** y en el **conocimiento**. El servicio llamado tiene que estar en marcha ahora, la latencia de quien llama incluye la del llamado, y quien llama sabe quién es el llamado y qué va a responder. La ventaja es un modelo de fallos simple: quien llama sabe que la llamada falló y puede decidir de inmediato si reintenta, usa una alternativa o informa de un error. El costo es que la cadena solo es tan disponible como su eslabón más débil. Una API lenta del transportista se convierte en un checkout lento.

Un mensaje a través de un broker elimina el acoplamiento en el tiempo y en la disponibilidad. Orders puede aceptar un pedido mientras Inventory se reinicia, porque el broker retiene el mensaje hasta que Inventory vuelve. Un consumidor nuevo, como un servicio de analítica, puede empezar a escuchar sin que Orders cambie. Esa ganancia es real, y es la razón para adoptar el estilo.

Pero el acoplamiento se ha movido, no ha desaparecido. Ahora los servicios están acoplados por el **contrato del evento**: su nombre, sus campos y su significado. Están acoplados por el **comportamiento del broker**: lo que promete sobre durabilidad, orden y entrega. Y están acoplados por suposiciones sobre los tiempos de los demás que nadie dejó escritas. El Capítulo 11 y el Capítulo 12 tratan el contrato, y el Capítulo 2 trata el broker.

En resumen, los sistemas síncronos fallan con estruendo y en cadena. Los sistemas asíncronos fallan en silencio y por piezas.

El precio del desacoplamiento temporal

El desacoplamiento temporal significa que el productor y el consumidor no tienen que estar activos al mismo tiempo. Es lo principal que compras con los eventos, así que conviene ser exactos sobre el precio.

El productor ya no conoce el resultado

Cuando Orders llamaba a Payments de forma síncrona, sabía si el pago había funcionado antes de responder a quien compraba. Con eventos, Orders publica `OrderPlaced` y responde “recibido”. Si el pago se aprueba es una pregunta para más tarde, y la respuesta llega, si llega, como otro evento.

El pedido tiene ahora un estado intermedio, “realizado pero aún no confirmado”, y el sistema tiene que decidir qué ve la persona mientras dura y qué pasa si dura más de lo previsto.

Nadie es responsable del flujo completo

En la versión síncrona, una función contenía la secuencia, y podías leer el proceso de negocio en un solo lugar. En la versión orientada a eventos, el proceso es la suma de reacciones repartidas entre servicios. Payments reacciona a `OrderPlaced`, Orders reacciona a `PaymentAuthorized` y `StockReserved`, Shipping reacciona a `OrderConfirmed`. El flujo existe solo como propiedad emergente, y nadie puede señalar dónde está definido.

Eso no importa cuando todo funciona y sale caro cuando algo no funciona, porque “¿dónde se atascó este pedido?” no tiene un único lugar donde mirar. El Capítulo 10 vuelve sobre esto con las sagas, y el Capítulo 13 cubre lo que necesitas observar para poder responder a la pregunta siquiera.

Los fallos dejan de ser excepciones

En una cadena de llamadas, un error se propaga por la pila hasta que algo lo maneja. En un flujo de mensajes no hay pila. Un consumidor que falla puede confirmar el mensaje y perderlo, rechazarlo y que se reentregue, o apartarlo para que lo vea una persona. Cada opción es una decisión de política con consecuencias para servicios que nunca ven el error. En mi opinión, el manejo de errores en un sistema síncrono es cuestión de flujo de control, y en un sistema asíncrono es cuestión de diseño, porque el manejo se reparte entre tiempo, procesos y a veces equipos.

El tiempo se vuelve una variable

Entre “publicado” y “procesado” hay un intervalo sin límite superior a menos que le pongas uno. Durante ese intervalo un producto puede agotarse, alguien puede cancelar, un precio puede cambiar. Un consumidor debe decidir si un evento que era cierto cuando se publicó sigue teniendo sentido cuando se procesa.

Nota: Por eso los eventos se nombran en pasado. `OrderPlaced` es un hecho sobre un momento que ya pasó. Un consumidor que lo trata como una instrucción para actuar ahora mismo ha supuesto, sin decirlo, que no pasó el tiempo. Un comando, como `CapturePayment`, le pide a un servicio que haga algo y puede ser rechazado; un evento informa de algo que ya ocurrió. El Capítulo 11 desarrolla la distinción.

Qué significa realmente “al final”

La mayoría de las explicaciones de la **consistencia eventual** caben en una frase: si no se hacen nuevas actualizaciones, todas las réplicas convergen al mismo valor. Es una frase exacta y casi inútil para quien construye un flujo de checkout, porque deja fuera cuánto dura ese “al final” y qué hacer mientras esperas.

Al final no es una duración

Un sistema que converge en 200 milisegundos y un sistema que converge después de un batch nocturno tienen ambos consistencia eventual. Para quien mira una página de confirmación, son productos distintos. Pediría un número para cada salto que importe, aunque sea aproximado. Digamos que `Orders` debe mostrar un pedido confirmado a los pocos segundos de la captura, y que `Notifications` puede tardar un minuto. Estas cifras son ilustraciones y las tuyas serán otras, pero convierten el “al final” en un requisito que se puede probar y sobre el que se pueden configurar alertas. Un retraso sin límite no es una decisión de diseño. Es la ausencia de una.

Al final no es una garantía

La convergencia ocurre solo si todo mensaje que importa se entrega al final, todo consumidor lo procesa y ningún paso lo descarta en silencio. Si un mensaje se pierde entre un commit en la base de datos y una publicación en el broker, las vistas no convergen. Siguen siendo distintas hasta que una persona o un trabajo de reparación lo nota. El “al final” es una promesa condicional, y la condición es que el sistema tenga mecanismos

que hagan la pérdida detectable y reparable. Elegir un broker no los proporciona. Las partes II y III de este libro previenen la mayor parte de las pérdidas, y el Capítulo 13 detecta y repara el resto.

Qué hacer mientras esperas

Durante la ventana de inconsistencia, otros lectores ven un estado intermedio. Considera esta instantánea de un pedido, un momento después de que se cobrara a quien compró:

```
// PaymentCaptured not yet consumed
orders.status      = "CAPTURE_REQUESTED"
payments.status    = "CAPTURED"
inventory.state    = "RESERVED"
shipping.shipment  = none
// LedgerEntryPosted not yet emitted
ledger.entries     = none
```

Cada servicio tiene razón sobre sus propios datos. La combinación parece inconsistente y es normal durante unos segundos. Solo se convierte en un problema cuando se queda así. La pregunta operativa útil no es “¿son consistentes los servicios ahora mismo?”, sino “¿cuánto tiempo lleva este pedido en este estado, y es más de lo que permitimos?”.

Una definición práctica de confiabilidad

“Confiable” se usa de forma tan laxa que dos personas de ingeniería pueden estar de acuerdo en que un sistema debe ser confiable y querer decir cosas opuestas. Quiero una definición que se pueda comprobar. Para los sistemas orientados a eventos uso cuatro propiedades. Dicen qué se está protegiendo; las tres preguntas anteriores son la manera de comprobar una unión frente a ellas.

Sin pérdida. Todo evento que representa un hecho de negocio llega a todo consumidor que lo necesita. La pérdida es el fallo más dañino porque es silenciosa: el sistema sigue funcionando, y el único síntoma es una ausencia que nadie nota hasta que alguien se queja. La ausencia de pérdida es una propiedad del camino completo, no del broker. Un broker durable no sirve de nada si el productor nunca logró almacenar el mensaje, o si el

consumidor confirmó antes de terminar el trabajo. El Capítulo 2 examina lo que prometen los brokers, y el Capítulo 4 y el Capítulo 5 abordan el lado del productor.

Sin duplicación dañina. Algunos mensajes se entregarán más de una vez. No es un defecto de un broker en particular sino el precio de no perder mensajes: cuando quien envía no puede saber si un mensaje llegó, lo seguro es enviarlo otra vez. Un sistema confiable no intenta evitar los duplicados en el transporte. Los vuelve inofensivos. La palabra “dañina” importa. Dos correos idénticos son una molestia, mientras que una segunda captura del mismo pago es un incidente financiero. El Capítulo 7 cubre la idempotencia, la herramienta principal.

Retraso acotado. Todo evento se procesa dentro de un tiempo que el negocio tolera, o el sistema dice a gritos que no fue así. Un mensaje entregado correctamente con tres días de retraso es, para muchos fines, un mensaje perdido. Acotado no significa rápido. Significa que existe un límite, que alguien lo eligió y que su incumplimiento genera una alerta. Una confirmación de pago puede necesitar segundos; un extracto semanal puede tolerar horas. El Capítulo 6 trata los retries y el backoff, y el Capítulo 13 trata el retraso y las alertas.

Explicabilidad. Para cualquier pedido, en cualquier momento, alguien puede reconstruir qué pasó: qué eventos se produjeron y se consumieron, qué decidió cada servicio y por qué. Es la propiedad menos glamurosa y decide si las otras tres pueden mantenerse. Un sistema que pierde un mensaje pero puede mostrar exactamente dónde es reparable. Un sistema que se comporta mal de una forma que nadie puede rastrear no lo es. El Capítulo 13 cubre los identificadores, registros y comparaciones que esto requiere.

Las cuatro tiran en sentidos opuestos, y por eso una definición con solo una de ellas es peligrosa. Perseguir la ausencia de pérdida con retries ilimitados amenaza el retraso acotado. Descartar los mensajes lentos para acotar el retraso amenaza la ausencia de pérdida. Negarse a reintentar para evitar duplicados trae de vuelta la pérdida. La confiabilidad es un equilibrio deliberado, elegido por flujo y dejado por escrito.

Por qué el happy path falla en producción

Con la definición en la mano, vuelve al primer mes de Parcelly y clasifica los fallos por propiedad. Unos pocos mecanismos explican la mayor parte del daño.

Lo que vio Parcelly	Propiedad en riesgo	Causa de fondo	Capítulo
Pedido aceptado, pero nunca ocurre nada aguas abajo	Sin pérdida	Evento confirmado en una base de datos pero nunca publicado	4
Pedido enviado dos veces	Sin duplicación dañina	Mensaje reentregado y procesado de nuevo	7
Un mensaje defectuoso bloquea una partición	Retraso acotado	Un mensaje falla siempre y se re-intenta indefinidamente	8
Pedido “confirmado” antes de “realizado”	Explicabilidad	Eventos relacionados procesados fuera de orden	9
El libro mayor no coincide con los pagos	Explicabilidad	Sin identificador compartido, sin forma de comparar las vistas	13

La cuarta fila es la incómoda. El reordenamiento no duplica nada. Daña el estado, por ejemplo cuando una actualización obsoleta sobrescribe una más reciente, y ninguna de las cuatro propiedades lo nombra directamente. Lo archivo bajo la explicabilidad porque el síntoma es un historial que nadie puede explicar, y el Capítulo 9 se ocupa del daño al estado en sí.

La primera fila merece un recorrido detallado, porque es el fallo que con más frecuencia pone fin a la luna de miel.

```
function placeOrder(request):
  transaction:
    orderId = insertOrder(request)
    // commit happens here; the order now exists in the database

    // the process can crash right here, between the two steps
    publish("orders", OrderPlaced(orderId))
```

Este bloque no es seguro, y se muestra como la línea base rota. El commit en la base de datos y la publicación en el broker son dos sistemas sin una transacción compartida, lo que el libro llama **dual write**. Si el proceso se detiene entre las dos líneas, el pedido existe y nadie se entera. Si se invier-

ten las líneas, puede publicarse un evento para un pedido que nunca se confirmó. Ninguno de los dos órdenes es correcto, y ambos pasan todas las pruebas que no matan el proceso justo en el peor momento.

No es un bug raro. Es el comportamiento por defecto de un diseño que nunca tuvo un modelo de fallos. El Capítulo 3 da el modelo, y el Capítulo 4 da la solución.

Un mapa del libro

El resto del libro responde a las preguntas que plantea la definición, en el orden en que un equipo suele encontrárselas.

Pregunta que te harás	Capítulo
¿Qué promete realmente el broker y qué no?	2
¿Qué puede romperse y qué verá cuando ocurra?	3
¿Cómo cambio mi base de datos y publico un evento sin perder ninguno?	4
¿Cómo llegan al broker los eventos publicados desde la base de datos?	5
¿Cómo hago retries sin empeorar las cosas?	6
¿Cómo vuelvo inofensivo el procesamiento repetido?	7
¿Qué hago con un mensaje que nunca puede tener éxito?	8
¿Cuándo importa el orden y cómo lo mantengo?	9
¿Cómo coordino trabajo entre servicios y cómo lo deshago?	10
¿Qué debe contener un evento y cómo debe llamarse?	11
¿Cómo cambio un evento sin romper a los consumidores?	12
¿Cómo veo, mido y reparo lo que está ocurriendo?	13
¿Cómo me recupero de datos perdidos, de una región perdida o de un cambio de broker?	14
¿Cómo pruebo los caminos de fallo?	15
¿Qué compruebo antes de salir a producción?	16

La Parte I, que abre este capítulo, construye el vocabulario y el modelo de fallos. La Parte II cubre cómo publicar con confiabilidad, la Parte III cómo consumir con confiabilidad, la Parte IV los flujos de trabajo y contratos, y

la Parte V cómo operar y demostrar el sistema. Lee las partes en orden en una primera pasada. Si hoy peleas con un problema concreto, la tabla es un índice.

Trade-offs y costos

Pasar a eventos no es un error, y este capítulo no es un argumento en contra. Es un argumento a favor de dar el paso con los ojos abiertos, porque el costo es real y se paga en lugares concretos.

Pagas en complejidad de estado. El flujo síncrono tenía dos resultados: hecho o fallido. El flujo orientado a eventos tiene muchos estados intermedios, y cada uno hay que nombrarlo, mostrarlo a los usuarios, darle seguimiento y resolverlo al final. Alguien tiene que ser responsable de esa máquina de estados.

Pagas en habilidad operativa. Depurar una cadena de llamadas requiere un stack trace. Depurar un flujo de mensajes requiere correlation ids, dashboards de retraso y la capacidad de razonar sobre tiempos. Un equipo que no los tiene descubre que el tiempo para entender un incidente crece incluso si los incidentes se vuelven menos frecuentes. El costo recae sobre quien está on-call.

Pagas en esfuerzo de pruebas. El happy path exige poco esfuerzo de prueba. Los caminos de fallo necesitan mensajes duplicados, mensajes reordenados y reinicios en momentos incómodos. El Capítulo 15 muestra cómo hacerlo sin una suite lenta e inestable, pero el esfuerzo no es cero.

Pagas en latencia por cosas simples. Una operación que era una llamada a una función se convierte en varios saltos a través de un broker. Cuando quien llama necesita la respuesta ahora, como al validar un código de descuento antes de mostrar un precio, una llamada síncrona es la mejor herramienta.

No usaría eventos cuando dos pasos deben tener éxito o fallar juntos y viven en el mismo servicio y la misma base de datos. Una transacción local es más simple, más sólida y más barata que cualquier patrón de este libro. También dudaría cuando el equipo no tiene capacidad para operar un broker y la supervisión que lo rodea, o cuando el sistema es lo bastante pequeño como para que una única unidad desplegable con una sola base de

datos haga el trabajo. Muchos buenos sistemas mezclan ambos estilos: una llamada síncrona para una pregunta que necesita respuesta, un evento para un hecho que a otros puede interesar. Elegiría por interacción, no por sistema.

La definición de cuatro propiedades también tiene un límite. Es una lente, no una especificación. No te dice cuál es el límite de retraso adecuado para tu negocio ni qué duplicados son dañinos para un consumidor dado. No conozco tus números, y esas respuestas rara vez salen solo de la ingeniería. Lo que puedo ofrecer es el hábito de pedir las.

Errores comunes

- **Probar solo el happy path.** Es tentador porque la solicitud de la funcionalidad lo describía y pasa rápido. Trata cada flecha del diagrama de flujo como una pregunta sobre fallos, y escribe al menos una prueba por respuesta (Capítulo 15).
- **Tratar al broker como la capa de confiabilidad.** Los brokers anuncian durabilidad, lo que hace tentadora esta idea. Un broker cubre el tramo entre la aceptación y la entrega, y los fallos que más duelen están antes y después. Enumera lo que promete el broker y luego enumera los huecos alrededor de la promesa (Capítulo 2 y Capítulo 3).
- **Usar “al final” como diseño.** Suena a decisión y evita la pregunta más difícil de cuánto tiempo. Asocia un límite y una alerta a cada salto asíncrono que importe, y di qué ve el usuario mientras espera.
- **Suponer que los duplicados son demasiado raros para manejarlos.** En un entorno de pruebas tranquilo nunca aparecen. Diseña los consumidores suponiendo que todo mensaje puede llegar más de una vez (Capítulo 7).
- **Confirmar en la base de datos y publicar en el broker como dos pasos simples.** Se lee con naturalidad y funciona en todas las demos. La solución es el transaccional outbox del Capítulo 4.
- **Definir la confiabilidad como uptime.** El uptime es barato de medir, pero un servicio puede estar activo todo el mes y aun así perder mensajes o procesarlos tarde. Mide directamente las cuatro propiedades: qué se perdió, qué se duplicó, cuánto se retrasaron las cosas y si pudiste explicarlas.

Ideas clave

- El happy path es una historia en la que todos los pasos salen bien, y un sistema probado solo contra él ha demostrado funcionar en el único caso que rara vez se mantiene mucho tiempo.
- Pasar de llamadas síncronas a eventos no elimina el acoplamiento; lo cambia del tiempo y la disponibilidad a los contratos, el comportamiento del broker y las suposiciones de tiempo no escritas.
- Los fallos orientados a eventos son silenciosos y se reparten entre servicios, de modo que un diseño sin modelo de fallos tiende a enterarse de sus uniones por la producción.
- El “al final” es una promesa condicional sin duración asociada, así que todo salto asíncrono que importe necesita un límite, una alerta y un estado definido de cara al usuario para el período de espera.
- Un sistema confiable no tiene pérdida, no tiene duplicación dañina, tiene retraso acotado y es explicable, y como estas propiedades se compensan entre sí, cada flujo necesita un equilibrio deliberado y escrito.
- Las uniones entre componentes, como un commit en la base de datos seguido de una publicación en el broker, causan más daño que los componentes mismos, y la mayor parte de este libro trata de esas uniones, examinadas con tres preguntas: qué puede fallar, cómo lo observarías y qué garantía sobrevive.
- Los eventos no siempre son la herramienta adecuada: cuando los pasos pueden compartir una transacción local, o quien llama necesita una respuesta inmediata, una llamada síncrona es más simple y más sólida.

El siguiente capítulo examina el broker mismo. Antes de poder diseñar alrededor de sus huecos, necesitamos saber exactamente qué promete una cola o un log cuando dice que un mensaje fue entregado, y qué deja discretamente en tus manos.

Capítulo 2 — Semántica de la mensajería

Imagina un viernes por la tarde en Parcelly, en pleno período de rebajas. El servicio Notifications lleva meses enviando correos de “tu pedido está confirmado” sin una sola queja. A medida que crece el backlog, el autoscaling añade consumidores y, unos minutos después, vuelve a quitar algunos. Al final de la tarde, soporte tiene dos tickets. Una persona compradora recibió tres veces el mismo correo de confirmación. Otra no recibió ninguno, aunque su pedido está confirmado.

El equipo abre el dashboard del broker y descubre que nada está roto. Todos los offsets se confirmaron con commit. Todos los consumidores están sanos. Las gráficas son planas. Sin embargo, el mismo sistema produjo un duplicado para una persona y un silencio para otra, y ambos resultados son coherentes con la documentación.

Alguien dice que el broker prometió “at-least-once delivery”, y otra persona dice que había supuesto que el broker se encargaba de esto. Las dos afirmaciones son ciertas, y ninguna explica lo que les pasó a las dos personas compradoras. Supongamos que Notifications lee un topic basado en log como un único grupo de consumidores, con su cliente confirmando los offsets de forma automática, y que se ejecutó un rebalanceo cada vez que el grupo cambió de tamaño. Esos dos hechos pueden producir ambos tickets, y el capítulo muestra cómo.

Para explicarlo, el equipo necesita una imagen precisa de lo que un broker promete, de lo que no promete y de qué partes de la vida del mensaje pertenecen a la aplicación. ¿Qué atraviesa realmente un mensaje entre un productor y un efecto terminado, y dónde empieza y termina cada garantía?

El problema y por qué es difícil

La mayoría de los ingenieros conocen un broker a través de su descripción más simple: publicas un mensaje y un consumidor lo recibe. Esa descripción oculta tres traspasos distintos, y cada pregunta de confiabilidad de este libro vive en uno de ellos.

El primer traspaso va del productor al broker. El segundo, del broker al consumidor. El tercero, del consumidor al efecto: la escritura en la base de datos, el correo, la llamada a un transportista. Un broker tiene mucho que decir sobre los dos primeros y casi nada sobre el tercero, porque el tercero ocurre en código y en sistemas que el broker no puede ver.

La dificultad es que cada traspaso termina con un ack, y un ack es una afirmación sobre el pasado que una parte hace a otra a través de una red. Dice “recibí esto” o “terminé esto”. No dice “y nada más puede salir mal”. Cuando se pierde un ack, quien envía no puede saber si el trabajo se hizo. Esa incertidumbre es la raíz de las tres semánticas de entrega, y volveremos a ella cuando tengamos el vocabulario.

Una segunda dificultad es que la palabra “broker” abarca dos familias que se comportan de maneras distintas, y esa diferencia importa. Una **cola** es una estructura del broker donde cada mensaje se entrega a un consumidor y se confirma de forma individual. Un **log de eventos** es una secuencia duradera de eventos en la que solo se añade, donde los consumidores leen a su propio ritmo y registran una posición. Lo que es cierto de una familia suele ser falso de la otra. El libro nombra la familia siempre que la distinción importa, y este capítulo es donde se construye esa distinción.

Una tercera dificultad es que las garantías de la documentación de un producto tienen un alcance acotado. “Entregado al menos una vez” es una afirmación sobre el límite del propio broker. No cubre lo que hace el consumidor después de recibir el mensaje, y depende de ajustes que el productor eligió al publicar. Leer la garantía sin su alcance es la manera de terminar en el viernes por la tarde de arriba.

Colas y logs

El modelo de cola

En una cola, el broker lleva el estado de cada mensaje. Un mensaje está esperando, entregado a un consumidor, o confirmado y desaparecido. Varios consumidores pueden leer de la misma cola; este arreglo se llama **consumidores competidores**, porque cada mensaje se entrega a uno de ellos a la vez y compiten por el siguiente.

Cuando un consumidor confirma un mensaje, el broker lo elimina. La retención es por tanto corta por diseño: un mensaje vive hasta que se procesa o hasta que expira. No hay una forma integrada de pedirle a la cola un mensaje que ya se confirmó, porque la cola no lo conserva.

La consecuencia es que una cola encaja bien con el trabajo que debe hacerse una vez y olvidarse. Un job de Parcelly como “generar el PDF de esta factura” es un candidato natural: se hace el trabajo y el mensaje ya no sirve para nada. El costo es que no hay historial disponible. Si un bug envía el texto equivocado durante una hora, los mensajes que llevaban los hechos ya no existen, y la reparación tiene que partir de otro lugar.

El modelo de log

En un log de eventos, el broker no lleva un estado por mensaje. Guarda los mensajes en orden, en particiones, y los conserva según una política de retención independiente de quién los haya leído. Una **partición** es una subsecuencia ordenada de un topic, y un **topic** es un stream de mensajes con nombre. Cada mensaje tiene un **offset**, su posición en la partición.

Los consumidores llevan la cuenta de hasta dónde han leído. Esa posición también se llama offset, y la registra el consumidor, o el broker en su nombre. Un **grupo de consumidores** es un conjunto de consumidores que se reparten el trabajo de un topic: cada partición se asigna a un consumidor del grupo a la vez. Grupos distintos leen el mismo topic de forma independiente, cada uno con sus propios offsets.

La propiedad que define al log es que leer no consume. Diez grupos pueden leer el mismo topic, y un grupo nuevo puede empezar desde el principio. **Replay**, la relectura deliberada del historial, es el término que este libro reserva para ese acto, y solo es posible porque el log conserva los da-

tos. El Ledger de Parcelly es un buen ejemplo de consumidor que se beneficia: si se descubre un bug en la forma de construir las entradas, se puede corregir el servicio Ledger y apuntarlo de nuevo a los eventos antiguos.

El costo es que el log traslada la contabilidad al lado del consumidor. En el modelo clásico de log, el broker no puede reentregar un mensaje fallido mientras avanza al siguiente, porque no hace seguimiento individual de los mensajes. Un consumidor que no puede procesar el mensaje del offset 500 tiene que decidir qué hacer antes de poder pasar con seguridad al 501.

Cómo elegir entre ellos

Aspecto	Familia de colas	Familia de logs
Unidad de seguimiento	Cada mensaje	Una posición por partición
Retención	Hasta que se confirma o expira	Por tiempo o tamaño, independiente de los lectores
Varios lectores independientes	Requiere una copia del mensaje por lector	Natural, una posición por grupo
Replay del historial	No disponible	Disponible dentro de la retención
Un mensaje que falla	Se reentrega por sí solo, los demás continúan	Bloquea la partición si no se maneja
Orden	FIFO en el almacenamiento, como mucho; se pierde entre consumidores competidores y con la reentrega	Fuerte dentro de una partición

No elegiría una familia por la popularidad de la herramienta. Partiría de tres preguntas: ¿alguien necesita volver a leer este historial?, ¿varios consumidores necesitan los mismos hechos? y ¿importa el orden para un mismo agregado? Un sí a cualquiera de ellas apunta hacia un log. Una cola de trabajo pura, sin necesidad de historial y con necesidad de retries independientes para cada mensaje, apunta hacia una cola.

Muchos equipos acaban operando ambas. Los eventos a partir de los cuales otros servicios construyen estado van a un log, y las tareas que deben ocurrir una sola vez van a una cola. Es un diseño legítimo, y viene con el

costo de operar y entender dos conjuntos de semántica. El Capítulo 9 vuelve sobre lo que significa el orden en cada familia.

Push, pull y el flujo de la entrega

Los brokers también difieren en quién inicia la entrega. En un modelo **push**, el broker envía los mensajes a los consumidores a medida que están disponibles. En un modelo **pull**, el consumidor pide mensajes cuando está listo.

Los brokers basados en colas suelen hacer push (RabbitMQ entrega a los consumidores, mientras que los consumidores de Amazon SQS hacen polling, así que incluso esto varía dentro de la familia), con un límite de cuántos mensajes sin confirmar puede tener un consumidor a la vez. Ese límite suele llamarse prefetch count o ventana. Los brokers basados en log suelen hacer pull: el consumidor pide el siguiente lote desde su offset actual. Ambos modelos pueden encontrarse en cualquiera de las dos familias, así que lo trataría como una tendencia y no como una ley.

La diferencia importa sobre todo por lo que ocurre cuando un consumidor es lento. En un modelo push sin límite, el broker sigue enviando, la memoria del consumidor se llena de mensajes en los que aún no ha empezado a trabajar, y esos mensajes quedan en un estado en el que ni están disponibles para otro consumidor ni están terminados. En un modelo pull, el consumidor controla su propio ritmo, y un consumidor lento se manifiesta como un retraso del consumidor creciente, no como un proceso sobrecargado. El **retraso del consumidor** es cuánto va un consumidor por detrás de la cabeza de una partición o de una cola, medido en mensajes o en tiempo.

La regla práctica es que un consumidor nunca debe tener más trabajo del que pueda terminar antes de que expire su reclamación sobre ese trabajo. El Capítulo 3 trata los consumidores lentos y el backpressure como fallos por derecho propio, y el Capítulo 6 explica cómo acotar la carga. Por ahora, recuerda que “entregado” y “en procesamiento” son estados distintos, y que la brecha entre ellos es donde pasa el tiempo.

Ack, reentrega y offsets

El ack en una cola

Un **acknowledgment**, que suele abreviarse como **ack**, es la declaración del consumidor de que terminó con un mensaje y el broker puede olvidarlo. Un **nack**, un ack negativo, dice que el consumidor no pudo procesar el mensaje. Según el broker y los ajustes, un nack devuelve el mensaje a la cola, lo descarta o lo envía a una dead-letter queue (DLQ).

La pregunta clave de diseño es cuándo enviar el ack. Solo hay dos posiciones razonables.

- **Ack antes de procesar.** El broker olvida el mensaje de inmediato. Si el consumidor se cae a mitad del trabajo, el mensaje se pierde. Esto produce comportamiento at-most-once.
- **Ack después de procesar.** El broker retiene el mensaje hasta que el trabajo termina. Si el consumidor se cae después de terminar pero antes de enviar el ack, el mensaje vuelve y el trabajo se hace dos veces. Esto produce comportamiento at-least-once.

No existe una posición que dé ambas propiedades, porque el ack y el trabajo son dos acciones separadas sobre dos sistemas separados, y una caída puede ocurrir entre ellas.

Visibility timeout y reentrega

¿Cómo decide una cola que un consumidor se ha caído? No puede observarlo directamente. Algunos brokers vigilan la conexión y tratan una conexión cerrada como un fallo. Otros usan un **visibility timeout**: cuando se entrega un mensaje, queda invisible para los demás consumidores durante un tiempo fijado. Si el consumidor envía el ack antes de que se agote el tiempo, el mensaje se elimina. Si no, el mensaje vuelve a ser visible y se entrega a otro. Esa segunda entrega es una **reentrega**.

El timeout es una conjetura sobre cuánto tarda el procesamiento, y equivocarse en cualquiera de los dos sentidos cuesta algo. Si es demasiado corto, a un consumidor lento pero vivo le entregarán su mensaje a un segundo consumidor mientras todavía está trabajando. Ahora dos consumidores procesan el mismo mensaje, y el ack del primero puede fallar, ser ignorado o, según el broker, tener éxito y eliminar el mensaje mientras el segundo

consumidor sigue trabajando en él. En este último caso, un fallo del segundo intento pierde el mensaje. Si es demasiado largo, una caída real deja el mensaje oculto durante todo ese período, y la demora se manifiesta como latencia para la persona compradora.

Digamos que el consumidor de Inventory de Parcelly normalmente termina una reserva en bastante menos de un segundo, y alguien fija el timeout en 30 segundos. Esa cifra es ilustrativa. Un contratiempo de la base de datos que haga que una reserva tarde 40 segundos producirá ahora una segunda entrega del mismo `OrderPlaced`, e Inventory intentará reservar el stock dos veces. Nada está roto en el broker. El timeout codificaba una suposición, y la suposición dejó de cumplirse.

Algunos sistemas permiten que un consumidor extienda el timeout mientras trabaja. Eso ayuda con los jobs legítimamente largos, y añade algo nuevo que puede salir mal: un consumidor que extiende el plazo para siempre sobre un mensaje que nunca terminará.

Offsets en un log

En un log no hay ack por mensaje. En su lugar, el consumidor hace un **commit de offset**: registra “he procesado todo lo anterior a esta posición”. Tras un reinicio o una reasignación, el siguiente consumidor lee desde el offset confirmado.

Aquí existen las mismas dos posiciones. Si haces el commit del offset antes de procesar, una caída pierde los mensajes posteriores al último completado, hasta la posición confirmada. Si lo haces después de procesar, una caída repite los mensajes entre el último commit y la caída. Los consumidores reales hacen commit por lotes, así que el número de mensajes repetidos tras una caída es un lote, no uno.

Muchas bibliotecas cliente pueden hacer el commit automáticamente. Es cómodo y también es la forma menos explícita de fijar tu semántica de entrega, porque, según el cliente, el commit puede ocurrir en un momento sin relación con el de la conclusión de tu procesamiento. Si el procesamiento se delega a otro hilo, el cliente puede confirmar trabajo que nunca se completó. En mi opinión, un consumidor que importa debe hacer el commit de forma explícita, en un punto que el equipo pueda señalar en el código y sobre el que pueda razonar.

Grupos de consumidores y rebalanceo

Cuando los consumidores entran o salen de un grupo de consumidores, el broker redistribuye las particiones entre ellos. Esto se llama **rebalanceo**. Durante un rebalanceo, una partición que leía un consumidor se reasigna a otro, y el nuevo propietario empieza en el último offset confirmado.

Los grupos basados en log hacen, de otra forma, la misma conjetura que el visibility timeout en una cola. Un consumidor que no vuelve al broker dentro de un intervalo configurado se presume muerto, sus particiones se reasignan y su trabajo sin confirmar lo vuelve a leer el nuevo propietario. Un consumidor lento que sigue vivo es la causa cotidiana de los duplicados por rebalanceo.

De aquí vienen los duplicados del escenario inicial. Considera un consumidor que ha procesado los mensajes del 100 al 150 pero solo ha hecho commit hasta el 120. Un deploy lo reinicia y la partición pasa a un segundo consumidor, que lee desde el 120. Los mensajes del 120 al 150 se procesan dos veces, y ninguna parte del sistema funcionó mal. El rebalanceo es un evento normal que puede ocurrir en cada deploy, escalado hacia arriba y caída, así que la reentrega es una ocurrencia cotidiana, no un accidente raro.

El silencio tiene la causa especular. Si el cliente confirmó offsets de mensajes que el consumidor solo había entregado a un emisor en segundo plano, y el proceso se detuvo antes de que el emisor terminara, el siguiente propietario empieza después de esos mensajes. Nunca se vuelven a leer, y ninguna línea de log lo dice. El mismo rebalanceo puede, por tanto, repetir el mensaje de una persona compradora y perder el de otra.

Hay un riesgo relacionado: durante un breve lapso, el antiguo propietario puede no saber aún que perdió la partición y seguir trabajando en ella. Entonces dos consumidores pueden procesar la misma partición a la vez. Los clientes bien diseñados intentan evitarlo, pero yo no diseñaría un consumidor que dependa de que esa prevención sea perfecta.

Semánticas de entrega

Con la mecánica ya en su sitio, las tres semánticas de entrega son consecuencias de dónde pones el ack.

At-most-once

At-most-once significa que cada mensaje se entrega cero o una vez. Se logra enviando el ack antes de procesar, o no reintentando una publicación fallida. Un mensaje puede perderse y nunca puede duplicarse.

Es una elección razonable cuando perder un mensaje cuesta menos que manejar duplicados. Un ping periódico de “ubicación actual” de un transportista de Parcelly es un caso: el siguiente ping vuelve irrelevante al perdido. Es la elección equivocada para `PaymentCaptured`, donde un mensaje perdido significa que el dinero se movió y nadie lo registró.

At-least-once

At-least-once significa que cada mensaje se entrega una o más veces. Se logra enviando el ack después de procesar y reintentando hasta que llegue un ack. Un mensaje nunca se pierde mientras el broker lo retiene, y puede entregarse de nuevo.

Esta es la semántica sobre la que se construye el libro. Es la única de las tres en la que el fallo es visible y recuperable: un duplicado es algo que puedes detectar e ignorar, y un mensaje perdido es algo que quizá no descubras en días. El precio es que cada consumidor debe tolerar un **duplicado**, es decir, un mensaje procesado o entregado más de una vez. El Capítulo 7 trata de cómo.

“Exactly-once”

La entrega es un acto físico a través de una red poco confiable, y el ack del receptor puede perderse como cualquier otro mensaje. Un emisor que no recibió un ack tiene dos opciones: enviar de nuevo, con el riesgo de un duplicado, o no hacerlo, con el riesgo de una pérdida. Ningún protocolo elimina esta disyuntiva para las dos partes por sí solas.

Aun así, exactly-once es un resultado real y útil, y es un efecto, no una garantía de entrega. Cuando la entrega at-least-once se encuentra con un consumidor cuyo procesamiento repetido tiene el mismo efecto que procesar una sola vez, el sistema combinado se comporta como si cada mensaje se hubiera manejado exactamente una vez. La receta es entrega at-least-once más un **consumidor idempotente**.

Algunos brokers anuncian procesamiento exactly-once, y la afirmación es legítima dentro de un límite. Un broker basado en log puede hacer atómico un ciclo de lectura, procesamiento y escritura cuando la entrada, la salida y el commit del offset viven dentro de ese mismo broker. Dentro de ese límite, una caída revierte todo a la vez. La garantía también exige que todo lector posterior lea solo datos confirmados, y se apoya en una identidad estable del productor para que una instancia antigua quede excluida tras un reinicio. El límite termina donde empiezan tus efectos secundarios. En el momento en que el consumidor escribe en una base de datos relacional, llama a un proveedor de pagos o envía un correo, la transacción del broker deja de cubrir el efecto y el problema de los duplicados regresa.

Por esa razón, el valor por defecto del libro sigue siendo at-least-once más consumidores idempotentes. Funciona con cualquier broker y cualquier efecto secundario, no depende de que una función esté habilitada y sobrevive a una migración a un producto distinto, un paso que cubre el Capítulo 14. Si puedes mantener todo un pipeline dentro del límite transaccional de un solo broker, la función puede reducir el trabajo, y aun así diseñaría los consumidores para que no dependan de ella.

Nota: Los productores también pueden crear duplicados, antes de que intervenga un consumidor. Un productor que agota el tiempo de espera de un ack de publicación normalmente volverá a publicar, y el broker puede tener ahora dos copias de un mismo evento. Ambas copias llevan el mismo `event_id`, siempre que el productor lo haya asignado una sola vez al crear el evento y no haya generado uno nuevo para el retry. Algunos brokers deduplican los retries del propio productor, mediante un número de secuencia dentro de una sesión del productor o un id de deduplicación dentro de una ventana de tiempo. Eso cubre un retry dentro de una misma conexión. No cubre un proceso que se reinicia y vuelve a publicar, como un relay del out-box, y por eso el consumidor sigue comprobando `event_id`. El Capítulo 4 explica cómo publicar sin perder eventos, y el Capítulo 7 explica cómo reconocer duplicados.

Qué garantiza realmente el broker

Todo lo anterior describe el lado del broker al consumidor. El lado del productor al broker tiene sus propios límites, y dependen más de la configuración que del nombre del producto.

Acks del productor

Cuando un productor publica, el broker puede enviar el ack en momentos distintos, y la forma útil de distinguir los ajustes es por cuántos nodos tienen el mensaje cuando el productor oye “hecho”. Sin ack, el productor no se entera de nada sobre el almacenamiento. Con ack del nodo receptor, un nodo ha aceptado el mensaje, lo haya volcado a disco o no. Con ack tras la replicación, el mensaje existe en un número requerido de nodos. El volcado a disco es una pregunta aparte, que viene más abajo.

La elección del productor importa por lo que una caída puede provocar. Con los dos primeros ajustes, un nodo del broker que muere justo después de enviar el ack puede llevarse el mensaje consigo. Con ack replicado, la respuesta llega más tarde y la ventana de pérdida es menor. El precio es una mayor latencia en cada publicación y una dependencia de que haya suficientes nodos disponibles para completar la escritura.

Los brokers que ofrecen publisher confirms, en la familia de colas, y niveles de ack, en la familia de logs, describen la misma idea. Comprueba qué significa realmente en tu configuración el ack que estás esperando. En mi opinión, esa es una de las páginas de documentación más valiosas que un equipo puede leer una vez y dejar por escrito.

Replicación y fsync

La replicación protege contra la pérdida de un nodo: el mensaje existe en varios. No protege contra que todo falle a la vez, y su garantía depende de cuántas copias tuvieron que existir antes de decirle “hecho” al productor. Comprueba tanto el nivel de ack como el número mínimo de réplicas que exige, porque “todas las réplicas” puede significar una cuando las demás se han quedado atrás. Comprueba también si un failover puede promover una réplica que no estaba al día, ya que eso puede perder datos que se le dijo al productor que estaban seguros.

Hay un punto relacionado, y a menudo mal entendido, sobre el disco. Escribir en un archivo normalmente deja los datos en el cache del sistema operativo, y los datos llegan al disco físico más tarde, a menos que el programa fuerce un volcado síncrono, que suele llamarse `fsync`. Forzar el volcado en cada mensaje es lento, así que muchos brokers no lo hacen en cada escritura. Confían en la replicación para cubrir la brecha y vuelcan según una programación. Que el `ack` de un broker concreto implique que los datos están en disco depende de su configuración, y yo no lo daría por supuesto.

No sé cuál de estos ajustes conviene a tu sistema, y depende de lo que cueste un evento perdido. Un `PaymentCaptured` perdido y un “page viewed” perdido son problemas distintos, y sería razonable configurarlos de forma diferente.

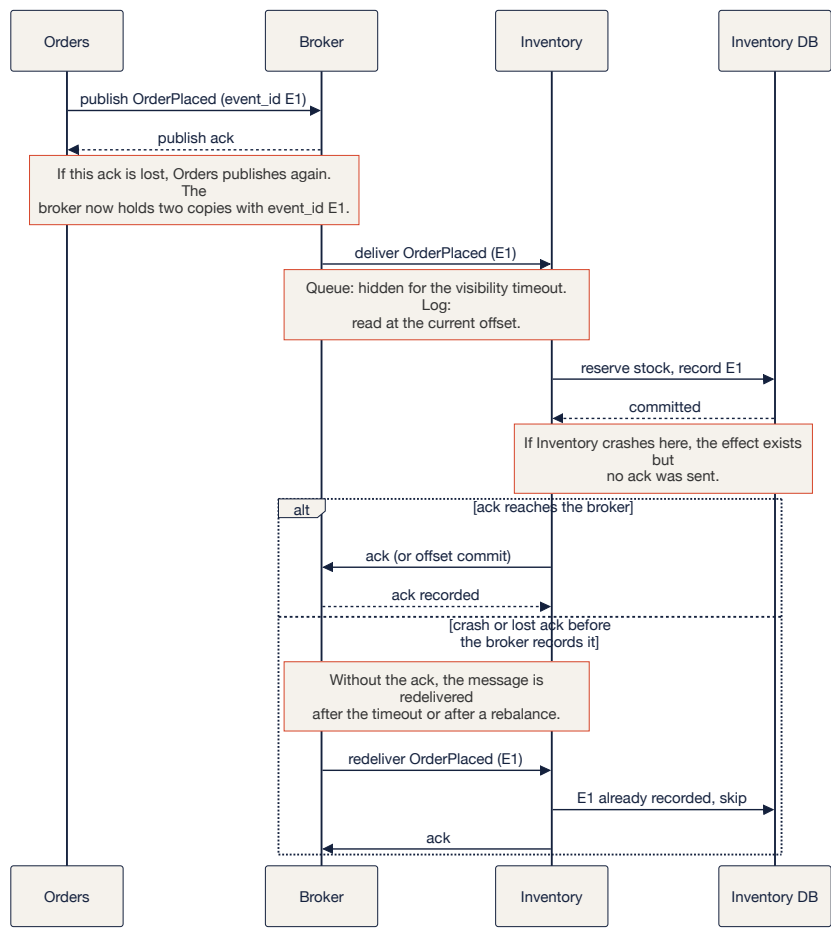
El límite de la garantía

Junta todo esto y el límite queda claro. Un broker puede prometer que un mensaje que aceptó se conservará y se ofrecerá a los consumidores, dentro de sus límites de retención o expiración, y solo con la durabilidad que permitan los ajustes de `ack` y replicación anteriores. No puede prometer que un mensaje llegue al broker en primer lugar, porque eso depende de lo que hizo el productor cuando su publicación agotó el tiempo. Tampoco puede prometer que el efecto de un consumidor ocurrió exactamente una vez.

Quedan dos brechas en manos de la aplicación. La primera está del lado del productor: un pedido guardado en la base de datos de `Orders` y un evento `OrderPlaced` publicado en el broker son dos escrituras, y una caída puede ocurrir entre ellas. El Capítulo 1 lo llamó el `dual write`, y el Capítulo 4 lo elimina. La segunda está del lado del consumidor, entre el efecto y el `ack`. El Capítulo 7 cierra esa brecha. Vistas como los tres traspasos con los que abrió este capítulo, el broker declara semántica de fallos para los dos primeros, y las dos brechas son lo que queda en tus manos: el `dual write` antes del primer traspaso, y la brecha entre el efecto y el `ack` en el tercero.

La vida de un mensaje

El diagrama siguiente sigue un mensaje `OrderPlaced` desde el servicio `Orders` hasta un efecto terminado en `Inventory`. Busca los dos puntos en los que un fallo deja al emisor sin respuesta, marcados en las notas: el ack de publicación perdido y el ack del consumidor perdido. Todo lo que ha descrito el capítulo ocurre dentro de esta única secuencia.



La rama `else` del diagrama es el punto del capítulo. La reentrega no es un error y ningún componente se comportó mal. Lo único que impide que el stock se reserve dos veces es que `Inventory` registró `E1` en la misma transacción que la reserva y lo comprobó de nuevo en la segunda entrega.

Así se ve en pseudocódigo un consumidor con esta propiedad. Usa un `inbox`, una tabla de valores `event_id` procesados en la base de datos del propio consumidor, y coloca el `ack` después del `commit`.

```

function handleOrderPlaced(message):
    // a redelivery of work that already committed is acknowledged,
    not redone
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique constraint on event_id makes a concurrent
duplicate
        // fail here, roll back, and come back as a redelivery that
the
        // check above then acknowledges
        inboxInsert(message.event_id)
        // transaction commits here, or rolls back on any error

    // ack comes last: a crash before this line causes a redelivery,
    // which the inbox check turns into an ack rather than a second
reservation
    ack(message)

```

Este es el diseño que vuelve segura la semántica at-least-once, y se introduce aquí solo para mostrar dónde pertenece el ack. El Capítulo 7 lo desarrolla, incluidos los casos en los que la comprobación del inbox por sí sola no basta.

Trade-offs y costos

Elegir una semántica es elegir qué fallo prefieres gestionar. Ninguna de las opciones elimina el fallo, y cada una lo traslada a otro lugar.

At-least-once te cuesta trabajo de idempotencia en cada consumidor.

Cada consumidor necesita una forma de reconocer el trabajo repetido, normalmente una tabla inbox en su propia base de datos. Eso es almacenamiento, una restricción única en la ruta caliente y una política de limpieza. Para un equipo con una docena de consumidores es un patrón que aplicar una docena de veces. Cuando el efecto de un consumidor es idempotente de forma natural, como fijar un estado a un valor fijo, el inbox puede ser innecesario, y aun así dejaría por escrito el porqué.

At-most-once te cuesta integridad. Es simple y rápido, y solo es aceptable cuando alguien ha decidido de antemano que un mensaje perdido sale más barato que un duplicado. Esa decisión debe ser explícita y quedar re-

gistrada. La forma habitual del error es un equipo que lo eligió por defecto, mediante ajustes de ack automático, sin darse cuenta.

El procesamiento transaccional dentro de un broker te cuesta portabilidad. Puede reducir el manejo de duplicados dentro de su límite, y ata el diseño a las funciones de un producto, a que todos los participantes permanezcan dentro de ese límite y al esfuerzo operativo de mantenerlo. Cuando cruzas a una base de datos relacional o a una API externa, necesitas idempotencia de todos modos. No rechazaría la función, y tampoco construiría la corrección solo sobre ella.

Los acks fuertes del productor cuestan latencia y disponibilidad. Esperar a varias réplicas hace más lenta cada publicación y hace que la escritura falle cuando son pocos los nodos alcanzables. Para los eventos que no deben perderse, ese trade-off suele ser el correcto. Para telemetría de alto volumen y poco valor, quizá no.

Los logs cuestan contabilidad del lado del consumidor, y las colas cuestan historial. Si eliges un log, te corresponden la política de commit de offsets y el manejo de un mensaje que bloquea una partición. Si eliges una cola, renuncias al replay. Cuando necesitas retries independientes para cada mensaje y ningún historial, un log puede ser más maquinaria de la que merece el trabajo.

Cuándo no aplicar el consejo de este capítulo: un sistema con un solo productor, un solo consumidor y tolerancia a una pérdida ocasional, como un feed interno de métricas, no necesita ni un inbox ni una replicación fuerte. El costo de las garantías debe corresponderse con el costo del fallo. El error no es elegir una garantía más débil; el error es no saber que lo hiciste.

Errores comunes

- **Tratar “entregado” como “procesado”.** Es tentador porque el dashboard muestra entregas y acks, y esas son las cifras que se ven sanas. Entregado significa que el broker pasó el mensaje. Procesado significa que el efecto se confirmó con commit. Haz seguimiento de ambos, y haz que tu ack siga al efecto.

- **Enviar el ack antes de terminar el trabajo.** Es tentador porque evita la reentrega y deja la cola con buen aspecto. Convierte cada caída de un consumidor en un mensaje perdido. Envía el ack después de que el efecto se confirme y maneja los duplicados que resulten.
- **Dejar activado por defecto el auto-commit de offsets.** Es tentador porque funciona en una demo y no requiere código. Según el cliente, puede hacer el commit en un momento sin relación con tu procesamiento, de modo que una caída puede perder o repetir un tramo arbitrario. Haz el commit de forma explícita, después del efecto.
- **Fijar el visibility timeout a ojo.** Es tentador porque el valor por defecto suele parecer bien. Un timeout más corto que una ejecución lenta pero sana provoca procesamiento duplicado concurrente. Basa el valor en el tiempo de procesamiento legítimo más lento que puedas observar, más un margen, y extiéndelo de forma deliberada para los jobs largos.
- **Crear que la función exactly-once de un broker cubre tu base de datos.** Es tentador porque la documentación usa esa frase. La garantía está acotada al límite del propio broker. Todo lo que quede fuera, como una fila de tu base de datos o un correo, necesita idempotencia.
- **Suponer que un mensaje publicado es un mensaje almacenado.** Es tentador porque la llamada de publicación devolvió. Según el nivel de ack, el mensaje puede existir solo en la memoria de un nodo. Averigua qué espera tu productor y déjalo por escrito.
- **Suponer que el rebalanceo es raro.** Es tentador porque ocurre en silencio. Los deploys, los eventos de escalado y las caídas pueden provocar uno, según el protocolo de grupo y los ajustes de pertenencia, y cada rebalanceo puede repetir un lote de mensajes. Diseña como si la reentrega ocurriera a diario, porque el costo de equivocarse es un efecto duplicado.

Ideas clave

- Un mensaje pasa por tres traspasos, del productor al broker, del broker al consumidor y del consumidor al efecto, y las garantías de un broker cubren solo los dos primeros.
- Las colas hacen seguimiento de cada mensaje y lo eliminan al recibir el ack, mientras que los logs retienen los mensajes con independencia

de los lectores y dejan que cada grupo de consumidores mantenga su propio offset, que es lo que hace posible el replay.

- Dónde pones el ack o el commit del offset decide la semántica: antes de procesar da at-most-once, después de procesar da at-least-once, y ninguna posición evita a la vez la pérdida y los duplicados.
- La reentrega es rutinaria, no excepcional, porque los visibility timeouts expiran, los consumidores se caen y los rebalanceos (que los deploys suelen causar) repiten lotes.
- Exactly-once es un efecto construido a partir de la entrega at-least-once y un consumidor idempotente, y el procesamiento exactly-once propio de un broker se detiene en el límite de ese broker.
- El ack de un productor es tan fuerte como los ajustes de replicación y de volcado que lo respaldan, así que el equipo debe saber y dejar registrado qué significa “publicado” para cada evento importante.
- Cada opción semántica traslada el fallo en lugar de eliminarlo, y la elección correcta es aquella cuyo fallo restante has decidido que puedes permitirte gestionar.

La semántica de la mensajería te dice lo que el broker hará y no hará en un día normal. El Capítulo 3 se ocupa de los días anormales y construye un modelo práctico de lo que realmente se rompe: particiones de red, caídas, consumidores lentos, relojes y personas.

Capítulo 3 — Un modelo de fallos práctico

Imagina Payments en Parcelly en mitad de una captura. Orders ha enviado un comando `CapturePayment` a Payments, y Payments ha llamado al proveedor de pagos externo para capturar el dinero. La conexión con el proveedor se queda en silencio. Treinta segundos después, el cliente HTTP de Payments se rinde y lanza un timeout.

Payments tiene ahora que responder una pregunta que no puede responder. ¿Capturó el proveedor el pago antes de que muriera la conexión, o nunca recibió la solicitud? Ambas historias producen exactamente la misma observación: silencio, y luego un timeout. Si Payments hace un retry y la primera captura había tenido éxito, a la persona compradora se le cobra dos veces. Si Payments informa de un fallo y la primera captura había tenido éxito, Orders cancela un pedido que ya está pagado.

Nada de esta historia es exótico. No hay un bug, ni un informe de caída, ni una ingeniera o un ingeniero que haya cometido un error. Una red hizo lo que hacen las redes, y el sistema no tuvo forma de saber en cuál de dos mundos estaba viviendo.

La mayor parte de lo que sale mal en un sistema orientado a eventos tiene esta forma: un paso ocurrió, o no, y falta la evidencia. Este capítulo nombra las maneras en que esa evidencia se pierde, para que puedas diseñar para cada una a propósito. La pregunta a la que responde es esta: ¿qué puede romperse exactamente en un sistema como el de Parcelly, qué verás cuando ocurra y en qué parte de este libro se maneja cada fallo?

El problema y por qué es difícil

Un modelo de fallos es una lista escrita de las cosas que supones que pueden salir mal y de las que supones que no. Todo sistema tiene uno, lo haya escrito alguien o no. La versión no escrita suele contener dos entradas: “el

servicio se cae” y “la red es lenta”. Son reales, pero son los fallos que se anuncian solos.

El caso simple es un fallo que puedes detectar, como una conexión rechazada o un error de base de datos. Cada uno te dice que el paso no se completó.

El caso difícil es el fallo que te deja en la incertidumbre. Un timeout no dice “la operación falló”. Dice “dejé de esperar”. La operación puede haber fallado, puede haber tenido éxito, o puede seguir ejecutándose y estar a punto de tener éxito después de que te rindieras. Basta un timeout y un efecto secundario al otro extremo para sentirlo.

Por eso el enfoque obvio, escribir código para el happy path y añadir un `try` y un `retry` alrededor de cada llamada, no se sostiene. Reintentar es una decisión sobre el futuro, tomada sin conocer el pasado. Si el intento anterior tuvo éxito, el `retry` es un duplicado. Si te niegas a reintentar, puede perderse un mensaje. Ninguna de las dos opciones es errónea en sí misma; cada una es errónea en algún mundo que no pudiste distinguir de aquel en el que estabas.

Hay una segunda razón. Los fallos se combinan: un deploy reinicia un consumidor mientras un broker hace failover y un productor hace un `retry`. Enumerarlos uno por uno sigue siendo el buen comienzo, pero producción los entrega juntos.

Así que el trabajo de este capítulo es la clasificación. Para cada fallo, haré las tres preguntas del Capítulo 1: qué puede fallar en esta costura, cómo lo observarías y qué garantía sobrevive. En las entradas siguientes, “qué es cierto en realidad” responde la primera, “qué ves” responde la segunda, y el patrón nombrado al final de cada entrada responde la tercera. Juntas, estas tres respuestas son la semántica de un fallo, y el resto del libro sigue pidiéndolas.

Diez fallos que importan

La lista siguiente es el conjunto con el que yo empezaría para un sistema construido con servicios, bases de datos relacionales y un message broker. No es completa, y ninguna lista lo es.

Particiones de red e incertidumbre por timeout

El escenario inicial es el caso modelo. Dos partes están conectadas por una red, y la red pierde, retrasa o reordena el tráfico. Una **partición de red** es la forma extrema, en la que dos grupos de máquinas no pueden alcanzarse en absoluto. Un timeout es la forma cotidiana, y es mucho más común.

Lo que cada lado puede observar es asimétrico. Quien llama solo ve que no llegó ninguna respuesta. Quien recibe puede haber hecho el trabajo y perdido la respuesta, o no haber visto nunca la solicitud. Lo mismo vale para cada salto de Parcelly: un servicio que llama a un transportista externo, un productor que envía al broker, un consumidor que envía un ack al broker.

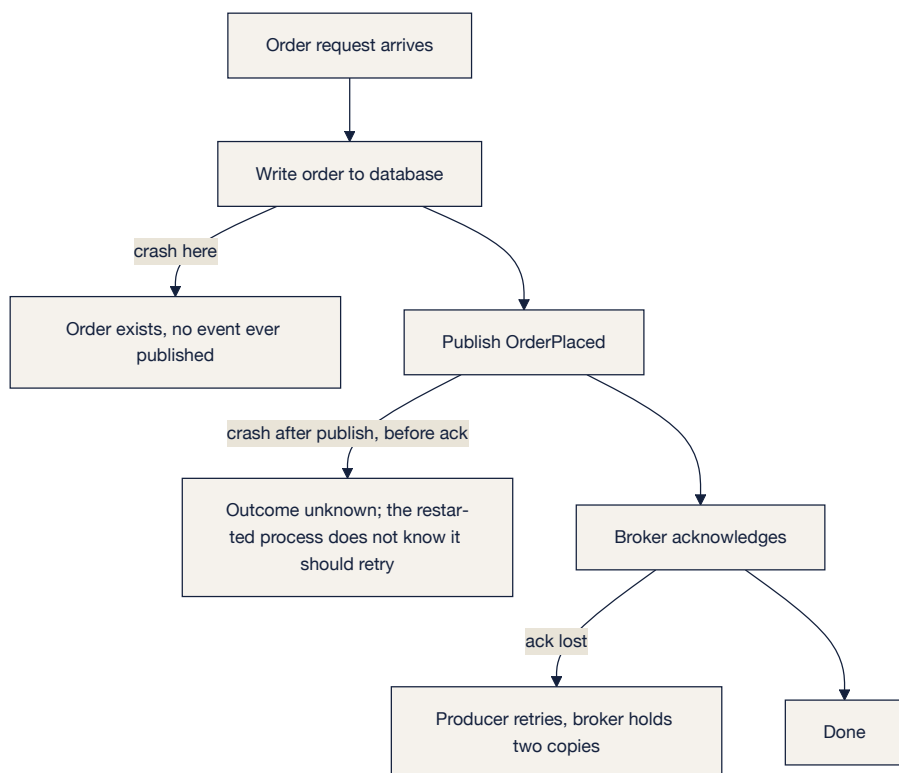
La única respuesta honesta a esto es hacer que la operación sea segura de repetir. Si Payments envía al proveedor una clave de idempotencia estable con cada captura, y el proveedor trata una clave repetida como la misma solicitud, entonces reintentar tras un timeout resulta inofensivo. El diseño deja de depender de la respuesta a “¿ocurrió?”, porque ambas respuestas llevan al mismo estado final. El Capítulo 6 explica cuándo y cómo reintentar, y el Capítulo 7 explica la idempotencia que vuelve seguro el retry.

Algunos sistemas externos no ofrecen esa clave. Entonces preguntas al sistema qué ocurrió antes de volver a actuar, que es un paso de conciliación, y el Capítulo 13 lo cubre.

Ventanas de caída del productor

Considera a Orders gestionando un pedido nuevo. Debe guardar el pedido en su base de datos y publicar `OrderPlaced` en el broker. Son dos sistemas, y ninguna transacción los abarca. Este es el dual write del Capítulo 1, y lo interesante es qué deja atrás cada ventana de caída.

El proceso puede caerse entre las dos escrituras, y el orden de las escrituras decide qué daño obtienes.



El diagrama dibuja el orden de primero la base de datos y muestra tres ventanas: una caída antes de la publicación, una caída después de ella con el resultado desconocido, y un ack perdido. Si el commit de la base de datos va primero y el proceso muere antes de publicar, el pedido existe y nadie más adelante se entera de él. Payments nunca autoriza, Inventory nunca reserva, y la persona compradora espera una confirmación que nunca llega. Nada da error. El pedido se queda ahí.

El orden inverso, que el diagrama no dibuja, tiene su propia ventana. Si publicas primero, una caída antes del commit deja un estado peor: un evento que anuncia un hecho que nunca llegó a ser cierto. Los servicios posteriores actúan sobre un pedido que no existe. Y en cualquiera de los dos órdenes, si el ack del broker se pierde en el camino de vuelta, el productor hará un retry y el broker puede tener dos copias del mismo evento.

Lo que se ve desde fuera es silencioso: pedidos sin eventos, eventos sin pedidos y algún duplicado ocasional. Nada en los logs dice “perdido”. El Capítulo 4 introduce el transaccional outbox, que elimina la ventana al hacer

que el cambio de estado y la intención de publicar sean una única transacción local. El Capítulo 5 explica cómo viajan después las filas del outbox hasta el broker.

Failover del broker

Los brokers se replican para que una máquina pueda morir sin llevarse los datos con ella. Cuando muere la máquina que tiene la copia líder de una partición o de una cola, otra toma el relevo. Esto es un **failover**, y es un comportamiento diseñado, no un mal funcionamiento.

Lo que importa es lo que se le dijo al productor antes del failover. Que un mensaje confirmado instantes antes de que muriera un líder sobreviva depende de cómo se configuró el broker para enviar el ack: nunca, después de que el nodo receptor lo haya aceptado (lo haya volcado a disco o no), o después de que un número requerido de réplicas lo tenga. Los ajustes más fuertes cuestan latencia y throughput. Los más débiles pueden perder mensajes confirmados en un failover. Esto difiere entre familias de brokers y entre configuraciones, así que no afirmaré aquí un comportamiento concreto para un producto concreto; el punto es que la garantía tiene un límite, y el límite es un ajuste que alguien eligió.

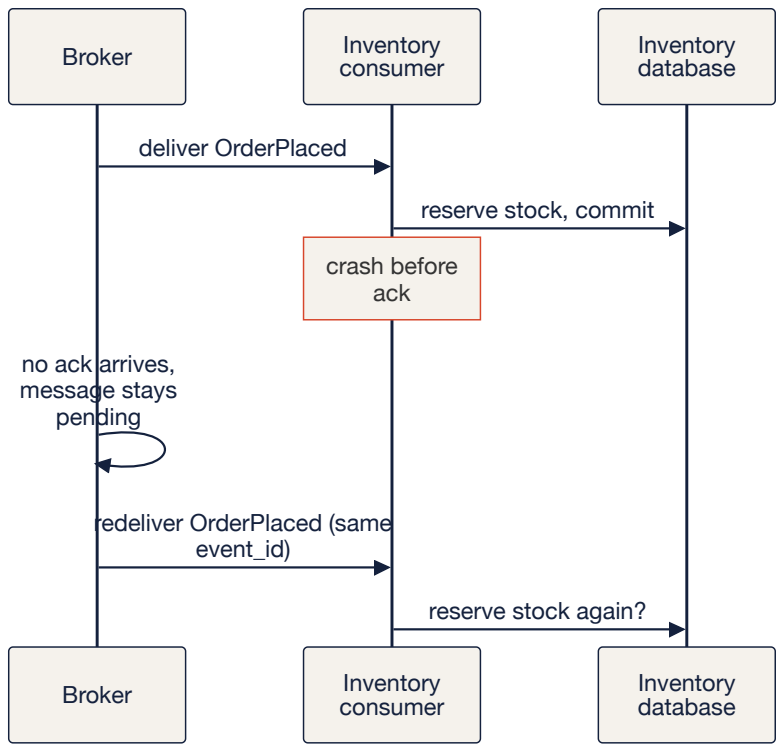
Durante el failover, los productores también ven errores y timeouts, reintentan y pueden publicar duplicados de mensajes que el antiguo líder ya había aceptado. El Capítulo 2 describe dónde se detienen las garantías del broker. La regla aquí es tratar “confirmado por el broker” como evidencia fuerte y no como prueba.

Lo que ves: una ráfaga de errores de publicación, una caída en el throughput de los consumidores y luego una oleada de reentregas. Si se perdió un mensaje, no ves nada en absoluto, y por eso importa la conciliación del Capítulo 13.

Caída del consumidor: procesado pero sin ack

Ahora, el lado del consumidor. Inventory recibe `OrderPlaced`, reserva stock en su base de datos y hace commit. Luego el proceso muere antes de poder enviar el ack del mensaje.

Desde el punto de vista del broker, el mensaje nunca terminó. En un broker basado en colas, el broker nota la conexión cerrada o expira el visibility timeout, y el mensaje se reentrega. En un broker basado en log, el offset del consumidor nunca se confirmó, así que el siguiente consumidor que sea propietario de la partición empieza desde la posición antigua y vuelve a leer el mensaje. En ambas familias, la segunda entrega llega para un trabajo que ya está hecho.



La última línea del diagrama es todo el problema. Un consumidor que reserva stock de forma incondicional en cada entrega lo reserva dos veces. Esta es la fuente ordinaria de duplicados, y es la razón por la que el valor por defecto del libro es la entrega at-least-once combinada con consumidores idempotentes. Un consumidor puede caerse en puntos distintos, y cada punto tiene una consecuencia diferente:


```

function handleOrderPlaced(message):
    // crash point A: before any work. Redelivery starts clean.
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique event_id makes a concurrent duplicate fail here
        inboxInsert(message.event_id)
    // transaction commits here
    // crash point B: committed, not acked. Redelivery hits the inbox
    check
    // above and is acknowledged without repeating the work.

    ack(message)
    // crash point C: after ack. Nothing to repair.

```

Esta es una versión compacta del handler del Capítulo 2, recortada para mostrar los puntos de caída. El efecto del punto de caída A es nulo. El efecto del punto de caída C es nulo. El interesante es B, y solo un consumidor que registra lo que ha procesado, en la misma transacción que el trabajo, lo sobrevive. El Capítulo 7 lo construye como corresponde.

Lo que ves: efectos secundarios duplicados, como dos reservas, dos correos, dos entradas de libro mayor, y ningún error en ninguna parte, porque cada intento de procesamiento tuvo éxito.

Consumidores lentos y backpressure

No todo fallo es una caída. Un consumidor que es simplemente lento puede hacer tanto daño, y es más difícil de notar porque nada está roto.

Supón que Notifications llama a un proveedor de correo que empieza a tardar 10 segundos por solicitud en lugar de 100 milisegundos. Notifications sigue funcionando, pero cada mensaje ocupa ahora un worker 100 veces más. Los mensajes llegan más rápido de lo que se terminan, así que el retraso del consumidor, que el Capítulo 2 definió, crece de forma constante.

La lentitud desencadena entonces fallos secundarios. En un broker basado en colas, un mensaje que supera su visibility timeout reaparece mientras el primer worker todavía lo está procesando, así que un segundo worker empieza el mismo mensaje. Los duplicados añaden carga a un sistema que ya

está lento. En un broker basado en log, un consumidor que no vuelve a llamar al broker a tiempo puede ser declarado muerto y sus particiones reasignadas, lo que inicia el siguiente fallo de esta lista.

El **backpressure** es la disciplina de acotar el trabajo en curso en cada etapa para que la carga se mantenga finita. En un sistema mediado por un broker, el broker absorbe la diferencia, que es el beneficio del desacoplamiento temporal, pero el buffer es finito y se agota de dos maneras distintas. Cuando se llena el disco del broker, este puede frenar o bloquear a los productores, o dejar de servir por completo, y cuál de esas cosas hace tu broker es algo que conviene averiguar antes del día en que ocurra. Sea cual sea, el productor lo ve, y eso es backpressure real: ruidoso y visible. Cuando un consumidor se queda más atrás que el período de retención de un log, o un mensaje en cola sobrevive a su expiración, el broker elimina mensajes que nunca se leyeron. Eso no es una demora. Es una pérdida silenciosa: no se lanza ningún error, ningún consumidor falla, y los eventos `PaymentCaptured` sin leer han desaparecido. Mucho antes de cualquiera de los dos límites, un backlog ya puede incumplir un plazo de negocio, como una autorización de pago que expira antes de capturarse.

Como el buffer del broker solo pospone el problema, la carga tiene que acotarse en varios lugares a la vez. Un consumidor puede limitar el número de mensajes en curso, o el lote que precarga. Puede limitar la concurrencia de sus workers, de modo que un proveedor de correo lento ate un número fijo de workers y no todos. Puede aplicar un rate limit a sus llamadas hacia la dependencia lenta, o pausar el consumo por completo mientras la dependencia se recupera. Una cola puede tener un límite de longitud con una política de desbordamiento explícita, de modo que tú elijas qué se descarta o se rechaza en lugar de dejar que lo elija el broker.

El lado productor y la topología también fijan límites. Del lado productor, el outbox absorbe a un broker que rechaza publicaciones, porque las filas esperan en la base de datos hasta que el relay pueda entregarlas. Y en un broker basado en log, el número de particiones limita cuántos consumidores de un grupo pueden trabajar en paralelo, de modo que añadir instancias deja de ayudar en una cifra fijada de antemano. Cada uno de estos límites cambia throughput por estabilidad, y cada uno necesita una métrica, o no sabrás que es el que te frena.

Lo que ves: retraso y antigüedad de los mensajes en aumento, un backlog creciente, contadores de reentrega que suben. El Capítulo 6 explica el backoff, los límites de mensajes en curso y la pausa del consumo, el Capítulo 8 explica qué hacer con los mensajes que no pueden procesarse a tiempo, y el Capítulo 13 explica las métricas de retraso y antigüedad que te avisan pronto.

Mensajes que fallan siempre y dependencias que fallan para todos los mensajes

Hasta ahora, todo fallo era cuestión de azar: una caída, una respuesta perdida, un mal momento. Algunos fallos son deterministas. Un **poison message** es un mensaje que falla en cada intento, sea cual sea el momento. Supón que un `OrderPlaced` llega a `Inventory` con un SKU que no existe en su catálogo. `Inventory` lanza un error, el mensaje se reentrega, el error se repite y nada del siguiente intento es distinto del anterior.

La causa también puede estar fuera del mensaje. Si el proveedor de `Payments` está caído, todas las capturas fallan, y todos los mensajes parecen venenosos mientras dure la caída. Los dos casos piden respuestas opuestas. Un mensaje que nunca tendrá éxito debe salir pronto del flujo principal, hacia una **dead-letter queue (DLQ)**, un lugar aparte donde los mensajes que no pueden procesarse esperan a una persona o a un job de reparación. Un mensaje que falla porque una dependencia está caída no debe enviarse a la DLQ en absoluto, porque tendría éxito tras la recuperación, y enviar a la DLQ todo un backlog durante una caída convierte un problema temporal en uno manual.

Lo que sale mal entre medias sigue un patrón. Los retries sobre un poison message consumen intentos y capacidad sin ninguna posibilidad de éxito. En un broker basado en log, o en una cola que preserva el orden, uno de esos mensajes puede bloquear todo lo que tiene detrás. Muchos consumidores reintentando contra una misma dependencia que falla añaden carga a lo que está intentando recuperarse: un retry storm. Y un bug de código puede fallar sin ningún error: el consumidor envía el ack, retorna y escribe el valor equivocado, de modo que nada de la ruta de fallo llega a ejecutarse.

Lo que ves: el mismo `event_id` apareciendo en los logs una y otra vez, contadores de reentrega que nunca bajan, una partición atascada, una DLQ que crece durante una caída y, para el bug silencioso, nada en absoluto hasta que una comprobación de conciliación discrepa. El Capítulo 6 explica cómo distinguir qué fallos merecen un retry, el Capítulo 8 trata los poison messages y la ruta de dead letters, y los capítulos 13 y 14 explican cómo encontrar y reparar el daño de un bug que no lanzó ningún error. El precio de cualquier clasificación es que puede ser errónea, así que mantén la decisión visible y reversible.

Relojes y timestamps en los que no puedes confiar

Toda máquina tiene un reloj, y no hay dos relojes que coincidan exactamente. Se desvían, se corrigen mediante sincronización, y una corrección puede mover un reloj hacia atrás. Una máquina virtual pausada despierta con un reloj equivocado por el tiempo que haya dormido.

La consecuencia para los eventos es concreta. `occurred_at` es un hecho sobre cuándo ocurrió algo según una máquina. Comparar el `occurred_at` de un evento de Orders con el de uno de Payments te dice algo sobre sus relojes tanto como sobre el orden de los eventos. Si ordenas por timestamp para decidir qué actualización gana, un servicio con el reloj unos segundos adelantado ganará todos los conflictos que debería perder.

Los timeouts tienen una trampa relacionada. Un deadline expresado como hora de reloj de pared, “esta autorización es válida hasta las 14:00”, lo compara una máquina cuya idea de las 14:00 puede diferir de la de quien lo emitió. Para medir duraciones en una sola máquina, la herramienta correcta es un reloj monotónico, que solo avanza. Para ordenar entre máquinas, yo no usaría la hora de reloj de pared en absoluto.

Lo que ordena los eventos de forma confiable es la estructura del sistema: en un broker basado en log, una partición con clave por el agregado, que da un orden total para los eventos de esa clave siempre que haya una única ruta de productor para cada clave y el número de particiones no cambie, o un número de versión explícito en el agregado que los consumidores puedan comparar. El Capítulo 9 trata el orden en profundidad. Trata `occurred_at` como información para personas y para analítica, y desconfía de él como entrada para la corrección.

Lo que ves: actualizaciones aplicadas fuera de orden, secuencias “imposibles” en los logs en las que un efecto parece preceder a su causa, y bugs que aparecen solo en un host.

Deploys y rebalanceos

Un deploy es un fallo planificado. Las instancias se detienen y se inician, y durante el tiempo intermedio el trabajo se interrumpe a medias. Si tu proceso de deploy envía una señal de terminación y el consumidor sale con mensajes en curso, has creado a propósito el caso de “procesado pero sin ack”, de forma repetida, en cada release.

Un **rebalanceo** es la versión de la familia de logs, y puede ocurrir sin un deploy. Cuando miembros entran en un grupo de consumidores o salen de él, o cuando un miembro se considera sin respuesta, las particiones se reasignan. Durante el traspaso, dos consumidores pueden creer brevemente que son propietarios de la misma partición, y un consumidor que iba lento puede seguir procesando mensajes después de que su partición se haya movido a otro lugar. El equivalente de la familia de colas son los consumidores competidores cuyos visibility timeouts expiran mientras el trabajo continúa.

El patrón de daño es una ráfaga de duplicados en el momento del deploy y, a veces, un breve bloqueo mientras se reasignan las particiones. Un deploy que recorre todas las instancias en rápida sucesión puede desencadenar varios rebalanceos seguidos, y en el peor caso los consumidores pasan más tiempo reasignando trabajo que haciéndolo. Dos mitigaciones reducen esto, al precio de más configuración que entender: la reasignación cooperativa, en la que solo se quitan las particiones que deben moverse, y una identidad estable de miembro, para que una instancia reiniciada pueda recuperar sus antiguas particiones sin una reorganización completa.

Haz que los consumidores se detengan de forma ordenada, terminando o liberando los mensajes en curso antes de salir, y supón que la ruta ordenada fallará a veces, de modo que el consumidor idempotente siga siendo la defensa real. También escalonaría los rolling deploys para que no desencadenen reasignaciones en un bucle apretado. Lo que ves: duplicados y picos de retraso que coinciden con las horas de release.

Errores humanos

La última categoría es la que la mayoría de los documentos de arquitectura omiten. Las personas operan estos sistemas, y las personas cometen errores bajo presión, normalmente mientras reparan un fallo anterior.

Imagina a la persona on-call de Parcelly que encuentra 40,000 dead letters y pulsa el botón para hacer un redrive de todas. Si se corrigió la causa, fluyen sin problema. Si no se corrigió, vuelven a fallar de inmediato e inundan el sistema que se intentaba sanar. Si algunas ya se habían gestionado con una solución provisional manual, son duplicados de trabajo que alguien hizo a mano.

Otras formas son igual de ordinarias. Alguien edita una fila directamente para desbloquear a un cliente, saltándose el outbox, de modo que no se publica ningún evento y los servicios posteriores divergen. Alguien despliega un productor con un payload cambiado antes de que los consumidores puedan leerlo. Alguien purga una cola en el entorno equivocado. Esto es lo que ocurre cuando las herramientas de reparación son amplias y faltan las salvaguardas.

Trataría las acciones de los operadores como parte del modelo de fallos y las diseñaría como cualquier otro componente: redrive en lotes pequeños con un interruptor de pausa, acciones de reparación que pasen por la misma ruta que los cambios normales para que se emitan eventos, y cada acción manual escrita en un registro de auditoría. El Capítulo 8 trata las dead letters y el redrive, el Capítulo 12 trata los cambios de payload, y el Capítulo 13 trata la conciliación, que es como te enteras de que una corrección humana salió mal.

Lo que ves: un pico repentino justo después de que alguien tocara algo, y datos que discrepan entre servicios sin ningún mensaje fallido que lo explique.

Perder un store completo

Todos los fallos hasta ahora suponían que los stores sobreviven: la base de datos conserva lo que confirmó y el broker conserva lo que confirmó con un ack. La última entrada abandona esa suposición. Se pierde un clúster o una región. Un broker pierde datos más allá de lo que cubría su replica-

ción. La base de datos de un consumidor se restaura desde un backup tomado hace seis horas. La base de datos de un productor se restaura a un punto anterior a los últimos cientos de pedidos.

No son un único fallo, porque el daño depende de qué store retrocedió en el tiempo. Si se restaura la base de datos de Inventory, olvida las reservas hechas después del punto de restauración, y puede que el broker ya no conserve los eventos que las recrearían. Si se restaura la base de datos de Orders, olvida pedidos sobre los que los servicios posteriores ya actuaron, y Payments conserva capturas de pedidos que Orders ya no conoce. Ninguno de los lados lanza un error. Cada uno es coherente consigo mismo y está equivocado respecto al otro.

Una propiedad ayuda. Como el inbox vive en la misma base de datos que los efectos, una restauración devuelve ambos al mismo momento. Los mensajes procesados antes del punto de restauración siguen en el inbox, así que un replay los omite, lo cual es correcto porque sus efectos también sobrevivieron. Los mensajes procesados después del punto de restauración son los que corren riesgo, y reproducirlos es seguro: el inbox ya no los recuerda, y sus efectos se perdieron con él.

Los efectos externos son la excepción. Una captura en el proveedor de pagos no retrocede con tu base de datos, así que esas llamadas necesitan sus propias claves de idempotencia, o el replay le cobra a la persona compradora una segunda vez. La regla es conciliar antes de reanudar el consumo: comparar lo que cree cada servicio, reparar las diferencias y solo entonces dejar correr el flujo.

Lo que ves: nada, al principio, que es justo el problema. Los servicios están sanos y los dashboards están en verde, mientras los totales discrepan entre ellos. El Capítulo 13 explica la conciliación que encuentra la discrepancia, y el Capítulo 14, Recuperación, desastres y migración de brokers, explica cómo restaurar, hacer replay y reanudar. Diseñar para esto cuesta backups que de verdad se prueban, una retención que supere tu tiempo de recuperación y la disciplina de ensayar una restauración antes de necesitarla.

El mapa de fallos

Aquí está el mismo material en un solo lugar. Tenlo a mano en tus revisiones de diseño.

Fallo	Qué ves	Qué es cierto en realidad	Dónde se maneja
Partición de red o timeout	Quien llama no recibe respuesta	La operación puede haberse ejecutado o no	Capítulo 6, Capítulo 7, Capítulo 13
Caída del productor entre la escritura y la publicación	Pedidos sin eventos, o eventos sin pedidos	El estado y el evento discrepan	Capítulo 4, Capítulo 5
Ack del broker perdido	Eventos duplicados en el broker	La primera publicación tuvo éxito	Capítulo 7
Failover del broker	Errores de publicación y luego reentregas; en raras ocasiones, pérdida silenciosa	La garantía depende de los ajustes de ack	Capítulo 2, Capítulo 13
Caída del consumidor después del trabajo	Efectos secundarios duplicados, sin errores	Trabajo hecho, mensaje todavía pendiente	Capítulo 7
Consumidor lento	Retraso y antigüedad de mensajes crecientes	Consumidor vivo, quedándose atrás	Capítulo 6, Capítulo 8, Capítulo 13
Retención o expiración durante el retraso	Nada; los eventos que faltan nunca llegan	El broker eliminó mensajes sin leer	Capítulo 8, Capítulo 13
Poison message o dependencia que falla	El mismo mensaje se reentrega; partición atascada; DLQ creciendo durante una caída	El fallo es determinista, o la causa está fuera del mensaje	Capítulo 6, Capítulo 8
Bug del consumidor sin error	Nada; los totales discrepan más tarde	El handler tuvo éxito y escribió el resultado equivocado	Capítulo 13, Capítulo 14
Desfase de reloj	Actualizaciones fuera de orden, secuencias imposibles	Los timestamps discrepan con la causalidad	Capítulo 9
Deploy o rebalanceo	Duplicados y retraso en el momento del release	La propiedad del trabajo cambió a medias	Capítulo 7, Capítulo 13
Error del operador	Pico tras una acción manual; divergencia silenciosa	Una reparación cambió el estado fuera de la ruta normal	Capítulo 8, Capítulo 13
Perder un store completo	Nada al principio; los servicios discrepan tras una restauración	Un store retrocedió en el tiempo, o sus datos desaparecieron	Capítulo 13, Capítulo 14

Dos cosas destacan. Primero, los duplicados aparecen en la columna “Qué ves” por tres causas distintas (ack perdido, caída del consumidor, deploy), y las reentregas por una cuarta. Por eso un solo patrón, el consumidor idempotente, se gana un capítulo entero. Segundo, la última columna sigue volviendo a los capítulos 13 y 14. Algunos fallos no pueden prevenirse ni siquiera detectarse en el momento en que ocurren, así que el sistema también necesita una forma de notar después que dos vistas de la verdad discrepan.

Dejar por escrito el modelo de fallos

Un modelo de fallos solo es útil si está escrito, actualizado y compartido. En mi opinión, la práctica vale más que cualquier entrada concreta, porque el acto de escribir obliga a hacerse la pregunta “¿qué suponemos aquí?”, y las suposiciones son donde viven las sorpresas.

Lo mantendría como un documento breve en el repositorio, junto a la descripción de la arquitectura, con una entrada por fallo y los mismos campos cada vez. Los campos reflejan las tres preguntas: “where” y “we assume” dicen qué puede fallar, “observable as” dice cómo lo verías, y “defense” dice qué garantía sobrevive. Una entrada utilizable se ve así:

failure:	consumer crashes after commit, before ack
where:	Inventory, handler for OrderPlaced
we assume:	this happens during every deploy, and at random
otherwise	
observable as:	redelivered message with the same event_id
effect if unhandled:	second stock reservation for the same order
defense:	inbox row inserted in the same transaction as the reservation
verified by:	test that kills the consumer between commit and ack
owner:	Inventory team
last reviewed:	2026-09-30

Los campos que importan son los tres últimos. “Verified by” convierte la entrada de una creencia en una afirmación que puede probarse, y el Capítulo 15 muestra cómo probarla. “Owner” significa que a alguien se le envía un page. “Last reviewed” admite que el modelo se pudre: se añaden servicios, se cambian brokers, y una suposición que era cierta el año pasado puede no serlo ahora.

Nota: El modelo también debe declarar lo que supones que no puede ocurrir. “Suponemos que la base de datos nunca pierde una transacción confirmada” es una entrada legítima. Es mejor dejar la suposición por escrito, y saber qué harías si fuera falsa, que dejarla implícita.

Usa el modelo en tres lugares. En las revisiones de diseño, recorre cada flujo nuevo contra la lista y pregunta a qué fallos está expuesto. En las revisiones de incidentes, busca a qué entrada pertenece el incidente y, si a ninguna, añade una. En las pruebas, convierte cada entrada en un escenario. Un fallo que aparece en el modelo pero en ninguna prueba es una esperanza.

Trade-offs y costos

Escribir un modelo de fallos cuesta tiempo, y la incomodidad de registrar lo que no sabes. Un equipo pequeño con un servicio y una cola no necesita todas las entradas de esta lista. Empieza por los tres o cuatro fallos que más dolerían.

El modelo también tiene un límite. Toda defensa de este libro supone que algunos fallos quedan fuera del alcance: que la base de datos es duradera, que la replicación del broker funciona como se configuró. Esas suposiciones son razonables, y cada una es un lugar donde el modelo puede estar equivocado. Enumerarlas te permite decidir si son aceptables, algo que no puedes hacer mientras son invisibles. Un bug en tu propio código es distinto. Está dentro del alcance, como un fallo que hay que detectar y del que hay que recuperarse, porque va a ocurrir, y el modelo tiene que decir cómo lo notarías y cómo repararías lo que escribiste.

Un modelo con 40 entradas no se leerá, y uno con tres pasará por alto casos. Apuntaría a una lista que una persona nueva en el equipo pueda leer de una sentada.

Diseñar para cada fallo con la máxima fuerza tampoco es el objetivo. Sobrevivir a un failover del broker sin pérdida cuesta latencia y dinero, y sobrevivir a un doble fallo cuesta más. Un correo de Notifications que a veces se envía dos veces no merece la misma defensa que una entrada de

Ledger que no debe duplicarse nunca. El trabajo del modelo es permitirte gastar según el riesgo. Si no puedo nombrar el costo de negocio de un fallo, dudaría en pagar un costo técnico para prevenirlo.

El modelo no vuelve confiable al sistema. Hace que tus suposiciones sean inspeccionables.

Errores comunes

- **Modelar solo las caídas.** Una caída es tentadora de modelar porque es limpia y detectable. Los fallos costosos son los ambiguos, como los timeouts y los consumidores lentos. Solución: para cada llamada, escribe qué hace quien llama cuando no llega ninguna respuesta.
- **Tratar un timeout como un fallo.** Resulta natural leer “sin respuesta” como “no ocurrió”. Solución: trátalo como desconocido, y diseña la operación para que repetirla sea seguro o comprobable.
- **Confiar en el ack del broker como prueba.** La palabra “confirmado” suena definitiva. Solución: conoce qué garantiza realmente tu ajuste de ack, y concilia para los casos en que no lo hace.
- **Ordenar por timestamp.** `occurred_at` está ahí mismo, en el envelope, y ordenar por él lleva una línea. Solución: ordena dentro de una clave de partición por versión del agregado, y guarda los timestamps para las personas.
- **Ignorar los deploys.** Los releases parecen rutina, no fallo. Solución: cuenta cada deploy como una caída programada de un consumidor, y prueba el apagado mientras hay mensajes en curso.
- **Dejar fuera a los operadores.** Las herramientas de reparación reciben menos revisión que el código de producto. Solución: incluye el re-drive, el replay y las ediciones manuales en el modelo, con límites de lote, interruptores de pausa y registros de auditoría.
- **Tratar una base de datos restaurada como un comienzo limpio.** Una restauración parece el final del incidente. Solución: tras cualquier restauración, concilia con los demás servicios antes de que se reanude el consumo, y revisa los efectos externos que no retrocedieron.
- **Hacer un retry de un fallo que no va a cambiar.** Los retries parecen diligentes. Solución: separa los fallos que dependen del momento de

los que dependen del mensaje, y envía el segundo tipo pronto a la ruta de dead letters.

- **Escribir el modelo una sola vez.** Un documento escrito durante una fase de diseño es una instantánea. Solución: asigna un responsable, revisalo después de los incidentes y fecha cada entrada.

Ideas clave

- Un timeout significa “dejé de esperar”, no “falló”, así que diseña operaciones que sigan siendo correctas se resuelva como se resuelva lo desconocido.
- Cada costura tiene sus propias ventanas de caída, y el daño es silencioso. Un dual write deja pedidos sin eventos, o eventos sobre pedidos que no existen, y un consumidor que se cae tras hacer commit pero antes de enviar el ack recibe un duplicado en cada reentrega, lo que convierte a los consumidores idempotentes en un requisito.
- El ack de un broker es evidencia con un límite fijado por los ajustes de replicación y de volcado, así que conoce el límite y planifica conciliar lo que queda fuera de él.
- Los consumidores lentos y los deploys causan duplicados y bloqueos sin que ningún componente esté roto, así que acota la carga, mide el retraso y la antigüedad de los mensajes, y detén los consumidores de forma ordenada.
- Un mensaje que falla siempre, o una dependencia que falla para todos los mensajes, necesita una respuesta distinta a la de un fallo de azar, así que clasifica antes de reintentar y reserva la dead-letter queue para lo que no se va a recuperar.
- Los timestamps de reloj de pared discrepan entre máquinas, así que ordena los eventos dentro de una clave de partición por versión del agregado y usa `occurred_at` para las personas.
- Los operadores y las restauraciones son parte del sistema: un store restaurado ha retrocedido en el tiempo, así que concilia antes de reanudar el consumo, y mantén un modelo de fallos escrito que nombre responsables, pruebas y fechas de revisión para que las reparaciones no se conviertan en el siguiente incidente.

El primer fallo que hay que eliminar es el del productor, donde un cambio de estado y su evento pueden separarse sin que nadie lo note. El Capítulo 4 abre la Parte II con el transactional outbox, que cierra esa ventana.

PARTE II

Publicar con confiabilidad

Capítulo 4 — El problema del dual write y el transactional outbox

Imagina Parcelly unas semanas después de lanzar las promociones para vendedores. Una ingeniera de soporte abre un ticket que dice, completo: “La persona compradora dice que completó el checkout del pedido 81724 pero la persona vendedora nunca recibió una etiqueta de envío. El pedido figura como realizado”. La ingeniera revisa la base de datos de Orders y encuentra la fila, con estado `PLACED`, creada a las 14:02:11. Luego revisa Payments y no hay ninguna autorización para ese pedido. Inventory tampoco tiene una reserva. Para el resto de Parcelly, el pedido no existe.

Busca en los logs de Orders alrededor de las 14:02 y encuentra una marca de deploy. El servicio se reinició 200 milisegundos después de confirmar el pedido y antes de publicar `OrderPlaced`. El código hizo exactamente lo que se escribió para hacer: guardar y luego publicar. Nada lanzó un error, porque el proceso que lo habría lanzado ya no existía. Supón que una consulta rápida muestra otros 11 pedidos con la misma forma, todos dentro del mismo minuto.

El equipo corrige los 11 a mano y añade una nota al runbook. A nadie le deja tranquilo. El código no está mal de ninguna manera que una revisión señalaría, y aun así puede perder trabajo cada vez que un proceso muere en el momento equivocado. Una máquina más grande no cambiará eso, ni tampoco una persona desarrolladora más cuidadosa. La pregunta de este capítulo es si un servicio puede cambiar su propia base de datos y avisar al resto del sistema de una manera que no pueda descomponerse, sin una transacción que abarque la base de datos y el broker.

El problema y por qué es difícil

El Capítulo 1 definió el dual write: dos sistemas que se actualizan sin una transacción compartida. Orders debe escribir una fila en su base de datos relacional y publicar un mensaje en el broker. La base de datos garantiza

la atomicidad de lo primero. El broker garantiza la durabilidad, en alguna configuración, de lo segundo. Nada garantiza que ocurran ambas cosas o ninguna.

El Capítulo 3 esbozó en un diagrama las ventanas de fallo del productor. Aquí quiero recorrerlas como una tabla, porque el orden de las dos escrituras es una decisión de diseño que la mayoría de los equipos toma sin darse cuenta, y detrás de cada elección hay un fallo esperando.

Todo orden de las escrituras se rompe

Toma las dos secuencias obvias y pregunta, para cada posible punto de fallo, cómo queda el mundo después. Es el mismo ejercicio que escribir la semántica de fallos de la costura entre la base de datos y el broker: qué puede fallar, cómo lo observarías y qué garantía sobrevive. La tabla siguiente es ese ejercicio para el productor.

Secuencia	Punto de fallo	Estado de la base de datos	Estado del broker	Daño
Commit y luego publicar	Caída después del commit, antes de publicar	Pedido guardado	Sin evento	Evento perdido. Los consumidores posteriores nunca se enteran del pedido
Commit y luego publicar	La publicación falla o agota el tiempo, el proceso sobrevive	Pedido guardado	Desconocido	Quien llama debe hacer un retry o rendirse. Rendirse pierde el evento
Commit y luego publicar	El broker acepta, el ack se pierde	Pedido guardado	Evento presente	Reintentar publica un duplicado
Publicar y luego commit	Caída después de publicar, antes del commit	Sin pedido	Evento presente	Evento fantasma. Los consumidores posteriores actúan sobre un hecho que nunca llegó a ser verdad
Publicar y luego commit	El commit falla (restricción, deadlock, timeout)	Sin pedido	Evento presente	El mismo evento fantasma, ahora con un error visible en el productor

Secuencia	Punto de fallo	Estado de la base de datos	Estado del broker	Daño
Publicar dentro de la transacción, commit al final	El commit falla después de publicar	Sin pedido	Evento presente	El bloque de transacción le da al código una falsa sensación de seguridad, y el broker no forma parte de él

La última fila merece un comentario, porque es la que parece más segura. Muchos frameworks permiten publicar desde dentro de un bloque de transacción de la base de datos. Se lee como si la publicación fuera parte de la transacción. No lo es. La llamada al broker ocurre de inmediato y no se puede revertir, y el commit sigue ocurriendo después. Es el orden de publicar y luego commit con mejor indentación.

La imagen especular es el callback “after commit” que ofrecen muchos frameworks, un hook que se ejecuta una vez que la transacción se ha confirmado. Los equipos suelen creer que es seguro, porque la publicación solo ocurre para datos confirmados. Es la primera fila de la tabla: commit y luego publicar, con la intención guardada en la memoria del proceso. Una caída entre el commit y el hook pierde el evento, y nada registra que alguna vez se debió.

También hay una cuarta forma que la tabla no captura, que es un broker lento. Si la publicación ocurre dentro de la transacción, la base de datos mantiene bloqueos y una conexión mientras espera al broker. Un broker que está lento durante un minuto se convierte en una base de datos lenta durante un minuto. Has acoplado la disponibilidad de dos sistemas que se suponía que eran independientes.

Por qué los retries no ayudan

La reparación tentadora es un bucle de retry alrededor de la publicación. Los retries ayudan con la segunda fila de la tabla, donde el proceso sobrevive y puede volver a intentarlo. No hacen nada por la primera fila, donde el proceso ya no existe, y crean el duplicado de la tercera fila. El Capítulo 6 trata los retries como corresponde. Por ahora, fíjate en que un bucle de retry vive en la memoria del proceso, y el fallo que nos importa es que el proceso desaparezca.

La segunda reparación tentadora es un job periódico que compara la base de datos con el broker y vuelve a publicar lo que falta. Algunos equipos lo construyen, y creo que es una red de seguridad razonable, a la que el Capítulo 13 vuelve como conciliación. Como mecanismo principal tiene dos problemas. El broker rara vez se puede consultar por clave de negocio, así que “lo que falta” se vuelve una conjetura costosa. Y el job mismo realiza el mismo dual write, de modo que puede fallar de las mismas maneras, solo que con menos frecuencia y más confusión.

Por qué no una transacción distribuida

Si el problema son dos sistemas sin una transacción compartida, la respuesta clásica es darles una. El **commit en dos fases (2PC)**, estandarizado para bases de datos y gestores de transacciones como XA, coordina varios gestores de recursos. En la primera fase cada participante se prepara, es decir, promete que puede confirmar y retiene sus bloqueos. En la segunda fase un coordinador le dice a cada participante que confirme, o que haga rollback si alguno no pudo prepararse.

Sobre el papel esto cierra la ventana. En la práctica rara vez es la respuesta para un broker y la base de datos de un servicio, por razones que son sobre todo operativas.

- **El soporte es desigual.** Muchos brokers populares no implementan ninguna interfaz de gestor de transacciones. Donde alguno la implementa, las garantías y el manejo de fallos difieren de los de la base de datos, y heredas los dos conjuntos de reglas.
- **Los participantes retienen bloqueos mientras el coordinador decide.** Si el coordinador se cae entre las dos fases, las transacciones preparadas quedan en duda. Las filas siguen bloqueadas hasta que alguien, o algo, las resuelva. La ventana de duda es exactamente el tipo de fallo raro y largo que nadie ensaya.
- **El coordinador pasa a ser un componente que debes operar, proteger y recuperar.** Necesita sus propios logs durables y un procedimiento de recuperación que tiene que ser correcto.
- **La disponibilidad se multiplica.** Una transacción que necesita que la base de datos, el broker y el coordinador estén sanos a la vez solo está disponible cuando lo están los tres. La independencia que hacía atractivo al broker desaparece.

- **El throughput y la latencia sufren.** Dos viajes de ida y vuelta, escrituras forzadas en cada participante y bloqueos retenidos a través de llamadas de red se acumulan.
- **No se extiende al mundo exterior.** Un proveedor de correo o de pagos no se unirá a tu transacción. Incluso un equipo que adopta 2PC para el broker sigue necesitando otra respuesta para cada llamada externa.

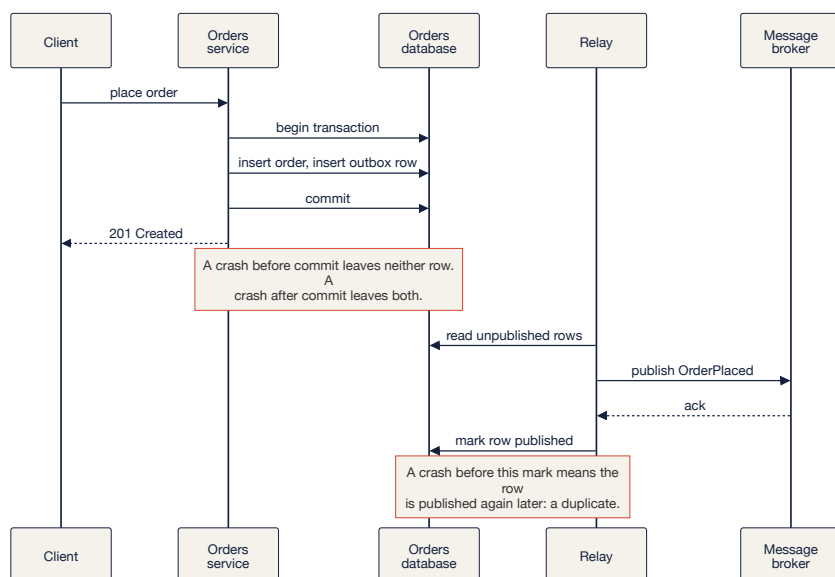
No descartaría 2PC en todos los casos. Un equipo que controla ambos recursos, opera un gestor de transacciones maduro y tiene la capacidad operativa para resolver transacciones en duda puede hacer que funcione. Para la mayoría de los equipos que construyen un marketplace con servicios, cada uno con su propia base de datos y un broker que no expone una interfaz XA, el costo es alto y el beneficio es estrecho. Lo que quieren es más pequeño que una transacción distribuida: quieren que el cambio de estado y la intención de publicar se confirmen juntos. Ese es un problema local, y las transacciones locales se entienden bien.

El transactional outbox

La idea es convertir el dual write en una única escritura más un movimiento posterior y repetible. En lugar de publicar en el broker dentro de la solicitud, el servicio escribe una fila que describe el evento en una tabla de su propia base de datos, en la misma transacción local que el cambio de estado. Esa tabla es el **transactional outbox**. A partir de aquí, el libro lo llama el outbox.

Un proceso aparte, el **relay**, lee las filas del outbox y las publica en el broker. El Capítulo 5 trata del relay. En este capítulo lo tratamos como una función que al final entrega cada fila al menos una vez, y miramos qué nos aporta eso.

El razonamiento es corto. O el pedido y su fila del outbox se confirman juntos, o ninguno lo hace, porque viven en una base de datos y en una transacción. Una vez que la fila existe, la intención de publicar es durable. Una caída después del commit ya no pierde el evento, porque la fila sigue ahí cuando el servicio, o su reemplazo, vuelve a arrancar. Una caída antes del commit no deja ni pedido ni fila, así que no existe ningún evento fantasma.



Mira las dos notas del diagrama. La primera muestra la ventana que cerramos. La segunda muestra la ventana que conservamos: el relay puede publicar y caerse antes de registrar que lo hizo. El outbox no elimina los duplicados, traslada el fallo a un lugar donde está acotado y se maneja. Vuelvo sobre esto más abajo.

La tabla

El outbox sigue la forma que usa el resto del libro. Lleva todos los campos del envelope, para que el relay pueda construir el evento sin preguntarle nada al servicio productor. Este capítulo añade cuatro columnas más allá del envelope, cada una por una razón.

```

table outbox:
  id                bigint primary key          // allocation order, not
commit order
  event_id          uuid unique not null        // stable across
redelivery
  aggregate_type    text not null               // "order", "shipment"
  aggregate_version  bigint not null            // aggregate's version,
bumped in this transaction
  event_type        text not null               // "OrderPlaced"
  schema_version    int not null
  occurred_at       timestamp not null          // time of the fact
  created_at        timestamp not null default dbNow() // insert time,
database clock
  producer          text not null              // "orders"
  correlation_id     text not null
  causation_id       text not null
  partition_key      text not null              // aggregate id, e.g.
"order-81724"
  payload           json not null
  headers           json null                  // transport hints, trace
context
  published_at      timestamp null              // null means not yet
published

```

Algunas de estas decisiones necesitan una defensa.

- **id y event_id son cosas distintas.** El `event_id` es un UUID que los consumidores usarán como clave de idempotencia, y no debe cambiar cuando el evento se publica dos veces. El `id` es un número de secuencia asignado por la base de datos que el relay puede usar para encontrar filas. No dice nada sobre el orden de commit, y la sección siguiente muestra por qué.
- **aggregate_version es la fuente de verdad del orden.** Es la versión del agregado, incrementada en la misma transacción que escribe la fila. Para los pedidos de Parcelly es el `order_version`. Los consumidores que necesitan orden por agregado comparan este número, no el `id` ni el orden de llegada. El Capítulo 9 muestra cómo lo usa un consumidor. El envelope estándar no tiene un campo para él, así que cuando el relay construye el evento copia la columna en el payload como `order_version`, y ahí es donde los consumidores la leen.
- **aggregate_type y partition_key son distintos.** La `partition_key` es lo que el broker usa para el orden, y es el id del agregado, como `order-81724`. El `aggregate_type` le dice al relay, y a una persona que

lea la tabla, a qué tipo de cosa se refiere la clave, y es una forma cómoda de enrutar filas a distintos topics.

- **created_at es distinto de occurred_at.** La primera es cuándo se escribió la fila, y la segunda es cuándo ocurrió el hecho. Normalmente ambas están a milisegundos de distancia. Divergen cuando un servicio registra un hecho que conoció tarde, como un escaneo de transportista que llega después del evento. Una métrica de retraso del relay debería usar `created_at`, porque el retraso es una propiedad de la tabla, no del negocio. Deja que la base de datos la llene mediante el valor por defecto de la columna en lugar de llamar a `now()` en el código de la aplicación, de modo que el reloj sea uno solo. Sigue siendo la hora de inserción, no la del commit: una transacción larga hace que la fila parezca más antigua que la vista que tiene el relay de ella, y esa parte del retraso no es culpa del relay.
- **headers contiene aspectos de transporte.** El contexto de trace y las pistas de enrutamiento van aquí, para que el payload siga siendo el hecho de negocio. Un equipo que no pueda decidir si un campo es un dato o una pista de transporte debería leer el Capítulo 11 antes de añadirlo al payload.
- **published_at admite nulos, y el nulo tiene significado.** Una fila sin `published_at` es una fila que el relay todavía le debe al broker. Algunos equipos usan en su lugar una columna `status`, con valores como `PENDING`, `PUBLISHED` y `FAILED`. Los dos diseños llevan la misma información. Prefiero la marca de tiempo porque además dice cuándo, lo cual es útil cuando preguntas cuánto se retrasó el relay, y porque hay menos que hacer mal. Si necesitas una forma de aparcas las filas que fallan de forma permanente, añade entonces una columna `failed_at`, no antes.

Ese momento llega antes de lo que parece. Una fila que el broker nunca aceptará, como un payload demasiado grande o uno que el serializador rechaza, bloquea su clave, y en el relay más simple la tabla entera, hasta que alguien la repare. La primera vez que eso ocurra, necesitarás una columna `failed_at` (o un estado `FAILED`, si elegiste columnas de estado) y una alerta sobre las filas aparcadas. El Capítulo 5 muestra cómo la usa el relay.

Un índice sobre las filas no publicadas, en orden de inserción, es el único índice que el relay necesita. Sin él, el relay recorre una tabla que crece para siempre. El Capítulo 5 trata la limpieza.

Escribir en la misma transacción

El código productor es corto, y esa brevedad es el punto. La función siguiente es todo el cambio en la ruta de solicitud de Orders.

```
function placeOrder(request):
    order = buildOrder(request)
    // a new order starts at version 1
    order.version = 1

    transaction:
        orderInsert(order)
        outboxInsert({
            event_id:      newUuid(),
            aggregate_type: "order",
            event_type:    "OrderPlaced",
            schema_version: 1,
            aggregate_version: order.version, // 1 for a new order
            occurred_at:   now(),
            producer:      "orders",
            correlation_id: order.id,
            causation_id:  request.call_id,
            partition_key: order.id,
            payload:       orderPlacedPayload(order)
        })
    // commit point: the order and its event exist together or not at
    all

    return order
```

Tres detalles importan más de lo que parece.

Primero, nada en la función habla con el broker. Una caída del broker no hace fallar la creación de pedidos, y un broker que tarda un minuto en responder no mantiene abierta una transacción de base de datos. Es una ganancia real de disponibilidad: Orders puede aceptar pedidos mientras el broker está caído, y el outbox crece hasta que el relay pueda vaciarlo.

Segundo, el `event_id` se crea antes del commit y se guarda en la fila. Cada republicación posterior de este evento lleva el mismo id. Esta es la propiedad de la que dependen los consumidores, y es la razón por la que generar el id en el relay, en el momento de publicar, sería un error.

Tercero, el payload se construye dentro de la transacción, a partir del estado que la transacción ha escrito. Si el payload se construyera más tarde volviendo a leer el pedido, una actualización concurrente podría cambiarlo, y el evento describiría un hecho que no existía en ese momento. Un evento es una instantánea de lo que ocurrió, y la instantánea se toma en el mismo lugar que el cambio.

El pseudocódigo supone que la inserción del pedido y la del outbox usan la misma conexión de base de datos y la misma transacción. Esto se rompe por accidente más a menudo de lo que parece. Una capa de acceso a datos que abre una segunda conexión para el outbox, o que hace auto-commit de la inserción del outbox, restablece en silencio el dual write, ahora entre dos tablas. La prueba de esto pertenece a tu suite: fuerza un fallo después de la inserción del pedido y verifica que no existe ninguna fila del outbox.

Orden de commit, orden de `id` y el bloqueo del agregado

Un pedido nuevo no tiene competidor, así que `placeOrder` no necesita bloqueo. Todo cambio posterior sobre un pedido existente sí lo necesita, y el bloqueo tiene que ir primero. Aquí está `cancelOrder`, que emite `OrderCancelled`.

```
function cancelOrder(order_id, reason):
  transaction:
    // lock the aggregate row before anything else is written
    order = orderSelectForUpdate(order_id)
    order.version = order.version + 1
    orderUpdate(order, status: "CANCELLED")
    outboxInsert({
      event_id:      newUuid(),
      aggregate_type: "order",
      aggregate_version: order.version,
      event_type:    "OrderCancelled",
      partition_key: order.id,
      payload:       orderCancelledPayload(order, reason)
    })
    // remaining envelope fields as in placeOrder
  // commit point: state, version bump and event together
```

El bloqueo es lo que hace confiable la versión, y también lo único que hace que el orden de `id` signifique algo para un agregado. Sin él, toma dos transacciones sobre el mismo pedido. La transacción B inserta su fila del outbox y obtiene el `id` 101, y luego se bloquea en la fila del pedido que retiene la transacción A. A inserta el `id` 102 y confirma, y B confirma des-

pués. El orden de commit es A y luego B, y el orden de `id` es B y luego A. Un relay que ve las dos filas en un mismo poll las publica invertidas. Con el bloqueo tomado primero, B no puede insertar hasta que A haya confirmado, así que los ids de un agregado siguen su orden de commit.

Aun así, no haría que los consumidores dependan del `id`. Las filas de distintos agregados se intercalan, un relay en paralelo reordena entre lotes, y nadie fuera de la base de datos de Orders puede ver el `id`. El `aggregate_version` viaja con el evento, como `order_version` en el payload, y dice en qué orden ocurrieron los hechos de este pedido. El Capítulo 5 vuelve sobre lo que un relay puede y no puede preservar, y el Capítulo 9 muestra el lado del consumidor.

Lo que debe el relay

El contrato del relay es deliberadamente pequeño. Selecciona las filas no publicadas, publica cada una en el broker y la marca como publicada una vez que el broker lo confirma. La versión más simple está a continuación, y el Capítulo 5 trata las preocupaciones de producción que ignora, como el batching, el bloqueo, el orden y el failover.

```
function relayOnce():
  // oldest first
  rows = outboxSelectUnpublished(limit: 100)

  for each row in rows:
    result = publishAndAwaitAck(topicFor(row.aggregate_type),
    eventFrom(row))
    if result is failure:
      // stop: publishing later rows would overtake this one for
      its key
      return
    // if the process dies here, the row stays unpublished
    // and will be published again: this is the duplicate window
    outboxMarkPublished(row.id, now())
```

El comentario dentro del bucle es toda la garantía. La fila se marca solo después de que el broker confirma, por eso la llamada espera el acuse de recibo y por eso un fallo detiene el lote. Una caída entre la publicación y la marca deja la fila sin publicar. En la siguiente pasada, el relay la publica de nuevo. La alternativa, marcar primero y publicar después, convertiría el fallo en un evento perdido en lugar de un duplicado. El outbox elige el fallo más seguro a propósito.

Qué garantiza el outbox

El outbox te da tres cosas y te quita una.

- **Atomicidad entre el estado y la intención.** El cambio de estado y la decisión de publicar se confirman juntos. No hay un pedido confirmado sin una fila del outbox, ni una fila del outbox sin un pedido confirmado.
- **Durabilidad de la intención.** Una vez que la transacción se confirma, el evento es tan durable como la base de datos. El relay puede estar caído, el broker puede estar caído, el productor puede ser reemplazado, y la fila espera.
- **Publicación at-least-once.** Toda fila confirmada se publica al final, suponiendo que el relay y el broker se recuperen al final, que el relay encuentre las filas por estado y no por un cursor de id, que la limpieza purgue solo las filas publicadas, y que una fila que el broker nunca aceptará se aparque y genere una alerta en lugar de quedar bloqueando para siempre (todo esto es el Capítulo 5). Una fila puede publicarse más de una vez.
- **No garantizado: la publicación exactly-once.** El relay no puede publicar de forma atómica en el broker y marcar una fila en la base de datos, por la misma razón por la que no podía la aplicación. La ventana pasó de la ruta de solicitud al relay, donde es más pequeña, observable y reparable.

Quiero ser cuidadoso con ese resumen. El outbox no afirma que los eventos nunca se pierdan. Afirma que la ventana de pérdida del dual write del lado del productor está cerrada, siempre que la aplicación escriba a través del outbox para cada cambio de estado que deba producir un evento. Una persona desarrolladora que actualiza la tabla a mano, o por una ruta que se salta el outbox, vuelve a abrir la ventana. El Capítulo 3 lo enumeró como un error humano, y es la forma más común en que un equipo con outbox sigue perdiendo eventos.

El duplicado pasa a ser problema del consumidor

Como la publicación es at-least-once, todo consumidor de un evento publicado por outbox debe tolerar duplicados. No es un defecto que haya que eliminar con ingeniería, es el precio de la garantía, y conviene ser claros

sobre quién lo paga.

El equipo productor hace la parte más ligera. Añade una tabla, escribe una fila y opera un relay. Los equipos consumidores hacen la parte más difícil, porque deben reconocer un `event_id` que ya han visto y evitar repetir su efecto. Payments no debe autorizar dos veces. Notifications no debe enviar el mismo correo dos veces. Ledger no debe registrar el mismo asiento dos veces.

Advertencia: Adoptar un outbox sin consumidores idempotentes cambia un problema de eventos perdidos por un problema de efectos duplicados. Para algunos eventos, como un cobro doble, el segundo problema es el más costoso. Entrega las dos cosas juntas, y consulta el Capítulo 7 para ver cómo reconoce un consumidor el trabajo repetido.

También se lo diría con claridad a los equipos productores: el outbox es un contrato con tus consumidores. El `event_id` debe mantenerse estable en cada republicación del mismo hecho, y el contrato debería escribirse en la documentación del evento, no dejarse implícito.

Un breve recorrido por los fallos

Para contrastar el diseño con las ventanas del Capítulo 3, recorre de nuevo las caídas de Orders.

1. El proceso muere antes de que la transacción se confirme. No hay pedido, ni fila del outbox, ni evento. El cliente ve un error o un timeout y puede hacer un retry de la solicitud, lo cual es un problema aparte para la capa de API, cubierto por la discusión sobre idempotencia del Capítulo 7.
2. El proceso muere justo después del commit. El pedido y la fila existen. El relay publica la fila en su siguiente pasada. Puede que el cliente no haya recibido su respuesta, pero el sistema es consistente.
3. El relay muere después de publicar y antes de marcar. El evento está en el broker y la fila está sin publicar. La siguiente pasada lo publica de nuevo. Los consumidores ven un duplicado con el mismo `event_id`.

4. El broker no está disponible. El relay no logra publicar, deja las filas sin publicar y lo intenta de nuevo más tarde. Orders sigue aceptando solicitudes. El retraso crece, y también debería hacerlo una alerta.
5. El servicio pierde su conexión durante el commit y no conoce el resultado. O existen el pedido y su fila, o no existe ninguno, y el servicio no puede saber cuál. No debe adivinar. La respuesta segura es que el retry del cliente lleve una clave de idempotencia a nivel de API, de modo que en cualquier estado en que esté la base de datos, el retry converja en un pedido y un evento (Capítulo 7).

Cada caso termina en consistencia o en un duplicado, y ninguno termina en un evento fantasma o en una pérdida silenciosa. Eso es todo lo que el outbox afirma, y basta para justificarlo.

Hay un duplicado que se sitúa antes del outbox, y ninguna defensa posterior lo ve. Cuando un cliente hace un retry tras perder la respuesta y la API no tiene clave propia, Orders crea un segundo pedido y un segundo `OrderPlaced` con un `event_id` nuevo. El inbox de cada consumidor ve entonces dos eventos distintos, porque lo son. El outbox hizo su trabajo dos veces. La solución corresponde a la API, con el enfoque de `Idempotency-Key` del Capítulo 7.

Alternativas y vecinos

El outbox no es la única forma de detener el dual write. Otros dos diseños merecen una comparación justa, ya que un equipo podría razonablemente preferir cualquiera de ellos.

Listen to yourself

En el patrón **listen-to-yourself**, el servicio no escribe primero en su base de datos. Publica el evento en el broker, luego consume su propio evento y aplica el cambio de estado en el handler. El broker pasa a ser la única escritura, y la base de datos pasa a ser una proyección de lo que el servicio ha escuchado.

Esto elimina el dual write, porque solo hay una escritura. También cambia la forma del servicio. La solicitud del cliente regresa antes de que exista el estado, así que quien llama no puede leer su propia escritura a menos que el sistema ofrezca una forma de esperar. La validación debe ocurrir antes

de publicar, ya que un evento, una vez publicado, es un hecho, y uno inválido no puede rechazarse después sin una compensación. Y el handler que aplica el evento debe ser él mismo idempotente, lo que trae de vuelta la misma carga para el consumidor.

Lo consideraría para servicios que ya piensan en eventos y pueden tolerar consistencia eventual en sus propias lecturas. Para Orders, donde el navegador de la persona compradora espera ver el pedido justo después de pulsar el botón, no lo haría. La brecha de read-your-own-write es un problema de producto que el equipo acabaría resolviendo con soluciones improvisadas.

Event sourcing

En el **event sourcing**, los eventos son la fuente de verdad. El servicio almacena la secuencia de eventos por agregado, y el estado actual se deriva de ellos. La escritura en el event store es el cambio de estado, así que no hay nada que mantener consistente con un evento aparte. Publicar en el broker consiste entonces en leer del event store, que es un relay con otro nombre.

El event sourcing resuelve el dual write cambiando lo que es la base de datos. También te da un historial completo y la capacidad de reconstruir proyecciones. Cuesta un modelo de programación distinto, la evolución de esquemas de eventos almacenados que nunca puedes reescribir a la ligera, y una curva de aprendizaje para todo el equipo. Es un compromiso arquitectónico significativo y no lo adoptaría solo para evitar un outbox. Si ya lo usas, igualmente necesitas un relay, y todo lo que dice este capítulo sobre la publicación at-least-once se aplica.

Comparación de las opciones

Opción	Qué elimina	Qué cuesta	Encaja mejor cuando
Commit y luego publicar	Nada	Pérdida silenciosa en una caída	Perder un evento es aceptable, como las métricas de mejor esfuerzo
2PC / XA	La ventana, en teoría	Bloqueos, coordinador, soporte desigual	Controlas ambos recursos y ejecutas tú la recuperación

Opción	Qué elimina	Qué cuesta	Encaja mejor cuando
Transaccional outbox	La ventana del lado del productor	Una tabla extra, un relay, duplicados	El estado vive en una base de datos relacional y los eventos no deben perderse
Listen to yourself	La segunda escritura	Consistencia eventual en las lecturas propias, validación temprana	El servicio ya es event-first
Event sourcing	La escritura de estado separada	Un modelo de estado distinto	El historial es en sí mismo un requisito

Para un servicio que ya almacena su estado en una base de datos relacional y necesita contárselo a otros, el outbox es el cambio más pequeño. Por eso la mayor parte de este libro lo supone.

Trade-offs y costos

El outbox es barato de describir y no es gratis de operar. Estos son los costos que pondría delante de un equipo antes de que se comprometa.

Más piezas móviles. Hay una tabla, un proceso relay y un retraso que monitorear. Cuando algo se rompe, existe un lugar nuevo donde mirar. El relay es un componente del que alguien es responsable, y necesita deploys, alertas y un runbook. El Capítulo 5 da el detalle.

Más carga en la base de datos. Cada evento es una inserción en la misma transacción que la escritura de negocio, y cada publicación es una lectura y una actualización. Las transacciones se alargan un poco, y el volumen de escritura sube. Para la mayoría de los servicios el aumento es pequeño comparado con el resto del trabajo. Un servicio que escriba volúmenes de eventos muy altos debería medirlo en su propio hardware en lugar de fiarse de mi “pequeño”.

Latencia añadida. El evento llega al broker después de que la transacción se confirma, y después de que el relay nota la fila. Con polling, el retraso depende del intervalo de polling. Con un enfoque basado en el log puede ser menor, y el Capítulo 5 compara los dos. En cualquier caso, el outbox no es la elección para flujos en los que el evento debe llegar al broker a pocos milisegundos de la solicitud, y conviene conocer tu presupuesto de latencia antes de elegir.

Duplicados por diseño. Tratado más arriba. Si un consumidor no puede hacerse idempotente, el outbox no mejora nada para él, y el equipo tiene que decidir si ese consumidor debería existir en esa forma.

El orden solo se preserva en parte. El relay lee las filas por `id`, y el orden de `id` no es el orden de commit a menos que cada transacción bloquee la fila del agregado antes de insertar la fila del outbox. Ese bloqueo serializa a quienes escriben un mismo agregado, lo cual es un costo en sí mismo en los agregados calientes. Los relays en paralelo empeoran el orden, y el bloqueo no ayuda entre agregados. La columna `aggregate_version` da a los consumidores algo que comprobar, y cuesta un incremento de versión en cada escritura. El outbox te da un lugar donde hacer cumplir el orden, no el orden gratis. El Capítulo 5 y el Capítulo 9 vuelven sobre esto.

Crecimiento y limpieza. Las filas publicadas se acumulan. Sin borrado o archivado, la tabla crece hasta que la consulta del relay se vuelve lenta. La retención es una decisión, e interactúa con la conciliación, ya que las filas antiguas también son tu evidencia de lo que se publicó.

Solo funciona para estado en una base de datos que admite transacciones. Si el estado vive en un almacén sin transacciones multifila, necesitas otro enfoque, como un change stream propio del almacén o un diseño con event sourcing. Un almacén que ofrece atomicidad solo dentro de un documento tiene su propia variante: mantén los eventos pendientes dentro del documento del agregado, y deja que el relay los drene desde ahí.

Cuándo no usarlo

No añadiría un outbox cuando:

- el evento es de mejor esfuerzo y su pérdida es aceptable, como una métrica o una pista de analítica de poco valor
- el servicio no tiene estado durable propio, así que no hay nada que confirmar junto con el evento
- el sistema es lo bastante pequeño como para que un solo proceso sea propietario del estado y de sus consumidores, y una llamada directa sea más simple
- el equipo todavía no puede hacer idempotentes a los consumidores, en cuyo caso arreglaría eso primero, ya que el outbox eleva lo que está en

juego en esa brecha

Nota: “Usamos un outbox” es una afirmación sobre la ruta del productor de un servicio. No dice nada sobre si los consumidores son seguros, si el orden se mantiene o si los eventos están bien diseñados. Trátalo como un patrón dentro de un conjunto, no como un sello de confiabilidad.

Errores comunes

- **Escribir la fila del outbox en una transacción aparte.** Es tentador porque un hook del framework o una abstracción de event bus esconde la conexión. Las dos escrituras pasan a ser un dual write entre dos tablas de la misma base de datos, y la ventana vuelve. Usa una conexión y una transacción, y prueba el rollback.
- **Publicar desde un hook “after commit”.** Es tentador porque solo se dispara para datos confirmados y no necesita tabla. Es commit y luego publicar con la intención en memoria, así que una caída entre ambos pierde el evento. Escribe la fila en la transacción y deja que el relay publique.
- **Insertar la fila del outbox antes de bloquear el agregado.** El código funciona en todas las pruebas con un solo escritor. Bajo concurrencia, el orden de `id` y el orden de commit divergen para un mismo pedido, y el relay puede publicar sus eventos invertidos. Bloquea primero la fila del agregado, incrementa su versión y luego inserta.
- **Publicar desde la ruta de solicitud y conservar el outbox como respaldo.** Parece un doble seguro. Produce dos publicadores, duplicados con distinta temporización y un orden sobre el que no puedes razonar. Deja que el relay sea el único publicador.
- **Generar `event_id` en el relay.** Parece inofensivo, ya que el relay es quien construye el mensaje. Una republicación recibe entonces un id nuevo, y los consumidores no pueden reconocerla como un duplicado. Genera el id cuando se escribe la fila.
- **Marcar la fila como publicada antes de que el broker confirme.** Reduce los duplicados y parece más ordenado. Convierte un duplicado

en un evento perdido, que es el peor fallo. Marca después de la confirmación y acepta el duplicado.

- **Construir el payload después del commit.** Volver a leer el agregado para llenar el evento parece mantener simple el código productor. Otra transacción puede cambiar los datos entretanto, y el evento deja de coincidir con el hecho. Construye el payload a partir de lo que escribió la transacción.
- **Dejar que algunas rutas de código se salten el outbox.** Un script de migración, una herramienta de administración o una corrección con SQL a mano cambian el estado y no emiten nada. Haz pasar todo cambio de estado que importe por la misma función, y revisa las reparaciones manuales para ver qué eventos debieron producir.
- **Olvidar la limpieza.** La tabla funciona durante meses, y luego la consulta del relay se vuelve lenta en el peor momento. Decide la retención cuando crees la tabla.
- **Tratar el outbox como el final de la historia de la confiabilidad.** Cierra una ventana. Los consumidores siguen necesitando idempotencia, retries y un lugar para los mensajes que nunca tienen éxito.

Ideas clave

- Un dual write no puede hacerse seguro eligiendo un mejor orden: commit y luego publicar pierde eventos, publicar y luego commit los inventa, y una publicación dentro de un bloque de transacción sigue siendo lo segundo.
- El commit en dos fases puede cerrar la ventana en principio, pero para la base de datos de un servicio y un broker suele costar más en bloqueos, operación del coordinador y disponibilidad de lo que vale el problema.
- El transactional outbox convierte el dual write en una transacción local más un movimiento repetible, escribiendo el evento como una fila en la misma transacción que el cambio de estado.
- La fila del outbox lleva el envelope completo del evento, un `event_id` estable creado en el momento de la escritura para que toda republicación pueda reconocerse, y un `aggregate_version` incrementado bajo

el bloqueo del agregado, porque el orden de `id` no es el orden de commit.

- El outbox garantiza la atomicidad del estado y la intención, la intención durable y la publicación at-least-once, y no garantiza que un mensaje se publique exactamente una vez.
- Los duplicados son el precio de la garantía, así que un outbox sin consumidores idempotentes cambia un fallo por otro.
- Listen-to-yourself y el event sourcing también eliminan el dual write, pero cambian cómo el servicio lee y almacena su estado, así que los elegiría por lo que te dan y no meramente para evitar un relay.

Publicar es solo la mitad del movimiento: algo debe leer esas filas y llevarlas al broker sin perderlas, reordenarlas ni inundarlo. El Capítulo 5 trata el relay, el polling frente al change data capture y cómo operar el outbox en producción.

Capítulo 5 — Relays, polling y change data capture

Imagina una mañana de miércoles en Parcelly. El equipo de Orders puso en producción el transactional outbox hace dos semanas. Cada evento `OrderPlaced` se escribe ahora en la tabla `outbox` en la misma transacción local que el propio pedido, y la ventana del dual write del Capítulo 4 está cerrada. El equipo considera resuelto el problema y pasa al siguiente ticket.

El relay es un proceso pequeño que se despierta cada segundo, lee las filas con un `id` mayor que el último que vio, las publica y recuerda el nuevo `id` más alto. Se ejecuta como una sola instancia, y nadie ha pensado en él desde el día en que se desplegó.

Entonces un deploy rutinario reemplaza al relay en el peor momento. La instancia anterior ha publicado en el broker un lote de 200 filas pero todavía no ha guardado su nueva posición. La instancia nueva arranca desde la última posición guardada, lee las mismas 200 filas y las publica otra vez. Inventory reserva stock por segunda vez para pedidos que ya había procesado. Una semana después se añade un job de limpieza que borra en silencio filas que el relay aún no ha leído. Al mes siguiente, una transacción de larga duración se confirma tarde y sus filas nunca salen de la tabla.

Ninguno de estos fallos está en el patrón outbox en sí. Todos están en la parte que mueve las filas al broker. ¿Cómo se opera esa parte para que tarde en fallar, se recupere rápido y sea honesta sobre los duplicados que aún puede producir?

El problema y por qué es difícil

El transactional outbox te da una garantía: si el pedido se confirmó, la intención de publicar su evento se confirmó con él. No dice nada sobre la segunda mitad del trayecto. Alguien tiene que leer las filas, entregárselas al

broker y recordar que lo hizo. Ese alguien es el **relay**, el proceso que mueve las filas del outbox al broker.

Un relay hace tres cosas: encuentra las filas que hay que publicar, las publica y registra que lo ha hecho. Son tres pasos separados, y el broker y la base de datos no comparten una transacción. El mismo problema de dual write que eliminamos del productor reaparece dentro del relay, con una ventana más estrecha.

Las ventanas de caída son pocas y se pueden enumerar. La tabla es la semántica de fallos del relay: qué puede fallar, cómo lo observarías y qué garantía sobrevive.

Cuándo ocurre la caída	Estado de la fila	Resultado
Antes de publicar	Sin publicar	La fila se recoge de nuevo, el evento se publica una vez. Seguro.
Después de publicar, antes de marcar	Sin publicar, pero el evento está en el broker	La fila se recoge de nuevo, el evento se publica dos veces. Un duplicado.
Después de marcar	Publicada	Nada que hacer. Seguro.

No existe ningún orden de los tres pasos que cierre la fila del medio. Si el relay marca primero y publica después, una caída entre ambos pierde el evento, que es el fallo que el outbox existe para evitar. Por eso el relay publica primero y marca después, y acepta que una caída entre ambos produce un duplicado. Es el mismo trade que hizo el capítulo del outbox: entrega at-least-once, con el manejo de duplicados trasladado a los consumidores (Capítulo 7).

Por tanto, la primera pregunta de diseño no es cómo evitar los duplicados. Es cómo mantener pequeña la ventana, cómo asegurarse de que las filas salgan en un orden sensato y cómo ver que el relay se está quedando atrás antes de que lo vea un cliente. La segunda pregunta de diseño es una elección entre dos mecanismos para leer las filas: hacer polling de la tabla, o leer el propio log de transacciones de la base de datos. Cada uno traslada la dificultad a otro lugar.

El relay con polling

Un **relay con polling** consulta la tabla del outbox a intervalos, publica lo que encuentra y marca esas filas como publicadas. Es el mecanismo más simple que funciona, y empezaría por aquí a menos que los números digan otra cosa.

El bucle básico

```
function relayLoop():
  while running:
    // oldest first
    rows = selectUnpublished(limit: 100)
    if rows is empty:
      sleep(pollInterval)
      continue

    // defined below: only broker-confirmed ids
    ackedIds = publishAked(rows)
    // a crash here leaves confirmed rows published but unmarked:
    // they will be published again, which consumers must tolerate

    // one update for the batch
    markPublished(ackedIds, now())
```

La consulta que hay detrás de `selectUnpublished` es donde se equivocan la mayoría de los relays con polling, así que merece una mirada más de cerca. La forma obvia es “dame las filas con un `id` mayor que el último que traté”. Parece eficiente, y es insegura.

Por qué un cursor sobre el id no es seguro

Los ids de fila suelen asignarse cuando se ejecuta la inserción, no cuando se confirma la transacción. Imagina dos transacciones de Orders. La transacción A inserta una fila del outbox con el id 101 y luego pasa un tiempo llamando a una función lenta antes de confirmar. La transacción B inserta el id 102 y confirma de inmediato. El relay hace polling, ve el 102, lo publica y mueve su cursor al 102. Entonces A confirma. La fila 101 queda ahora detrás del cursor y nunca se leerá.

El fallo es silencioso. Nada da error, no aparece ninguna línea de log y el evento nunca se publica. Es la misma pérdida que el outbox se construyó para eliminar, reintroducida por una optimización.

La solución que prefiero es dejar de recordar una posición y consultar por estado: seleccionar las filas donde `published_at` es nulo, ordenadas por `id`. Una fila que se confirma tarde sigue sin publicar cuando llega el siguiente poll, así que se encuentra. El costo es un índice sobre las filas no publicadas, y una tabla que debe mantenerse lo bastante pequeña para que esa consulta sea barata. Un índice parcial que cubra solo las filas no publicadas mantiene la consulta rápida sin importar cuánto historial tenga la tabla, si tu base de datos admite uno. Si no, el trabajo de retención que veremos más adelante en este capítulo importa más. Un índice parcial no elimina todo el costo: en las bases de datos que conservan versiones antiguas de las filas, cada fila se inserta, se actualiza y luego se borra, y ese vaivén deja filas muertas que el motor debe limpiar.

La lectura basada en estado arregla la pérdida. No arregla el orden, que es una pregunta aparte que responde la sección sobre cómo preservar el orden.

Batching e intervalo de polling

Dos ajustes dan forma al comportamiento del relay: cuántas filas toma por poll y cuánto duerme cuando no hay nada que hacer.

Un lote más grande amortiza el costo de la consulta y de la actualización de marcado, y mejora el throughput. También ensancha la ventana de caída: un relay que muere a mitad de un lote de 500 deja hasta 500 eventos por republicar. Un lote más pequeño republica menos tras una caída y, con la reserva de filas que se describe más abajo, mantiene los bloqueos durante menos tiempo. Digamos que de 50 a 200 filas es una primera estimación razonable; mide antes de afinar.

El intervalo de polling es un mínimo para la latencia. Con un intervalo de un segundo, un evento puede esperar casi un segundo antes de que el relay siquiera lo mire. Acortar el intervalo reduce la latencia y aumenta el número de consultas vacías que la base de datos tiene que atender. Para Parcelly, uno o dos segundos de retraso entre que `OrderPlaced` se confirma y se publica son invisibles para las personas compradoras. Para un flujo en el que una persona espera frente a una pantalla, puede que no lo sean.

Marcar después de publicar

El paso de marcado merece su propio cuidado. Publica cada fila, recoge el acuse de recibo del broker y marca solo las filas que el broker ha confirmado. No marques una fila cuya publicación no se haya confirmado, aunque la llamada haya regresado sin error, porque muchas bibliotecas cliente envían de forma asíncrona y reportan el fallo más tarde.

```
function publishAked(rows):
    ackedIds = []
    for each row in rows:
        result = publishAndWaitAck(topicFor(row.aggregate_type),
        buildEnvelope(row),
                                key: row.partition_key)

        if result is failure:
            // stop here: later rows with the same partition_key must
            not overtake this one
            break
        ackedIds.append(row.id)
    // only these rows may ever be marked
    return ackedIds
```

Esta función nunca toca la tabla. Devuelve los ids que el broker confirmó, y todos los relays de este capítulo marcan exactamente esos ids y ningún otro. Una fila que falló, y toda fila posterior a ella en el lote, queda sin publicar y se vuelve a leer en el siguiente poll. La publicación y el marcado se mantienen separados a propósito, para que el marcado pueda protegerse con un bloqueo o un lease en el único lugar donde ocurre. Cada evento va al topic de su tipo de agregado, con `partition_key` como clave del mensaje, el mismo enrutamiento que usa el Capítulo 4.

El `break` es una decisión deliberada, y tiene que ver con el orden, que vuelve a aparecer más abajo. Saltarse una fila fallida y publicar las filas posteriores es más rápido, y puede reordenar los eventos de un mismo agregado. Detenerse es más lento y seguro. Me detendría, y dejaría que el monitoreo me avise de que el relay está atascado. Este esquema detiene todo el lote por simplicidad; bloquear solo la clave afectada es el refinamiento que se describe más abajo, en la sección sobre cómo preservar el orden.

Un relay activo o varios

La topología más simple es una única instancia de relay que hace todo el trabajo. Su comportamiento de orden es más fácil de razonar, porque lee en orden de `id` y publica en ese orden, con la salvedad sobre el `id` y el orden de commit que se trata en la sección sobre cómo preservar el orden. Su debilidad es la disponibilidad: si esa instancia muere, no se publica nada hasta que arranque otra. Para muchos sistemas, un reinicio en pocos segundos es aceptable, y un solo relay con un supervisor que lo reinicie es la respuesta correcta.

Cuando un relay no da abasto, o cuando una interrupción en la publicación es inaceptable, ejecutas varios. Ahora tienes un problema nuevo: dos relays no deben publicar la misma fila al mismo tiempo. La herramienta estándar es un bloqueo de fila con semántica de omisión. Muchas bases de datos relacionales ofrecen una forma de seleccionar filas para actualizar omitiendo las filas que otra transacción ya ha bloqueado, comúnmente descrita como un bloqueo al estilo `SKIP LOCKED`. El relay reserva un lote dentro de una transacción y los demás relays pasan de largo esas filas.

```
function claimAndPublish():
  transaction:
    rows = selectUnpublished(limit: 100, lockRows: true,
    skipLocked: true)
    ackedIds = publishAked(rows)
    // unconfirmed rows are not marked; releasing the locks lets
    another poll retry them
    markPublished(ackedIds, now())
    // commit point: row locks are released here, and the marks become
    visible
```

Esto funciona, y trae costos que a menudo se pasan por alto. Primero, la transacción permanece abierta mientras el relay habla con el broker, así que un broker lento significa bloqueos retenidos durante mucho tiempo y una pila creciente de transacciones abiertas. En las bases de datos que conservan versiones antiguas de las filas para los lectores concurrentes, una transacción abierta durante mucho tiempo también retrasa la limpieza de las filas muertas, lo que perjudica justo a la tabla con más movimiento. Segundo, si la transacción se aborta después de publicar, por una conexión caída o un marcado fallido, cada fila que publicó se publica de nuevo. Una alternativa más corta es una transacción breve de reserva que marca las filas con una hora de reserva, una publicación fuera de cualquier transac-

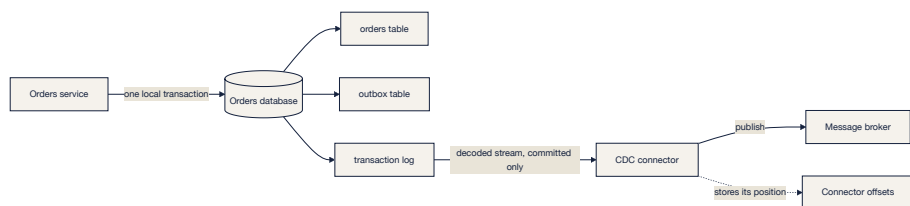
ción y una transacción de marcado aparte, con un timeout de reserva para que las filas de un relay muerto vuelvan a poder reservarse. Cuesta un estado más sobre el que razonar. Tercero, los relays en paralelo pierden el orden global. El relay A puede reservar las filas de la 1 a la 100 mientras el relay B reserva de la 101 a la 200, y B puede terminar primero. Para eventos sobre pedidos distintos, esto no importa. Para dos eventos sobre el mismo pedido, sí.

El remedio es repartir el trabajo por `partition_key` en lugar de por fila. Asigna a cada relay un conjunto de claves, por ejemplo aplicando un hash a `partition_key` sobre un número fijo de buckets y dando cada bucket a exactamente un relay a la vez. Dentro de un bucket, un solo worker trata las filas en orden, y entre buckets se ejecutan en paralelo. Es la misma idea que las particiones en un broker basado en log, aplicada un paso antes. La palabra “exactamente” necesita el lease de la sección de failover de más abajo, y el traspaso de buckets durante un rebalanceo es el momento en que el orden por clave corre más riesgo, porque el responsable anterior y el nuevo pueden publicar a la vez para la misma clave. El Capítulo 9 explica por qué el orden por clave es el orden que realistamente se puede conservar.

Change data capture

El segundo mecanismo no consulta en absoluto la tabla del outbox. El **change data capture (CDC)** lee el propio log de transacciones de una base de datos y convierte los cambios de fila confirmados en un stream de registros de cambio. Un conector de CDC, del cual Debezium es el ejemplo conocido, se engancha a ese log, nota que se insertó una fila en `outbox` y publica un mensaje en el broker.

La tabla conserva su papel, porque la garantía transaccional sigue viniendo de escribirla en la misma transacción que el cambio de estado. Lo que cambia es quién la lee. Ningún proceso de la aplicación hace polling. La base de datos ya registra cada cambio confirmado en un log ordenado para su propia recuperación y replicación, y el conector lee ese log como lo haría una réplica.



Fíjate en el camino desde el log de transacciones hasta el broker. El *change stream* que la base de datos expone a los conectores (PostgreSQL lo llama *logical decoding*, MySQL expone su *binary log*) contiene solo transacciones confirmadas, y aparecen en orden de *commit*. Eso elimina por completo el problema del *commit* tardío de la sección de *polling*: el log ya es un orden total de *commits*, así que una fila que se confirma tarde aparece tarde, en el lugar correcto.

Cómo se comporta

De este diseño se derivan algunas propiedades.

El orden sigue el orden de *commit*. Los eventos aparecen en el orden en que se confirmaron las transacciones. Un conector que publica con la *partition_key* de la fila como clave del mensaje conserva el orden por clave en un broker basado en log, siempre que el productor que lleva dentro esté configurado de modo que un *retry* no pueda reordenar solicitudes que van en vuelo a la vez. Revisa los ajustes de tu cliente; la misma salvedad se aplica a un relay con *polling* que publica de forma asíncrona.

La latencia es baja y estable. No hay intervalo de *polling*. El conector ve un cambio poco después del *commit*, así que el retraso suele ser pequeño y no depende de la frecuencia con la que se despierte nada.

El conector lleva su propia posición. Registra hasta dónde ha leído en el log, y tras un reinicio continúa desde ahí. Los conectores suelen guardar la posición de forma periódica y no después de cada evento, así que una caída puede republicar todos los eventos desde la última posición guardada. El CDC es *at-least-once*, como el *polling*. La ventana de duplicados se ha movido, no ha desaparecido.

La marca de publicado es opcional. Como el conector no necesita encontrar filas sin publicar, la tabla no necesita una columna *published_at* por el bien del relay. Muchos equipos que usan CDC borran cada fila del

outbox poco después de insertarla. La inserción ya quedó registrada en el log, así que el conector la sigue viendo. La tabla se mantiene diminuta, y el problema de la limpieza en gran medida desaparece.

Hay que dar forma al envelope. Un registro de cambio en bruto describe una fila, no un evento. El conector o un paso de transformación debe mapear las columnas del outbox al envelope estándar: `event_id`, `event_type`, `partition_key` como clave del mensaje, `payload` como cuerpo. La mayoría de los conectores admiten enrutamiento y mapeo de campos precisamente para esto. Mantén las columnas del outbox alineadas con el envelope para que el mapeo sea mecánico.

Lo que cuesta

El CDC no es gratis, y los costos son operativos.

Ahora operas un conector, normalmente dentro de un framework de conectores, además de la infraestructura del broker de la que depende. Alguien tiene que aprender cómo falla.

La base de datos tiene que exponer su log al conector, lo que normalmente implica activar un nivel de logging concreto y crear una posición de replicación que la base de datos retiene en nombre del conector. Si el conector se detiene y nadie lo nota, la base de datos conserva segmentos de log que no puede descartar, y el disco se llena. Es una forma de fallo conocida en la familia del CDC, y los detalles difieren según la base de datos. Trataría “el conector está caído” como un page, porque el daño no se limita al broker.

Los cambios de esquema en la tabla del outbox, y en cualquier tabla que vigile el conector, deben planificarse. Un cambio de nombre o de tipo de una columna puede romper el mapeo.

La configuración inicial, los snapshots y los reinicios del conector tras interrupciones largas también merecen un ensayo. El comportamiento depende del producto y de la base de datos, y no lo supondría a partir de un diagrama.

Nota: El CDC es una familia de técnicas, y los productos difieren en cómo leen el log, cómo almacenan las posiciones y qué ocurre tras una interrupción larga. Trata esta sección como la forma general, y lee la documentación de tu conector y de tu base de datos para conocer el comportamiento exacto de las versiones que ejecutas.

Cuando el log ya no está

“El disco se llena” es una mitad de un trade, y la otra mitad es peor. La base de datos o conserva el log para la posición del conector sin límite, lo que llena el disco, o limita lo que conserva, o hace expirar los logs según un calendario, lo que protege el disco. Muchas bases de datos te dejan elegir. Si pones un límite y el conector está caído más tiempo que ese límite, su posición guardada apunta a un log que ya no existe. Los eventos del hueco se pierden, o el conector tiene que empezar de nuevo desde un snapshot.

Lo que un nuevo snapshot le hace a un outbox depende de una decisión anterior. Si la tabla aún contiene filas, un snapshot las reemite todas, así que cada fila retenida se convierte en un duplicado para cada consumidor. Si, como es habitual con CDC, las filas se borraron justo después de insertarse, el snapshot no encuentra nada, y los eventos del hueco no pueden recuperarse de la tabla en absoluto. Eso es una pérdida silenciosa: sin error, sin duplicado, un pedido que nunca llegó a Inventory. El borrado que mantuvo diminuta la tabla es el mismo borrado que elimina la evidencia.

Un failover del primary de la base de datos añade un tercer caso. Que la posición del conector se traslade al nuevo primary varía según la base de datos y la versión, y lo averiguaría ensayando un failover y no leyendo un diagrama.

De esto se desprenden tres hábitos. Elige el límite de retención a propósito, como un trade entre disco y cuánto tiempo de interrupción puedes sobrevivir, y escribe el número. Alerta sobre el margen, es decir, cuánto de la ventana de retención ha consumido ya el retraso del conector, antes de que se agote. Y decide antes de la interrupción cómo recuperarías los

eventos del hueco, por ejemplo con un job de conciliación (Capítulo 13) que compare el estado de los pedidos con lo que se publicó. El Capítulo 14 trata la recuperación una vez hecho el daño.

Elegir entre polling y CDC

Ningún mecanismo es mejor en abstracto. La tabla muestra dónde difieren.

Aspecto	Relay con polling	Conector de CDC
Piezas móviles	Un proceso pequeño que es tuyo	Conector, su framework, ajustes del log de la base de datos
Latencia	Acotada por abajo por el intervalo de polling	Normalmente baja y estable
Fuente del orden	Orden de la consulta más tu propio cuidado	Orden de commit del log
Transacciones que se confirman tarde	Seguro con una consulta basada en estado	Las maneja el log
Carga en la base de datos	Consultas repetidas, bloques de fila	Lecturas del log, segmentos de log retenidos
Duplicados tras una caída	Posibles, la ventana es el lote sin marcar	Posibles, la ventana es la posición sin guardar
Crecimiento de la tabla	Requiere trabajo de retención	Las filas pueden borrarse de inmediato
Fallo que más duele	Retraso silencioso, filas atascadas	Conector detenido, log retenido
Habilidades requeridas	Habilidades corrientes de aplicación	Internos del log de la base de datos, operación de conectores

Mi opción por defecto para un equipo que empieza es un relay con polling. Son unas pocas docenas de líneas de código, el equipo ya sabe cómo operar y depurar un servicio así, y sus modos de fallo son visibles en la tabla que consulta. Pasaría a CDC cuando se cumpla una de tres cosas: la latencia del polling es demasiado alta y no se puede reducir sin saturar la base de datos, el volumen del outbox hace que las consultas de polling sean un costo medible, o ya operas una plataforma de CDC y alguien es responsable de ella.

No adoptaría CDC solo porque suena más serio. Un equipo que nunca ha operado un log de replicación y pone sobre uno el flujo de sus pedidos ha cambiado un problema que entendía por uno que no entiende. Lo contrario también vale: un equipo que ya opera CDC para otros fines gana poco escribiendo además un relay con polling.

Operar el relay

Elegir el mecanismo es la primera mitad del trabajo. La segunda mitad es el trabajo que lo mantiene sano.

Retención y limpieza

Un relay con polling deja filas publicadas a su paso. Si se deja solo, la tabla crece sin límite, la consulta de filas no publicadas se vuelve más lenta y la base de datos gasta esfuerzo en filas que nadie lee. La limpieza hay que diseñarla, no añadirla después.

Importan dos decisiones: cuánto tiempo conservar las filas publicadas y cómo borrarlas. Conservarlas un tiempo tiene valor. Cuando un consumidor reporta un evento faltante, el outbox es el primer lugar donde miras para saber si alguna vez se publicó. Digamos que unos pocos días son un punto de partida razonable. Cuánto es lo correcto depende de lo rápido que tu equipo note los eventos faltantes, que es una pregunta sobre personas y alertas, no sobre almacenamiento.

```
function purgePublishedRows():
  while true:
    deleted = deleteRows(
      where: published_at is not null
        and published_at < now() - retentionPeriod,
      limit: 1000
    )
    // small batches keep locks short and replication lag low
    if deleted is 0:
      return
    sleep(shortPause)
```

Borra en lotes pequeños para que el job no retenga bloqueos durante mucho tiempo ni inunde a las réplicas. Si la tabla es grande, particionarla por tiempo y descartar particiones enteras es más barato que borrar fila por fila, donde tu base de datos lo admita. Elijas lo que elijas, la condición de

purga debe incluir `published_at is not null`. Una limpieza que borre solo por antigüedad borrará, el día en que el relay se quede atrás, eventos que nunca se publicaron. Ese es el fallo del escenario inicial, y es una pérdida de datos, no un duplicado.

Advertencia: Nunca purgues filas del outbox solo por antigüedad. A un relay atascado durante más tiempo que tu período de retención se le borrarán las filas sin publicar, y esos eventos se habrán perdido para siempre. Purga solo las filas marcadas como publicadas, y genera una alerta sobre las filas sin publicar que envejecen.

Con CDC, la limpieza es distinta pero no inexistente. Las filas pueden borrarse justo después de insertarse, así que la tabla se mantiene pequeña. Lo que crece ahora es el log retenido, y lo que hay que vigilar es el retraso del conector frente a él. Borrar de inmediato también significa que un nuevo snapshot no puede recuperar nada, como explica la sección anterior sobre un log desaparecido, así que sopesa el borrado inmediato frente a una retención corta de las filas publicadas.

Monitorear el relay

Un relay que se detiene es invisible desde fuera. Orders sigue teniendo éxito, el outbox acepta filas, y todos los demás servicios se quedan en silencio sin eventos. Nadie recibe un error. Por eso el relay necesita sus propias señales, y pondría tres de ellas antes del lanzamiento.

1. **Antigüedad de la fila sin publicar más vieja.** Es el mejor número individual, porque mide lo que experimentan los usuarios: cuánto tiempo lleva esperando el evento más lento. Genera una alerta cuando supere un umbral atado al negocio, no a la tecnología. Si el correo de confirmación de una compra debería llegar en un minuto, una antigüedad de cinco minutos de la fila más vieja sin publicar es un incidente.
2. **Número de filas sin publicar.** Un backlog que crece indica que el relay es más lento que el productor, aunque la fila más vieja todavía sea joven.
3. **Tasa de errores de publicación y duración de los lotes.** Estas explican las dos primeras. Un relay que reintenta la misma fila en cada poll

mostrará un conteo de errores estable y una antigüedad creciente.

Para un conector de CDC, las mismas preguntas se aplican en otras unidades: cuánto está la posición del conector por detrás de la cabeza del log, si se está ejecutando y cuánto log retiene la base de datos para él. El Capítulo 13 vuelve sobre estas señales y sobre la conciliación, que atrapa lo que las métricas de retraso no ven.

Failover y doble publicación

Un relay debe sobrevivir a sus propios reinicios, y eso incluye el caso en que dos instancias se ejecutan brevemente a la vez. Los deploys, las particiones de red que hacen que la instancia anterior parezca muerta y los orquestadores que reinician un proceso antes de que el anterior haya terminado producen todos este solapamiento.

Para un relay de un único activo, usa un lease: una fila o un bloqueo en la base de datos que el relay activo renueva a intervalos cortos y que expira si el titular se detiene. Un standby toma el relevo tras la expiración. El detalle que hace tropezar a la gente es que el titular anterior puede no saber que su lease ha expirado. Podría quedar pausado por una recolección de basura larga o una red estancada, despertar y publicar un lote que ya no le pertenece.


```

function relayWithLease():
    while running:
        // returns an increasing number
        token = acquireOrRenewLease(ttl: leaseTtl)
        if token is none:
            sleep(standbyInterval)
            continue
        // small: the batch must fit inside the lease
        rows = selectUnpublished(limit: 20)
        ackedIds = publishAkedWithin(rows, token)
        // marking rows the broker confirmed is true even if the lease
        lapsed meanwhile
        markPublished(ackedIds, now())

function publishAkedWithin(rows, token):
    ackedIds = []
    for each row in rows:
        if leaseRemaining(token) < safetyMargin:
            // stop before publishing, not after: a lapsed lease
            cannot be undone
            break
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
            buildEnvelope(row),
                                   key: row.partition_key)
        if result is failure:
            break
        ackedIds.append(row.id)
    return ackedIds

```

El token creciente es un fencing token, y la idea solo funciona donde el recurso protegido lo comprueba. La base de datos puede comprobarlo: un commit que lleve un token obsoleto puede hacerse fallar, y el Capítulo 7 usa exactamente eso para consumidores y APIs. El broker normalmente no puede, así que el token no impide que un relay obsoleto publique. El marcado es otra cuestión. El titular obsoleto marca solo las filas que el broker confirmó, así que sus marcas son verdaderas, y revertirlas solo crearía más duplicados. Lo que un lease debe limitar es la publicación concurrente, porque dos titulares que publican a la vez producen duplicados y pueden poner una fila más antigua de una clave después de una más nueva.

Cuatro cosas lo limitan. Mantén el TTL del lease muy por encima de la duración del peor caso de un lote, y mantén los lotes pequeños, porque el lease se renueva una vez por bucle y un broker lento basta para superarlo. Comprueba el lease restante antes de cada publicación, como hace el esquema. Donde un broker basado en log admita fencing del productor por época, úsalo, para que el propio broker rechace al titular anterior. Y cuan-

do el titular anterior despierte, deja que lo descubra preguntándole a la base de datos, en una transacción que bloquee la fila del lease, antes de que publique nada más. Una comprobación que no retiene el bloqueo es un check-then-act, y falla de la misma manera.

Ese es el límite honesto. Un failover puede producir una doble publicación, y ningún esquema de lease la elimina. Lo que un lease te compra es que la ventana sea corta y poco frecuente. Lo que elimina el daño es un consumidor que trate la segunda copia como el mismo evento, usando el `event_id` que es estable en ambas publicaciones. Por tanto, el relay debe tomar el `event_id` de la fila del outbox y nunca generar uno nuevo cuando publica. Un relay que acuña un identificador nuevo en cada envío convierte un duplicado inofensivo en dos eventos aparentemente distintos. Algunos brokers aceptan un id de deduplicación proporcionado por el productor y descartan las repeticiones dentro de una ventana de tiempo; donde el tuyo lo haga, ponlo en `event_id`. Estrecha la ventana gratis y no reemplaza el inbox del consumidor. El Capítulo 7 trata el lado del consumo.

Preservar el orden

El outbox preserva el orden de los eventos solo hasta donde lo hace el relay, y el relay solo puede preservar el orden que contiene la tabla. El Capítulo 4 mostró que el orden de `id` no es el orden de commit: una transacción puede tomar un `id`, esperar un bloqueo y confirmar después de una transacción que tomó un `id` posterior. El `id` es el orden de inserción, no el orden de commit.

De esto se derivan dos cosas para un relay con polling. Si un `id` menor para una clave se confirma después de que ya se publicó uno mayor para la misma clave, los dos salen invertidos, y la consulta basada en estado que rescató la fila tardía es la que la publica tarde. Si ambos se hacen visibles en un mismo poll con los ids invertidos, salen invertidos en un mismo lote. Por tanto, un relay que publica “en orden de `id`” no entrega los eventos por agregado en el orden en que el agregado cambió, a menos que se cumpla una de dos condiciones.

La primera es que los productores bloqueen la fila del agregado antes de insertar la fila del outbox, de modo que las transacciones de la misma clave se serialicen y sus ids sigan su orden de commit (Capítulo 4). La segunda es que el orden viaje en los datos: la columna `aggregate_version`, que

los consumidores comparan y que usa el Capítulo 9. Haría ambas cosas. El bloqueo es un seguro barato para el relay, y la versión es lo que sobrevive a todo lo que viene después del relay. Un conector de CDC evita el primer problema porque el log está en orden de commit, y aun así no ayuda entre particiones, así que la versión sigue siendo útil.

Dicho eso, las propias decisiones del relay siguen importando. Cada una de estas puede romper el orden que tiene:

- Saltarse una fila fallida y publicar filas posteriores para la misma clave.
- Ejecutar relays en paralelo sin particionar por clave.
- Reintentar una publicación fallida fuera de banda, de modo que una fila posterior llegue primero.
- Usar claves de mensaje distintas para eventos de un mismo agregado.

El valor por defecto seguro es el head-of-line blocking por clave: si una fila para `order-81724` no se puede publicar, deja de publicar las filas posteriores para `order-81724`, y deja que sigan las filas de otras claves. Esto cuesta latencia para el único agregado bloqueado y protege su secuencia. Si el problema es una fila que nunca podrá publicarse, como un payload que el serializador rechaza, alguien tiene que repararla, y el monitoreo de más arriba es la manera en que se entera.

Sé claro sobre lo que hace el esquema anterior. Con un `break` sobre todo el lote y una consulta que va de la más antigua a la más nueva, una fila imposible de publicar se devuelve primero en cada poll, y bloquea todas las claves para siempre. Bloquear por clave necesita dos adiciones, mostradas como un refinamiento de `publishAcked` que reemplaza su `break`: el relay recuerda qué claves fallaron en este lote, y una fila que falla repetidamente se aparca, de modo que la consulta deja de devolverla y también deja de devolver las filas posteriores de su clave.

```

function publishAcked(rows):
    ackedIds = []
    blockedKeys = emptySet()
    for each row in rows:
        if row.partition_key in blockedKeys:
            // an earlier row of this key failed: do not overtake it
            continue
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
        buildEnvelope(row),
                                key: row.partition_key)

        if result is failure:
            blockedKeys.add(row.partition_key)
            // attempts + 1; failed_at = now() after N attempts
            recordFailure(row.id)
            continue
        ackedIds.append(row.id)
    return ackedIds

```

El select excluye entonces toda clave que tenga una fila aparcada, de modo que las filas posteriores de esa clave esperan y las filas de otras claves fluyen. Aparcar es el momento en que se necesita la columna `failed_at` del Capítulo 4, junto con un contador `attempts`, para que un fallo transitorio se reintente unas cuantas veces antes de aparcarse la fila. Una fila aparcada es una cuyo `failed_at` está establecido. Genera una alerta por cada fila aparcada, porque una fila aparcada es un agregado bloqueado y una persona tiene que repararla o descartarla. El costo son dos columnas, una consulta más difícil de indexar y un procedimiento de reparación que alguien debe escribir antes del primer incidente.

El orden también se pierde después del relay, en el consumidor y en el broker. Un relay que cumple su parte te da una secuencia que vale la pena preservar, no una garantía de que sobreviva al resto del camino. El Capítulo 9 sigue el orden desde el broker hasta el handler.

Trade-offs y costos

Ambos mecanismos intercambian simplicidad por alcance operativo, y ambos dejan una ventana de duplicados que no puedes cerrar.

Un relay con polling cuesta carga de base de datos y latencia a cambio de quedarse dentro de código que tu equipo ya entiende. Sus puntos débiles son la consulta de filas sin publicar sobre una tabla que crece, el comportamiento de los bloqueos cuando se ejecutan varios relays y el retraso si-

lencioso cuando nadie vigila la fila más vieja. Si tu sistema publica a un ritmo modesto y un segundo de retraso es aceptable, el relay con polling es difícil de superar en costo de propiedad.

El CDC cuesta un componente nuevo y habilidades nuevas a cambio de menor latencia, lectura en orden de commit y una tabla que se mantiene pequeña. Sus puntos débiles son un conector detenido que hace que la base de datos retenga log, o un log limitado que expira bajo un conector detenido y pierde eventos, los cambios de esquema que rompen mapeos y los procedimientos de recuperación que rara vez se ensayan. Si no tienes a nadie que vaya a ser responsable del conector, no lo adoptaría, por muy atractivo que se vea el diagrama de arquitectura.

Cuándo no usar un relay de outbox en absoluto: si un evento puede perderse sin consecuencias, como una notificación informativa en la que un mensaje perdido no cuesta nada, una publicación directa después del commit puede ser aceptable y más barata. Si tus eventos deben reflejar el estado en una base de datos que no ofrece una transacción local entre el estado y el outbox, el relay no puede reparar eso, y necesitas un diseño distinto.

Dos costos son comunes a ambos. Ninguno te da publicación exactly-once: el duplicado tras una caída entre la publicación y el marcado, o entre la publicación y el guardado de la posición, forma parte del diseño. Y ninguno ayuda si el relay y el productor discrepan sobre el envelope, así que mantén sincronizadas las columnas del outbox y los campos del envelope.

Errores comunes

- **Usar un cursor sobre el id en lugar de una consulta basada en estado.** Es tentador porque evita recorrer en busca de filas sin publicar. Las transacciones que se confirman tarde quedan entonces detrás del cursor y nunca se publican. Consulta por `published_at is null`, o usa un mecanismo basado en log que ya ordene por commit.
- **Marcar las filas antes de que el broker confirme.** Marcar primero parece eliminar los duplicados. Convierte una caída en un evento perdido. Publica, espera el acuse de recibo y luego marca.
- **Generar un nuevo identificador de evento en el relay.** Parece inofensivo, ya que el relay es quien envía. El inbox del consumidor ya

no puede reconocer un evento republicado como el mismo. Copia `event_id` de la fila.

- **Purgar solo por antigüedad.** Mantiene pequeña la tabla y pasa todas las pruebas en las que el relay está sano. Borra eventos sin publicar cuando el relay no lo está. Purga solo las filas publicadas y genera una alerta sobre las sin publicar que envejecen.
- **Saltarse una fila fallida para mantener el throughput.** Las filas posteriores del mismo agregado la adelantan entonces. Bloquea por clave en su lugar, aparca la fila que sigue fallando y arregla la causa.
- **Crear que el orden de `id` es el orden de commit.** Se cumple en toda prueba con un solo escritor. Con escritores concurrentes un `id` menor puede confirmarse después, y el relay publica el par invertido. Bloquea la fila del agregado antes de la inserción en el outbox y lleva `aggregate_version`.
- **Confiar en que el lease aísla al broker.** Un relay obsoleto que despierta después de que su lease expiró sigue publicando, porque el broker no comprueba el token. Mantén los lotes dentro del TTL, comprueba el lease antes de cada publicación y deja que los consumidores absorban el resto.
- **Ejecutar CDC con un límite de retención que nadie eligió.** El valor por defecto parece un detalle hasta que el conector está caído un fin de semana y la posición ya no existe. Elige el límite, genera una alerta sobre el margen y sabe qué le hace un nuevo snapshot a tu outbox.
- **Ejecutar varios relays sin particionar por clave.** Los bloqueos de fila evitan el trabajo doble y no dicen nada del orden. Asigna buckets de claves a workers únicos.
- **No monitorear el relay.** El outbox hace confiable al productor, lo que convierte el fallo del relay en el único que queda, y un relay que se ha detenido se ve exactamente igual que un sistema tranquilo. Genera una alerta sobre la antigüedad de la fila sin publicar más vieja.
- **Tratar una caída del conector como un problema del broker.** En CDC, un conector detenido hace que la base de datos retenga su log. Envía un page, y vigila el disco, no solo el retraso.

Ideas clave

- Un relay hace tres cosas separadas, encontrar, publicar y marcar, y el broker y la base de datos no comparten ninguna transacción, así que una caída entre publicar y marcar siempre deja un posible duplicado.
- Publica primero y marca después: el otro orden convierte una caída en un evento perdido, que es el fallo que el outbox existe para eliminar.
- Un relay con polling debería seleccionar las filas sin publicar por estado, no por un cursor de id creciente, porque las transacciones se confirman en un orden distinto al de la asignación de ids, e incluso así el orden por agregado solo se mantiene si los productores bloquean primero el agregado y llevan un `aggregate_version`.
- Para ejecutar varios relays con polling, reserva las filas con bloqueo al estilo skip-locked y particiona por `partition_key`, de modo que las filas de un agregado las trate siempre en orden un solo worker.
- El change data capture lee los cambios confirmados del log de la base de datos en orden de commit y elimina el polling y el problema del commit tardío, al precio de un conector, un trade de retención del log entre un disco lleno y eventos perdidos, y habilidades de las que tu equipo tiene que hacerse responsable.
- Empieza con un relay con polling a menos que la latencia, la carga o una plataforma de CDC existente digan otra cosa, y trata la elección como un trade de simplicidad frente a alcance operativo.
- Monitorea la antigüedad de la fila sin publicar más vieja, purga solo las filas publicadas, aparca las filas que nunca se publican, copia `event_id` de la fila del outbox y espera duplicados tras un failover, porque un lease limita la publicación concurrente y solo un consumidor idempotente hace inofensivo el resto.

El relay ahora entrega cada evento al menos una vez, en un orden que vale la pena conservar, y a veces dos. El siguiente problema está al otro lado del broker: qué debe hacer un consumidor cuando su trabajo falla, que el Capítulo 6 aborda con los retries.

PARTE III

Consumir con confiabilidad

Capítulo 6 — Retries bien hechos

Imagina una mañana de martes en Parcelly, al comienzo de una promoción para vendedores. Orders envía un comando `CapturePayment` y Payments se lo pasa al proveedor externo de pagos. El proveedor está en apuros y responde con lentitud. Las llamadas de Payments empiezan a agotar el timeout, así que Payments marca el mensaje como fallido y el broker lo reentrega. Todas las instancias de Payments hacen lo mismo, en el mismo instante, con la misma espera fija de un segundo.

En un minuto, el proveedor no recibe la carga normal de la promoción, sino la carga normal más todas las llamadas fallidas que vuelven una y otra vez, en oleadas sincronizadas. La lentitud del proveedor, que podría haber durado unos segundos, ahora dura mientras sigan llegando los retries. Mientras tanto, los consumidores de Payments están ocupados reintentando mensajes que no tendrán éxito, así que los mensajes sanos se acumulan detrás. En algún punto de esa pila, unas cuantas capturas que en realidad sí habían tenido éxito en el proveedor se envían de nuevo.

Nadie escribió una mala línea de código. Cada instancia hizo lo sensato: falló, esperó, lo intentó otra vez. El daño vino de mil decisiones sensatas que nadie coordinó.

Este capítulo trata de convertir un retry en una decisión en lugar de un reflejo. La pregunta que responde es esta: cuando una llamada falla, ¿cómo decides si volver a intentarlo, cuándo, cuántas veces y en qué capa, de modo que reintentar ayude al sistema a recuperarse en lugar de impedirselo?

El problema y por qué es difícil

Un retry es una promesa de volver a intentarlo, no una promesa de tener éxito. En este capítulo un **intento** es cualquier prueba, de modo que los intentos son iguales a uno más el número de retries. El enfoque obvio,

“envolver la llamada en un bucle y reintentar unas cuantas veces”, falla por cuatro razones distintas, y se agravan entre sí.

Primero, **no todo fallo merece un retry**. Una solicitud rechazada porque la tarjeta fue declinada también será rechazada en el décimo intento. Reintentarla desperdicia capacidad y retrasa el momento en que alguien nota el problema real. El bucle de retries tiene que saber qué tipo de fallo tiene delante, y la mayoría de los bucles no lo sabe.

Segundo, **los retries añaden carga en el peor momento**. Una dependencia falla cuando está sobrecargada, degradada o reiniciándose. Los retries son solicitudes adicionales dirigidas exactamente a esa dependencia. Si cada cliente reintenta tres veces, lo que hace cuatro intentos en total, un servicio en apuros ve hasta cuatro veces su tráfico normal justo cuando menos puede permitírselo.

Tercero, **los retries se sincronizan**. Las partes que fallan juntas reintentan juntas. Una espera fija convierte un único fallo en un pico periódico, porque todos los clientes despiertan en el mismo instante. Es el thundering herd, y es la razón por la que la espera necesita aleatoriedad y no solo duración.

Cuarto, **un retry repite una acción cuyo primer resultado puede ser desconocido**. El Capítulo 3 mostró que un timeout no dice si la solicitud fue procesada. Por tanto, reintentar tras un timeout arriesga hacer el trabajo dos veces. Sin idempotencia (Capítulo 7), un retry es un duplicado esperando a ocurrir.

Todo lo demás en este capítulo es una forma de responder a uno de esos cuatro problemas. La clasificación responde al primero. El backoff con jitter y un retry budget responden al segundo y al tercero. La idempotencia, que construye el Capítulo 7, responde al cuarto, y la sección sobre su ausencia, más adelante, explica por qué no se puede omitir.

Clasificar los fallos

Antes de decidir cómo reintentar, decide si hacerlo. El Capítulo 1 dio tres preguntas para hacer a cualquier unión: qué puede fallar, cómo lo observarías y qué garantía sobrevive a ello. Las respuestas juntas son la semántica de un fallo, y una decisión de retry es esa semántica puesta a trabajar.

Me resulta útil ordenar cada fallo en una de tres clases, según lo que dice sobre si volver a intentarlo puede ayudar en absoluto, y luego añadir una cuarta señal que no dice nada sobre el mensaje.

Fallos transitorios

Un **fallo transitorio** es aquel cuya causa se espera que desaparezca por sí sola. Un reinicio de conexión, un timeout, un `503 Service Unavailable`, una víctima de deadlock en la base de datos, una respuesta de throttling, una elección de líder en el broker. La solicitud no estaba mal. El mundo estorbó brevemente.

Reintentar es la respuesta correcta, con una espera, porque la espera es lo que permite que la causa se despeje.

Fallos permanentes

Un **fallo permanente** es aquel que se repetirá en cada intento, porque la causa está en la solicitud o en los datos. Un payload que no pasa la validación, una referencia a un registro que sigue ausente mucho después de cuando debería haber llegado (en un sistema orientado a eventos un registro ausente puede ser transitorio, porque el evento que lo crea puede llegar tarde; el Capítulo 9 trata ese caso), un `400 Bad Request`, una tarjeta declinada por fondos insuficientes, un evento cuya versión de esquema el consumidor no entiende.

Reintentar es esfuerzo desperdiciado y a veces dañino. La respuesta correcta es detenerse, registrar por qué y enviar el mensaje a un lugar donde una persona o un job de reparación pueda ocuparse de él. El Capítulo 8 trata adónde van los fallos permanentes. Fíjate en particular en la tarjeta declinada: para Payments no es un error en absoluto. Es un resultado de negocio esperado que se convierte en un evento `PaymentFailed`, y no interviene ningún retry.

Fallos desconocidos

Un **fallo desconocido** es aquel cuya clase no puedes determinar. Un timeout es el caso clásico, y también lo es una excepción inesperada con un mensaje que nadie anticipó. El timeout pudo ser transitorio. La excepción puede ser un bug del consumidor que fallará en cada intento.

Lo desconocido es una clase real, no un hueco en la taxonomía, y la forma de tratarla es una decisión de política. Mi opción por defecto es tratar los fallos desconocidos como transitorios, pero con un límite más estricto que los transitorios conocidos, porque el costo de reintentar por error un fallo desconocido son unos cuantos intentos desperdiciados, mientras que el costo de descartarlo por error es trabajo perdido. Cuando se agotan los intentos de todas las capas, un fallo desconocido se maneja como uno permanente: va a los dead letters con todo su historial.

Cuando se desconoce el resultado de un intento, pregunta antes de volver a actuar. Antes de un retry, y desde luego antes de una compensación, consulta al proveedor el estado de la operación mediante su clave de idempotencia. Si la captura ocurrió, el seguimiento correcto es un reembolso, no una anulación. Si no ocurrió, un retry o una anulación son seguros. Adivinar en cualquiera de las dos direcciones es la manera en que una persona compradora acaba cobrada por un pedido que Parcelly cree haber cancelado.

Diferidos: se sabe que la dependencia está caída

La cuarta señal no es un veredicto sobre el mensaje. Cuando un circuit breaker (descrito más abajo) está abierto, el fallo de una llamada no dice nada sobre la solicitud: la llamada nunca llegó a la dependencia. Llamo **diferida** a esta clase. Un mensaje diferido no consume un intento, no cuenta para ningún límite y nunca se envía a dead letters por ello. El consumidor deja de extraer mensajes, o pausa la partición, y el mensaje espera donde está hasta que valga la pena volver a llamar a la dependencia.

La distinción importa porque la alternativa es aritmética. Un breaker que permanece abierto durante 30 segundos, delante de un handler que permite tres intentos dentro del handler con un segundo de separación, deja que un mensaje sano gaste todo su presupuesto en aproximadamente los tres primeros segundos y termine en la DLQ por un problema que nunca tuvo.

Hacer explícita la clasificación

La clasificación tiene que vivir en un lugar donde se pueda leer y revisar. Una tabla que asigna a cada señal de fallo una clase y un tratamiento es uno de los artefactos pequeños más valiosos de un consumidor. Para las

llamadas de Payments al proveedor, una versión ilustrativa se ve así:

Señal	Clase	Tratamiento
Reinicio de conexión, fallo de DNS	Transitorio	Retry con backoff
Timeout en una llamada de captura	Desconocido	Retry con la misma clave de idempotencia, límite estricto
429, respuesta de throttling	Transitorio	Retry, respetar la indicación de espera del proveedor si la da
5xx del proveedor	Transitorio	Retry con backoff
400, fallo de validación	Permanente	No reintentar, enviar a dead letters
Tarjeta declinada	Permanente, resultado de negocio	Emitir <code>PaymentFailed</code> , no reintentar
Error no reconocido	Desconocido	Retry con límite estricto, luego a dead letters
Circuit breaker abierto	Diferido	No se consume ningún intento; pausar el consumo, reanudar cuando el breaker lo permita

Dos advertencias. Primera, los códigos de estado son indicios, no garantías; un proveedor que devuelve `500` ante una solicitud mal formada la ha clasificado mal por ti, y tendrás que aprenderlo con la experiencia con ese proveedor. Segunda, la tabla debe tomar por defecto desconocido, nunca permanente ni transitorio, para cualquier señal que no haya visto. Un error nuevo que nadie clasificó no es evidencia de nada.

El clasificador en sí es pequeño:

```
// business outcomes such as a card decline are handled by the caller
// before this function is reached; they are not errors
function classifyFailure(error):
    if error is ValidationError:
        return "permanent"
    if error is CircuitOpen:
        // the call never reached the dependency: no verdict on the
message
        return "deferred"
    if error is ConnectionError or error is Throttled:
        return "transient"
    if error is Timeout:
        // the request may or may not have been processed
        return "unknown"
    return "unknown"
```

Exponential backoff con jitter

Una vez que decides reintentar, la siguiente pregunta es cuándo. Reintentar de inmediato no da tiempo a que una causa transitoria se despeje. Reintentar tras una espera fija le da tiempo, pero mantiene a todos los clientes al mismo paso.

El **exponential backoff** hace que la espera crezca con cada intento: por ejemplo, un segundo, luego dos, luego cuatro, hasta un techo. Los primeros retries son rápidos, lo que conviene a los fallos que se despejan en un instante. Los retries posteriores son lentos, lo que alivia a una dependencia que realmente está caída. El techo evita que la espera crezca más allá de lo que el negocio puede tolerar.

El **jitter** añade aleatoriedad a cada espera. Sin él, las partes que fallaron en el mismo momento calculan las mismas esperas y reintentan al unísono, y la manada sobrevive intacta al backoff. Con él, los retries se reparten a lo largo de un intervalo, y la dependencia ve una mancha de solicitudes en lugar de un muro.

Hay varios esquemas de jitter. Dos conviene conocerlos por su nombre. En el **full jitter**, la espera es un valor aleatorio entre cero y el techo exponencial de ese intento. En el **equal jitter**, la mitad de la espera es fija y la mitad es aleatoria, de modo que la espera mínima nunca queda cerca de cero. El full jitter reparte mejor la carga, mientras que el equal jitter garantiza un mínimo para la espera. Empezaría con full jitter, y pasaría a equal jitter solo si la dependencia sufre por retries muy rápidos.

```

function computeDelay(attempt, baseDelay, maxDelay):
    // the ceiling doubles with each attempt, capped at maxDelay
    ceiling = min(maxDelay, baseDelay * 2 ^ (attempt - 1))
    // full jitter: anywhere between zero and the ceiling
    return randomBetween(0, ceiling)

function callWithRetry(operation, policy):
    attempt = 0
    while true:
        try:
            return operation()
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent":
                raise error
            if failureClass == "deferred":
                // no attempt is counted; the consumer pauses, not
retries
                raise error
            attempt = attempt + 1
            if attempt >= attemptLimit(failureClass, policy):
                raise RetriesExhausted
            sleep(computeDelay(attempt, policy.baseDelay,
policy.maxDelay))

function attemptLimit(failureClass, policy):
    // unknown failures get the stricter limit promised above,
    // counted over all attempts so far, whatever their class
    if failureClass == "unknown":
        return policy.maxAttemptsWhenUnknown
    return policy.maxAttempts

```

Este bucle es correcto ante los fallos que maneja, pero está incompleto en dos aspectos que abordan las secciones siguientes. No sabe cuánto tiempo le queda a quien llama, y no sabe cuántos retries está haciendo ya el sistema entero.

Si el proveedor envía una indicación de cuándo volver a intentarlo, como un valor de espera en una respuesta de throttling, respétala. El proveedor conoce su propia recuperación mejor que tu fórmula. Trata la indicación como un mínimo para tu espera calculada, no como su reemplazo, y conserva el jitter para que los clientes con la misma indicación no vuelvan juntos. Un mínimo sigue sin ser licencia para dormir más allá del deadline: si la indicación es más larga que el tiempo que queda, deja de reintentar y deja que decida la siguiente capa.

Deadlines y retry budgets

El backoff con jitter decide el espaciado de los retries. Otros dos límites deciden cuántos puede haber.

Deadlines

Un **deadline** es el momento a partir del cual el resultado ya no es útil. Un flujo dirigido por solicitudes tiene uno natural: la persona compradora está mirando una página de pago y dentro de 10 segundos se irá. Un consumidor en segundo plano tiene uno más blando: una etiqueta de envío que llega después del corte del transportista llega tan tarde como una que nunca llega.

La lógica de retries que cuenta intentos pero ignora el tiempo puede pasarse mucho de lo útil. Tres intentos con un timeout de 30 segundos cada uno y esperas crecientes entre ellos son minutos de espera. Prefiero dar a cada operación reintentada un presupuesto total de tiempo, y dejar de reintentar cuando la siguiente espera lo cruzaría, diga lo que diga el conteo de intentos.

```
function callWithDeadline(operation, policy, deadline):
    attempt = 0
    while true:
        remaining = deadline - now()
        if remaining <= 0:
            raise DeadlineExceeded
        try:
            // never let one attempt outlive the whole budget
            return operation(timeout: min(policy.attemptTimeout,
remaining))
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent" or failureClass ==
"deferred":
                raise error
            attempt = attempt + 1
            delay = computeDelay(attempt, policy.baseDelay,
policy.maxDelay)
            // the next attempt needs time to run, not only time to
start
            needed = delay + policy.minimumUsefulAttempt
            if attempt >= attemptLimit(failureClass, policy) or needed
>= deadline - now():
                raise RetriesExhausted
            sleep(delay)
```


El deadline también debe viajar. Si Orders da 10 segundos a un flujo y llama a Payments, Payments debe transmitir lo que queda, no empezar con 10 segundos nuevos propios. De lo contrario, cada capa reintenta dentro de la paciencia de la capa anterior, y la suma excede en silencio lo que el usuario está dispuesto a esperar.

Retry budgets

Un límite de intentos por llamada no protege a una dependencia cuando muchas llamadas fallan a la vez. Cada llamada se mantiene dentro de su límite de tres retries, y aun así la dependencia recibe cuatro veces su tráfico normal.

Un **retry budget** limita los retries como proporción del tráfico total, no por llamada. Una forma común es un token bucket: cada llamada exitosa añade una fracción de token, cada retry gasta uno, y cuando el bucket está vacío las llamadas fallan rápido sin reintentar. En régimen estable, los retries son una pequeña fracción de las solicitudes. En una caída generalizada, el presupuesto se agota en un instante y el sistema deja de amplificar su propio fallo.

Los números los eliges tú y dependen de cuánta carga adicional pueda absorber la dependencia. No ofreceré una cifra, porque cualquier cifra que diera sería una conjetura sobre tu dependencia. Lo que importa es que el presupuesto exista, que se comparta entre quienes llaman dentro de un proceso y que su agotamiento sea visible en un dashboard. Un presupuesto basado en una proporción y mantenido por proceso suele bastar, porque si cada proceso limita su propia proporción ya se limita el total. Un presupuesto compartido entre instancias es una opción, no una mejora estricta: necesita un contador distribuido, y ese contador es una nueva dependencia en la ruta de fallo, con un modo de fallo propio.

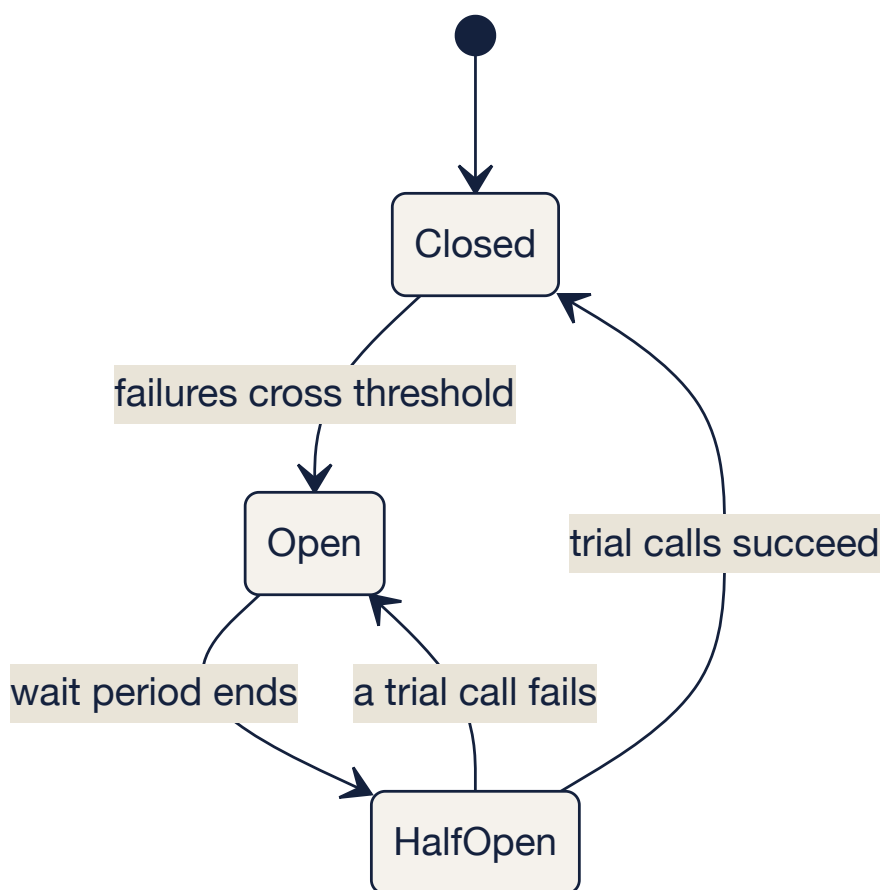
El precio es que algunas llamadas que habrían tenido éxito con un retry fallarán. Ese es el sentido de un presupuesto: cambia éxito individual por supervivencia del sistema, y debe hacerlo a propósito.

Retry storms y el circuit breaker

Un **retry storm** es lo que ocurre cuando reintentar se convierte en la carga dominante sobre una dependencia que falla. El escenario inicial de Parcelly fue uno. Necesita tres ingredientes: una dependencia lenta o que falla, clientes que reintentan, y ningún mecanismo que diga a esos clientes que se detengan. Las capas lo empeoran. Si Orders hace tres intentos contra Payments, y Payments hace tres intentos contra el proveedor, una sola captura fallida puede convertirse en nueve llamadas al proveedor. Con otra capa de tres intentos se convierten en 27.

Dos defensas abordan la tormenta directamente. El retry budget de arriba limita cuántos retries hará el sistema. El circuit breaker limita si debería llamar siquiera.

Un **circuit breaker** observa los resultados de las llamadas a una dependencia y tiene tres estados. Está cerrado en operación normal, con las llamadas pasando. Cuando los fallos cruzan un umbral, se abre, y las llamadas fallan de inmediato sin tocar la dependencia, dándole margen para recuperarse. Tras una espera, pasa a semiabierto y deja pasar unas pocas llamadas de prueba. Si tienen éxito, se cierra. Si no, se abre de nuevo.



El breaker y la política de retries son complementarios y responden a preguntas distintas. El backoff decide cuánto esperar antes del siguiente intento de una llamada. El breaker decide, a partir del patrón entre llamadas, si vale la pena hacer intentos en absoluto. Un breaker abierto convierte el siguiente intento del bucle de retries en un fallo instantáneo. Ese fallo no es transitorio ni desconocido: es la señal **diferida** de la clasificación anterior. El bucle de retries no debe contarla, ni aplicarle backoff, ni girar sobre ella. Devuelve el control al consumidor, que decide qué dejar de hacer.

Un consumidor tiene una opción extra de la que carece quien llama en respuesta a solicitudes. Cuando el breaker está abierto, un consumidor debe dejar de extraer mensajes del broker, o pausar la partición, en lugar de consumir mensajes y hacerlos fallar uno a uno. Los mensajes que permanecen en el broker están a salvo mientras lo permita su retención, cuestan almacenamiento y aparecen como retraso del consumidor. Los mensa-

jes que se consumen, se reintentan y fallan gastan sus conteos de intentos, y podrían ir a dead letters por un problema que nunca fue suyo. Pausar es la forma del breaker del lado del consumidor, y yo lo construiría antes que la mayoría de lo demás en este capítulo.

Pausar plantea dos preguntas de seguimiento. Primera, sin tráfico circulando, nadie prueba el estado semiabierto. Reanuda con una concurrencia de uno, o envía una llamada de sondeo sintética, para que las llamadas de prueba las haga algo que sabe que son pruebas. Segunda, en un broker basado en log un consumidor en pausa debe mantener vivo su bucle de poll, pausando las particiones pero no el polling, o el grupo decidirá que el consumidor está muerto y reasignará sus particiones (la ventana de liveness descrita más abajo).

El costo de un breaker es que puede fallar por error. Los umbrales demasiado sensibles se abren ante un parpadeo breve y convierten un fallo transitorio en una caída. Los umbrales demasiado permisivos se abren demasiado tarde. Un breaker también añade estado que alguien debe entender cuando un flujo deja de funcionar a las tres de la mañana: la primera pregunta es si la dependencia está caída o solo el breaker está abierto.

Retries bajo sobrecarga: backpressure

Un breaker reacciona ante una dependencia que ha fallado. El **backpressure** es la disciplina para la etapa anterior, cuando una dependencia está lenta, saturada o recuperándose, y el sistema debe decidir cuánto trabajo entregarle. El Capítulo 3 lo sitúa dentro del modelo de fallos. Esta sección es el mecanismo.

Vuelve al escenario inicial. La lentitud del proveedor no vino de un bug. Vino de más trabajo del que podía absorber, y cada instancia de Payments siguió alimentándolo. Un consumidor sin límite de trabajo en vuelo se comporta exactamente así: sigue extrayendo, sigue iniciando llamadas y sigue añadiendo a la pila que ya es demasiado pesada. Los límites siguientes son los que yo pondría, aproximadamente en este orden.

- **Acotar el trabajo en vuelo.** Limita el número de mensajes que un consumidor retiene a la vez. En un broker basado en colas, eso es el prefetch o el conteo en vuelo. En un broker basado en log, es el número de registros por poll más el número de handlers en ejecución. Sin

tope, la memoria del consumidor y el reloj de reentrega del broker se convierten en el límite, y ambos son peores límites que uno elegido por ti.

- **Reducir la concurrencia cuando la dependencia se ralentiza.** Una concurrencia que es correcta con 50 milisegundos por llamada es incorrecta con cinco segundos por llamada, porque el mismo número de espacios ahora contiene muchas veces más trabajo en espera. Baja el número de llamadas concurrentes cuando sube la latencia, y súbelo de nuevo, despacio, cuando la latencia se recupera.
- **Pausar en lugar de fallar.** Cuando la dependencia está saturada, deja de extraer, como con un breaker abierto. Un consumidor en pausa se manifiesta como retraso, que es visible y seguro. Un consumidor que extrae y falla gasta intentos.
- **Atender el tráfico vivo antes que los retries.** Los retries y el redrive compiten con los primeros intentos por la misma dependencia. Cuando la capacidad es escasa, deben ganar los primeros intentos, porque un retry es trabajo que el sistema ya sabe que está teniendo problemas. Ejecuta los consumidores de retry topics y de redrive con menor concurrencia, o déjalos esperar mientras el flujo principal tenga backlog.
- **Descartar carga en la entrada.** En algún punto aguas arriba tiene que existir un límite al trabajo aceptado, o la cola crece sin límite mientras la dependencia se recupera. Rechaza o difiere el trabajo nuevo, con una señal clara a quien llama, en lugar de encolarlo indefinidamente detrás de una dependencia que no puede atenderlo.

Estos límites y el retry budget son dos vistas de una misma idea. El presupuesto limita cuánto trabajo extra añaden los retries, y el backpressure limita cuánto trabajo, extra o no, está en vuelo. Ninguno basta solo: un presupuesto sin tope en vuelo deja que una dependencia lenta llene todos los espacios con primeros intentos, y un tope en vuelo sin presupuesto deja que los espacios se llenen de retries.

El precio es retraso y rechazo. Un sistema que descarta carga dice que no a algunos clientes, a propósito, y un sistema que pausa entrega algunos mensajes tarde. Ambos son un cambio de throughput por supervivencia, y quien es responsable de la respuesta a “qué trabajo espera” es la persona

responsable de negocio del flujo, no quien ajusta los números. El Capítulo 8 aplica la misma idea de rate limit al redrive, y el Capítulo 9 aplica el límite en vuelo a los lanes con clave.

Dónde viven los retries

Un mensaje puede reintentarse en varias capas, y las capas se comportan de manera distinta. Mezclarlas sin saberlo es como un mensaje termina reintentado muchas más veces de las que nadie pretendía, o ninguna.

Retry a nivel de aplicación

El consumidor reintenta dentro del handler: el código de arriba, envuelto alrededor de una llamada al proveedor. El mensaje permanece sin confirmar mientras corren los retries. Esta capa es barata, rápida y buena para fallos que se despejan en milisegundos o segundos. Sus límites son que retiene la capacidad del consumidor mientras espera, que una caída del consumidor pierde el estado del retry (el broker reentregará, y el conteo de intentos empieza de nuevo, por lo que un mensaje que hace caer el proceso necesita la detección que describe el Capítulo 8) y que las esperas largas arriesgan exceder los propios timeouts del broker.

En un broker basado en colas, el mensaje sin confirmar queda oculto durante el **visibility timeout**, tras el cual se reentrega. Si los retries dentro del handler duran más que ese timeout, el broker reentrega mientras la primera copia aún se está reintentando, y ahora hay dos copias en ejecución. En un broker basado en log, un consumidor que pasa demasiado tiempo entre polls puede ser considerado muerto por su grupo de consumidores y ver reasignadas sus particiones, con el mismo resultado. La regla que yo seguiría: el tiempo total más largo de los retries dentro del handler debe quedar cómodamente dentro del timeout del broker, y todo lo más largo pertenece a otra capa.

Reentrega del broker

El consumidor hace fallar el mensaje y el broker lo entrega de nuevo. Las dos familias de brokers difieren aquí, y la diferencia importa.

En un broker basado en colas, la reentrega viene incorporada. El consumidor hace `ack` del mensaje, o deja que expire el `visibility timeout`, y el mensaje vuelve a la cola, de inmediato o tras una demora configurada, donde cualquier consumidor competidor puede recibirlo.

Un broker basado en log no reentrega por sí solo. La posición de lectura del consumidor avanza a medida que hace `polling`, y el `offset` confirmado solo registra dónde reanudaría un consumidor reiniciado (el Capítulo 2 lo trata). Por tanto, fallar al procesar un registro y no confirmar su `offset` no hace nada por sí mismo: el consumidor lee el siguiente registro, y si después confirma, esa confirmación cubre también el registro fallido, que se omite en silencio. El registro fallido se vuelve a leer solo tras un reinicio, un rebalanceo o un paso explícito del consumidor.

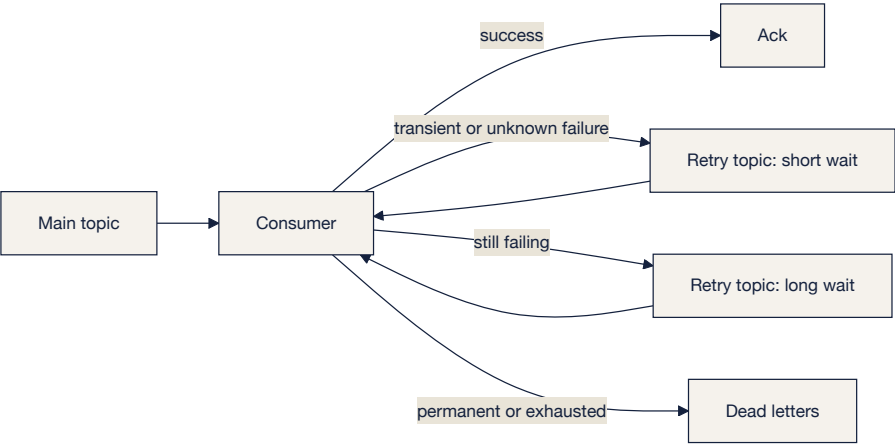
Por tanto, en un log el consumidor tiene que elegir una de tres acciones. Puede retroceder (`seek`) al `offset` fallido, o pausar la partición, y reintentar en el sitio, lo que preserva el orden y detiene todo lo que hay detrás. Puede publicar el registro en un `retry topic` y seguir adelante. O puede escribirlo en un `dead-letter topic` y seguir adelante. Cada una es una decisión deliberada en el código del consumidor, no algo que el broker haga por ti.

La reentrega del broker es duradera: sobrevive a una caída del consumidor. También es tosca. Muchos brokers reentregan de inmediato o con un calendario fijo, sin `backoff`, y el conteo de intentos puede estar disponible solo como un contador de entregas que el consumidor tiene que leer y atender. Trátala como la red de seguridad bajo las demás capas, no como la política principal de `retries`.

Retry topics y colas de demora

Cuando un fallo necesita una espera más larga de la que puede sostener el `retry` dentro del handler, una estructura común es sacar el mensaje del flujo principal y dejar que vuelva más tarde. Un consumidor que hace fallar un mensaje lo publica, con un conteo de intentos incrementado y una hora de no antes de, en un **retry topic** (o cola de `retries`), y confirma el original. Un consumidor aparte, o el mismo en un `topic` distinto, lo recoge cuando llega la hora. Los brokers que admiten entrega retardada, o demoras por mensaje, lo ofrecen directamente. Donde no, un arreglo común es una escalera de `retry topics`, cada uno con una espera fija, uno por cada paso del `backoff`.

Las esperas son fijas por topic por una razón. En un log, un consumidor no puede saltarse un registro que aún no le toca: debe esperar en la cabecera hasta que el registro esté listo. Si todos los registros de un topic tienen la misma espera, el topic ya está en orden de vencimiento, y esperar en la cabecera no cuesta nada extra. Un topic que mezclara demoras cortas y largas retendría las cortas detrás de las largas. Las horas de no antes de por mensaje funcionan solo en brokers con entrega retardada nativa.



Esta capa libera el flujo principal, de modo que un mensaje que falla ya no detiene a los mensajes que van detrás. Sobrevive a los reinicios, porque el estado vive en el broker. También te da un lugar donde mirar: la profundidad del retry topic es una medida directa de cuánto trabajo espera a una dependencia que se recupera.

Cuesta más maquinaria. Hay más topics que crear, nombrar, monitorear y proteger. La republicación es en sí misma una operación de publicación que puede fallar, lo que trae de vuelta el dual write del Capítulo 1, así que el orden importa: publica primero en el retry topic, luego confirma el original. Si el proceso muere entre ambos pasos, el mensaje existe dos veces. Eso es un duplicado, que el consumidor debe tolerar de todos modos.

Tres reglas evitan que la escalera dañe aquello que debe proteger:

- **Un topic propio por consumidor.** Un retry topic pertenece a un grupo de consumidores o suscripción. Si otro grupo puede leerlo, ese grupo reprocesa un mensaje que nunca falló.

- **Preservar la identidad.** El mensaje republicado conserva la clave original, el `event_id` y los headers de negocio. Una clave nueva rompe el orden, y un `event_id` nuevo rompe el inbox.
- **Dimensionar la retención según la espera.** La retención de un retry topic debe superar la espera más larga más el retraso que esperes en él, o el mensaje expira mientras espera.

El costo más importante es el **orden**, que trata la sección siguiente.

Elegir las capas

En la práctica, un consumidor usa las tres. Un retry breve a nivel de aplicación absorbe los parpadeos, como un reinicio de conexión o un deadlock, que se despejan en uno o dos segundos. Un retry topic o una cola de demora cubre los fallos que necesitan minutos. La reentrega del broker cubre las caídas. Una dead-letter queue (DLQ) cierra la cadena y recibe lo que ha agotado sus intentos o nunca podrá tener éxito; el Capítulo 8 trata su diseño.

Las capas deben diseñarse juntas, con un único presupuesto total de intentos, no tres independientes. Si cada capa permite tres intentos, un mensaje puede probarse 27 veces, y nadie planeó eso. La política del final de este capítulo hace explícita la multiplicación y la limita.

Retries y orden

Reintentar un mensaje cambia el orden en que se procesan los mensajes. Esto se sigue de la estructura en capas anterior, y pasa inadvertido hasta que causa daño.

Considera el consumidor de Inventory de Parcelly, que recibe `OrderPlaced` y luego `OrderCancelled` para el mismo pedido, en ese orden. Supón que el procesamiento de `OrderPlaced` falla de forma transitoria y el consumidor lo mueve a un retry topic, y luego sigue con `OrderCancelled`. La cancelación no encuentra ninguna reserva que liberar, porque, hasta donde Inventory sabe, todavía no existe ninguna. Luego llega el `OrderPlaced` reintentado y crea una reserva que nada liberará jamás. El estado final es incorrecto, y cada paso individual parecía razonable.

Hay dos opciones honestas, y cada una tiene un precio.

- **Bloquear y reintentar en el sitio.** En un broker basado en log, el consumidor retrocede al offset que falla, o pausa la partición, y lo reintenta, con backoff, antes de tocar nada de lo que hay detrás. Se preserva el orden de esa partición. El precio es que toda la partición espera detrás de un mensaje, de modo que un solo mensaje fallido puede detener a todos los agregados de esa partición. Es la elección correcta cuando el orden importa más que el throughput, y es la razón por la que clasificar rápido un fallo permanente importa tanto: un poison message en la cabecera de una partición bloquea todo lo que hay detrás.
- **Apartar y aceptar el reordenamiento.** El mensaje fallido va a un retry topic y el consumidor continúa. Se preserva el throughput. El precio es que el consumidor debe tolerar la llegada desordenada, mediante técnicas como las comprobaciones de versión, o bien aparcando también todo mensaje posterior del mismo agregado hasta que el anterior tenga éxito. El Capítulo 9 trata estas técnicas.

Elegiría por stream. Para un stream de eventos en el que los mensajes de un agregado deben aplicarse en orden, reintentaría en el sitio e invertiría en una clasificación rápida y una salida a dead letters. Para un stream de hechos independientes, donde el orden entre mensajes no importa, apartar es barato y mantiene el flujo en movimiento.

En los brokers basados en colas sin una función de agrupación, el orden entre consumidores competidores ya es débil de entrada, así que los retries empeoran ligeramente una propiedad existente en lugar de romper una garantía. Aun así, un consumidor que supone orden no debe confiar en que una cola se lo proporcione. Si usas una función de agrupación (colas FIFO, grupos de mensajes, sesiones), el trade-off es el mismo que en un log: un mensaje fallido bloquea su grupo hasta que tiene éxito o va a dead letters, igual que un retry en el sitio bloquea una partición.

Retries sin idempotencia

Cada retry es un intento repetido de un trabajo cuyo resultado anterior puede ser desconocido. Cuando el intento anterior falló antes de hacer nada, el retry es inofensivo. Cuando falló después de hacer el trabajo, el

retry hace el trabajo dos veces.

Esto no es un caso límite raro. Considera el escenario inicial. Payments llama al proveedor para capturar 4,990 unidades menores de EUR. El proveedor captura el dinero y envía su respuesta, y la respuesta se pierde. Payments ve un timeout. El timeout se clasifica como desconocido, y la política dice que se reintente. Sale la segunda solicitud de captura. Que la persona compradora sea cobrada una o dos veces depende ahora por completo de si algo hace que la segunda solicitud sea reconocible como una repetición.

Ese algo es una **clave de idempotencia**: un identificador estable, enviado con la solicitud, que permite al receptor reconocer que ya ha visto este trabajo. Para la llamada al proveedor, es la misma clave en cada intento de la misma captura, que un proveedor que respete esas claves tratará como una sola captura. Para consumir eventos, normalmente es el `event_id`, comprobado contra el inbox en la misma transacción que el trabajo. El Capítulo 7 construye ambas.

Advertencia: Una política de retries es segura solo sobre una operación idempotente. Antes de habilitar retries en cualquier llamada que tenga un efecto, como cobrar una tarjeta, enviar un correo o reservar stock, establece qué hace inofensivo el segundo intento. Si nada lo hace, el número correcto de retries automáticos es cero, y el fallo debe ir a una persona. Dado que un broker at-least-once reentregará tras una caída o un timeout quiera lo que quiera el handler, “cero” tiene que imponerlo el handler: debe detectar que ya ha visto este mensaje, o confirmar antes del efecto y aceptar at-most-once, y luego enrutar el fallo a una persona.

La dependencia funciona en ambos sentidos. La idempotencia es lo que hace seguros los retries, y los retries son lo que hace necesaria la idempotencia. Un sistema que reintenta agresivamente sin consumidores idempotentes producirá efectos duplicados a un ritmo proporcional a su tasa de fallos, es decir, en los momentos en que el sistema ya está bajo estrés y nadie tiene tiempo de limpiar.

Hay también un caso más sutil. Una operación puede ser idempotente para la misma clave y aun así no ser segura de reintentar, porque la clave cambia. Si un handler construye un identificador nuevo en cada intento, como un UUID nuevo en cada llamada, el proveedor ve solicitudes distintas cada vez. La clave debe derivarse una sola vez, de entradas estables como el `event_id` o el pedido, y reutilizarse en cada intento.

Una política de retries para Parcelly

Para reunir las piezas, esto es lo que yo dejaría escrito para el consumidor de `CapturePayment` de Payments. Los números son ilustraciones, no recomendaciones, y tu dependencia necesitará los suyos.

```
retryPolicy for CapturePayment:
  // illustrative values
  attemptTimeout: 5 seconds
  inHandler:
    maxAttempts: 3
    maxAttemptsWhenUnknown: 2
    baseDelay: 200 milliseconds
    maxDelay: 2 seconds
    jitter: full
  retryTopics:
    steps: [1 minute, 5 minutes, 30 minutes]
    appliesTo: transient and unknown failures; the deadline
caps both
  // after this, a capture is stale
  totalDeadline: 2 hours
  // all layers and redeliveries together
  maxTotalAttempts: 8
  retryBudget: per process, visible on a dashboard
  circuitBreaker: pause consumption when open; deferred calls cost
no attempt
  onExhausted: dead-letter with the full attempt history
  onStale: ask the provider for the capture's status, then
           void if not captured, refund if captured
  idempotencyKey: derived once from command_id, reused on each
attempt
```

Léela como un conjunto de decisiones, cada una con su razón. La capa dentro del handler absorbe los parpadeos breves. Los retry topics cubren un proveedor caído durante minutos. El deadline dice cuándo el trabajo deja de valer la pena, y quien decide eso es el negocio, no la ingeniería. Una captura obsoleta puede tener que convertirse en un `OrderCancelled`, pero la compensación depende de lo que realmente ocurrió en el provee-

dor. Si el último intento agotó el timeout, la captura puede haber tenido éxito, y entonces el seguimiento correcto es un `PaymentRefunded`, no un `PaymentVoided`. Por eso la política pide primero el estado. La clave, derivada una sola vez, es lo que evita que toda la estructura cobre dos veces a una persona compradora.

Dos líneas de la política merecen una segunda mirada. `maxTotalAttempts` existe porque las capas se multiplican: tres intentos dentro del handler, por una primera pasada más tres pasos de retry topic, son hasta 12 llamadas al proveedor, antes de contar cualquier reentrega tras una caída. Un total único de 8 (ilustrativo) es el número del que alguien realmente es responsable. Y todo el horizonte de retries, dos horas aquí, debe ser más corto que el tiempo que el proveedor recuerda las claves de idempotencia. Los proveedores conservan las claves durante una ventana limitada, y un retry que llega después de la ventana es una solicitud nueva con una clave vieja.

Cuando el redrive del Capítulo 8 devuelve una captura muerta, el contador de intentos empieza de nuevo, pero permanece un contador de redrive aparte, de modo que el historial no se pierde y un mensaje que sigue volviendo se ve como tal.

Fíjate en lo que falta: un único ajuste de “conteo de retries”. En un sistema con capas, ese único número esconde al menos cuatro decisiones.

Trade-offs y costos

Los retries son una de las formas más baratas de mejorar la confiabilidad aparente de un sistema, y una de las más fáciles de usar en exceso. Sus costos son específicos.

Latencia. Cada retry añade tiempo de espera. El backoff y el jitter reparten la carga, pero también alargan el tiempo hasta el éxito. Un flujo que reintenta durante minutos puede completarse correctamente y aun así ser inútil para la persona compradora que se fue tras 10 segundos. Donde una persona espera, favorecería un fallo rápido y un mensaje claro sobre un retry largo y silencioso.

Amplificación. Los retries multiplican la carga, y los retries en capas la multiplican de nuevo. Una política que es razonable en cada servicio puede ser peligrosa en la cadena. Alguien tiene que ser responsable del presupuesto de intentos de extremo a extremo, porque ningún servicio por sí solo puede verlo.

Capacidad. Un consumidor que reintenta en el sitio ocupa su espacio mientras espera. Durante una caída larga, todos los consumidores pueden terminar esperando, y el sistema deja de avanzar incluso en el trabajo que tendría éxito. Pausar el consumo mediante un breaker, acotar el trabajo en vuelo con backpressure y mover las esperas largas a retry topics son las defensas habituales.

Fallos ocultos. Un retry que al final tiene éxito parece un éxito desde fuera. Si no cuentas retries ni intentos, una dependencia puede degradarse durante semanas detrás de una política que la absorbe en silencio. El Capítulo 13 trata qué medir: los conteos de intentos, la tasa de retries y la antigüedad del mensaje reintentado más viejo pertenecen a un dashboard.

Complejidad. Los retry topics, las colas de demora y los presupuestos son infraestructura que alguien debe construir, nombrar, monitorear y explicar. Para un sistema pequeño con un consumidor y una dependencia, un retry dentro del handler con backoff y una salida a dead letters puede ser todo lo que necesitas, y una escalera de retry topics puede ser más maquinaria de la que merece el problema.

Cuándo no reintentar. No reintentes fallos permanentes. No reintentes operaciones con efectos que no sean idempotentes. No reintentes pasado el deadline. No reintentes cuando un circuit breaker dice que la dependencia está caída, y no cobres a ese mensaje un intento por ello. No reintentes dentro de una capa que ya está dentro del retry de otra capa, a menos que hayas calculado la multiplicación. Y no reintentes como una forma de evitar una decisión: un fallo que sigue volviendo es información, y la respuesta correcta suele ser detenerse y mirar.

Errores comunes

Reintentar todo de la misma manera

Es tentador porque un solo bucle es el código más corto. Trata una tarjeta declinada como un reinicio de conexión y desperdicia capacidad en fallos que no pueden tener éxito. Clasifica primero y da a cada clase su propio tratamiento.

Esperas fijas sin jitter

Una espera constante parece predecible y simple. Sincroniza a todas las partes que fallaron juntas en un pico que se repite. Usa exponencial backoff con jitter, y, en mi opinión, full jitter como punto de partida.

Contar intentos pero no tiempo

Un límite de intentos parece una cota de seguridad. Con timeouts largos y esperas crecientes todavía puede prolongarse minutos más allá de lo útil. Fija un deadline para toda la operación y detente cuando la siguiente espera lo cruzaría.

Reintentar en cada capa

El responsable de cada capa añade un retry sensato, y la multiplicación es invisible para todos ellos. Decide un presupuesto total de intentos para la cadena, y pon los retries donde menos cuesten.

Reintentar mientras el breaker está abierto

Es tentador mantener el bucle en marcha, porque el bucle funcionaba. Un consumidor que extrae mensajes y los hace fallar contra una dependencia caída consume sus intentos y puede enviarlos a dead letters por error. Trata el breaker abierto como una señal diferida que no cuesta ningún intento, pausa el consumo y deja que los mensajes esperen en el broker.

Dejar que un consumidor extraiga sin límite

Un consumidor sin acotar es el valor por defecto, y da la sensación de máximo throughput. Contra una dependencia lenta, llena todos los espacios con trabajo en espera y alimenta el retry storm. Acota los mensajes en vue-

lo, baja la concurrencia cuando sube la latencia y deja que los primeros intentos ganen a los retries.

Ignorar los timeouts del broker

Los retries dentro del handler parecen locales al handler. Si duran más que el visibility timeout o que la ventana de liveness del grupo, el broker reentrega o reasigna, y dos copias del trabajo corren a la vez. Mantén el tiempo total dentro del handler bien dentro de los límites del broker.

Reintentar pasando por encima de un mensaje que debe mantener el orden

Mover un mensaje fallido a un retry topic es una solución rápida para una partición detenida. Reordena los eventos de ese agregado. Decide primero el requisito de orden, y luego elige entre bloquear y apartar.

Reintentar efectos no idempotentes

El bucle de retries lleva una tarde añadirlo, y la idempotencia se pospone. El primer timeout tras una llamada exitosa produce un efecto duplicado. Establece qué hace inofensivo el segundo intento antes de habilitar el primer retry.

Generar una clave nueva en cada intento

Parece diligencia, ya que cada solicitud queda identificada de forma única. Frustra el propósito, porque el receptor no puede reconocer una repetición. Deriva la clave una sola vez de entradas estables y reutilízala.

No contar los retries

Si solo se registran los fallos finales, una dependencia que se degrada es invisible hasta que los retries dejan de bastar. Registra cada intento, con su clase y su espera.

Ideas clave

- Un retry es una promesa de volver a intentarlo, no una promesa de tener éxito, así que la primera pregunta ante cualquier fallo es si volver a intentarlo puede ayudar en absoluto.

- Clasifica los fallos como transitorios, permanentes o desconocidos, deja la clasificación escrita como una tabla y trata los fallos no reconocidos como desconocidos con un límite más estricto.
- Usa exponential backoff con jitter, acota toda la operación con un deadline y comparte un retry budget para que los retries no se conviertan en la carga dominante sobre una dependencia que falla.
- Un circuit breaker decide si llamar siquiera; un breaker abierto es una señal diferida que no cuesta ningún intento, y para un consumidor la forma más fuerte es pausar el consumo. El backpressure hace el mismo trabajo antes, acotando el trabajo en vuelo y dejando que el tráfico vivo gane a los retries y al redrive.
- Los retries pueden vivir en el handler, en la reentrega del broker y en retry topics, y deben diseñarse juntos bajo un único presupuesto de intentos, con el tiempo dentro del handler contenido dentro de los timeouts del broker.
- Apartar un mensaje fallido reordena los eventos de su agregado; elige entre bloquear en el sitio y tolerar el reordenamiento según el requisito de orden del stream.
- Una política de retries es segura solo sobre operaciones idempotentes, con la clave de idempotencia derivada una sola vez y reutilizada en cada intento.

Los retries deciden con qué frecuencia vuelves a intentarlo, pero no pueden decidir si intentarlo dos veces es inofensivo. El Capítulo 7 construye la otra mitad: el consumidor idempotente, que convierte un duplicado de un desastre en un no-evento.

Capítulo 7 — Idempotencia

Imagina el servicio Ledger de Parcelly en mitad de un rolling deploy, mientras sus consumidores se reinician de uno en uno. Un consumidor ha leído un evento `PaymentCaptured`, ha escrito una entrada del libro mayor por 4,990 unidades menores de EUR y está a punto de confirmar el mensaje cuando el proceso recibe su señal de apagado. El ack nunca sale de la máquina. El broker, haciendo exactamente lo que promete la entrega at-least-once, entrega el mismo mensaje al siguiente consumidor que lo pida.

El segundo consumidor lee el evento, no encuentra nada fuera de lo común y escribe una segunda entrada en el libro mayor. El libro mayor es de solo añadir y correcto por construcción, así que ahora registra, fielmente y de forma permanente, que la persona compradora pagó dos veces. Nada se ha caído. Ninguna alerta ha saltado. Cada componente se comportó como fue diseñado, y los libros están mal.

Notifications tiene su propia versión de la misma tarde. Una persona compradora recibe tres veces el correo “tu paquete está programado”, porque el mensaje fue reentregado después de que un handler lento superara su visibility timeout. Ese caso es vergonzoso. El del libro mayor es caro.

Los capítulos anteriores establecieron que los duplicados son normales: el broker reentrega, el relay publica dos veces tras un failover y los retries del Capítulo 6 son duplicados por diseño. Este capítulo responde la pregunta que esos capítulos dejan abierta. Cuando el mismo mensaje llega dos veces, ¿qué debe hacer exactamente un consumidor para que la segunda llegada no cambie nada?

El problema y por qué es difícil

El objetivo suena modesto: procesar cada mensaje una vez, hasta donde el mundo pueda ver. La dificultad es que el consumidor no puede saber si ha visto un mensaje antes sin recordarlo, y recordar es en sí mismo una escritura que puede fallar en el peor momento.

Empieza por el origen de los duplicados, porque la lista es más larga que el broker.

Origen	Cómo aparece el duplicado
Reentrega del broker	Se perdió el ack, expiró el visibility timeout o un rebalanceo movió la partición antes de confirmar el offset
Relay	Un relay publicó una fila, se cayó antes de marcarla como publicada y la publicó de nuevo tras reiniciarse (Capítulo 5)
Retry de la aplicación	Un consumidor reintentó una llamada, o un productor reintentó una publicación cuyo primer intento en realidad había tenido éxito (Capítulo 6)
Redrive y replay	Alguien devolvió dead letters a su cola, o releyó deliberadamente un log de eventos desde un offset anterior
Personas y clientes	Una persona compradora hizo doble clic en “Pagar”, una app móvil reintentó en una red inestable, el emisor de un webhook entregó dos veces

La mayoría de estos casos producen el *mismo* mensaje dos veces, con el mismo `event_id`. Unos pocos producen dos mensajes distintos que expresan la misma intención, y esa distinción importará cuando elijamos una clave.

La solución tentadora es mirar antes de actuar. Preguntar si este mensaje ya ha sido procesado y, si no, procesarlo. Quiero darle nombre como el antipatrón **check-then-act**, porque parece correcto y falla en silencio.

```
function handlePaymentCaptured(message):
  // UNSAFE: the check and the action are separate steps
  if inboxContains("ledger-poster", message.event_id):
    ack(message)
    return

  postLedgerEntry(message.payload)
  inboxInsert("ledger-poster", message.event_id)
  ack(message)
```

Léelo bajo fallo. Si el proceso muere después de `postLedgerEntry` y antes de `inboxInsert`, la entrada existe y el inbox no lo sabe, así que la reentrega la publica de nuevo. Invierte las dos escrituras y el fallo se invierte también: una caída después de `inboxInsert` y antes de la escritura en el libro mayor marca el mensaje como manejado cuando no lo fue, y el efecto se pierde para siempre. Ningún orden es seguro, porque las dos escrituras no son un único acto atómico.

Hay un segundo fallo que no tiene nada que ver con las caídas. Dos consumidores pueden recibir el mismo mensaje casi en el mismo instante, por ejemplo cuando un primer intento lento supera su `visibility timeout`. Ambos ejecutan la comprobación, ambos no encuentran nada y ambos siguen adelante. La ventana entre “comprobar” y “actuar” es pequeña, y con volumen suficiente se alcanza de forma rutinaria.

Así que el problema real tiene dos mitades. La memoria de lo que se procesó debe escribirse de forma atómica con el efecto, y la memoria debe poder arbitrar entre intentos concurrentes. El resto del capítulo construye ambas, empezando por el caso en que no necesitas ninguna.

Idempotencia natural y forzada

Una operación es **idempotente** cuando aplicarla dos veces deja el sistema en el mismo estado que aplicarla una. Algunas operaciones son idempotentes por su forma, y yo buscaría estas antes de construir cualquier maquinaria.

Fijar un valor es idempotente: `set order.status = "PAID"` puede ejecutarse 10 veces y el pedido sigue siendo PAID. Borrar una fila por id es idempotente. Insertar una fila con una clave primaria que derivas del evento, e ignorar el conflicto, es idempotente. Escribir el estado nuevo completo de un agregado (“el envío ahora está `DISPATCHED`, con este número de seguimiento”) es idempotente. A esto lo llamo **idempotencia natural**: el duplicado no tiene efecto extra por lo que la operación es, no por nada que hayas añadido.

Otras operaciones no lo son. `balance = balance + 4990` aplicada dos veces son dos depósitos. `send email` envía dos correos. `append a ledger entry` añade dos. `increment the counter of units sold` cuenta la unidad dos veces. Llamar al endpoint de captura de un proveedor de pagos sin una clave puede cobrar dos veces. Para estas, la idempotencia hay que añadirla, lo que llamo **idempotencia forzada**: registras que el trabajo se hizo y consultas el registro antes de volver a hacerlo.

La idempotencia natural tiene dos límites honestos. El primero es que depende en silencio del orden. Fijar `status = "PAID"` es idempotente, pero un duplicado retrasado de un evento más antiguo puede sobrescribir un estado más nuevo: un `OrderPlaced` reentregado que llega después de

`OrderConfirmed` y devuelve el estado a `PLACED`. La operación es idempotente y el sistema sigue estando mal. El Capítulo 9 trata el orden y las comprobaciones de versión; por ahora, observa que “fijar a un valor” es seguro frente a duplicados solo cuando el valor no puede quedar obsoleto.

El segundo límite es que resulta tentador sobreestimarla. Un handler que “solo actualiza una fila” a menudo también escribe un registro de auditoría, emite un evento de seguimiento o incrementa un contador, y cada uno de ellos es un incremento disfrazado. Cuando reviso un consumidor, enumero cada efecto que tiene y pregunto de cada uno por separado si sobrevive a una segunda ejecución. Un solo efecto no idempotente basta para necesitar el inbox.

Mi regla práctica es preferir la idempotencia natural donde el efecto lo permita, y dejar escrito en el handler por qué lo permite. Cuando el efecto es un contador, una adición, un correo o una llamada externa, usa la forzada. Y ante la duda, usa el inbox de todos modos: su costo es pequeño y predecible, y el costo de equivocarse no lo es.

Elegir la clave de idempotencia

Una **clave de idempotencia** es lo que el consumidor usa para reconocer trabajo repetido. Antes de construir la tabla tienes que decidir qué entra en ella, y la elección determina qué duplicados puedes atrapar.

El `event_id`

Para el consumo de eventos, el valor por defecto es el `event_id` del envelope. Es único por evento y, de forma crucial, **estable entre reentregas**: cada copia del mensaje que te entrega el broker lleva el mismo valor. El productor lo asigna una sola vez, cuando el evento se escribe en el outbox (Capítulo 4), de modo que un relay que publica dos veces publica dos veces el mismo `event_id`. Esa es exactamente la propiedad que necesita una comprobación de deduplicación.

El `event_id` identifica un *mensaje*. Responde a “¿he manejado antes este hecho concreto?” y nada más amplio.

La clave de negocio

A veces dos mensajes distintos describen la misma intención. Supón que Orders reintenta un comando `CapturePayment` tras un timeout y, por un bug o una decisión de diseño, cada retry se escribe como un comando nuevo con un `command_id` nuevo. Payments recibe ahora dos comandos con ids distintos que significan ambos “capturar el pago del pedido 81724”. Deduplicar por el id del mensaje no atrapa ninguno.

El remedio es una **clave de negocio**: un identificador derivado del significado del trabajo y no del envelope. Para la captura podría ser el `payment_id` más la operación, por ejemplo `capture:payment-5521`. Para una entrada del libro mayor podría ser el id del pago más el tipo de entrada. Una clave de negocio dice “esta unidad de trabajo puede ocurrir una vez”, sin importar cuántos mensajes la pidan.

Aspecto	<code>event_id</code> / <code>command_id</code>	Clave de negocio
Atrapa la reentrega del mismo mensaje	Sí	Sí
Atrapa dos mensajes con la misma intención	No	Sí
Quién la asigna	El productor, automáticamente	Una decisión humana por operación
Riesgo	Pasa por alto duplicados lógicos	Una clave demasiado gruesa rechaza trabajo legítimo
Dónde se almacena	Tabla genérica de inbox	A menudo una restricción única en la tabla del dominio

El riesgo de la penúltima fila merece un caso concreto. Si la clave de negocio de los reembolsos es solo `payment_id`, un segundo reembolso parcial, legítimo, del mismo pago se rechaza como duplicado. La clave debe incluir todo lo que hace que dos solicitudes sean *genuinamente distintas*: aquí, quizá una clave de idempotencia del reembolso suministrada por quien llama. Elegir la clave significa decidir qué es “el mismo trabajo” en tu dominio, lo cual es una decisión de diseño y no un ajuste de framework.

En mi opinión las capas funcionan bien juntas. Usa el `event_id` en un inbox genérico como base para todos los consumidores, y añade una restricción única de clave de negocio en la tabla del dominio dondequiera que

dos mensajes distintos pudieran significar razonablemente lo mismo. El inbox protege contra el broker y el relay. La restricción protege contra el resto del sistema, incluidos sus bugs.

El patrón inbox

El **inbox** es la tabla de ids de mensajes procesados del lado del consumidor. El Capítulo 4 dio al productor un transaccional outbox para que un cambio de estado y su evento se confirmen juntos. El inbox es la imagen especular: el consumidor confirma su cambio de estado y su registro de haber manejado el mensaje juntos, en una sola transacción local en su propia base de datos.

```
table inbox:
  consumer      text not null
  event_id      uuid not null
  processed_at  timestamp not null
  primary key (consumer, event_id)
```

La columna `consumer` permite que un servicio ejecute varios consumidores independientes contra una misma tabla, de modo que el consumidor de contabilización de Ledger y su consumidor de informes recuerden cada uno su propio historial. Por eso la clave primaria es el par (`consumer`, `event_id`). Solo con `event_id`, el primer consumidor en manejar un mensaje haría fallar el insert del segundo consumidor, y el efecto del segundo se omitiría como si fuera un duplicado. Un servicio con un único consumidor por tabla puede prescindir de la columna y usar `event_id` como clave, que es la forma más simple usada en otras partes del libro.

El handler escribe la fila del inbox y el cambio de negocio en la misma transacción.

```
function handlePaymentCaptured(message):
    if inboxContains("ledger-poster", message.event_id):
        // fast path only; this check is not the safety net
        ack(message)
        return

    transaction:
        postLedgerEntry(message.payload)
        // primary key (consumer, event_id): a concurrent duplicate
        // fails here, the transaction rolls back, and the
        // message is redelivered and finds the committed row
        inboxInsert("ledger-poster", message.event_id)
    // commit point: effect and inbox row exist together or not at all

    ack(message)
```

Recorre los fallos. Si el proceso muere antes del commit, ni la entrada del libro mayor ni la fila del inbox existen, así que la reentrega procesa el mensaje desde cero. Si muere después del commit y antes del ack, ambas existen, y la reentrega toma el camino rápido: la comprobación encuentra la fila, se confirma el mensaje y no se escribe nada. El efecto y su memoria nunca pueden discrepar, porque se confirman como una unidad.

La comprobación del principio es una optimización. Ahorra una transacción en el caso común de duplicado, y puede eliminarse sin perder corrección. La red de seguridad es la restricción única dentro de la transacción, que es donde ocurre el arbitraje real.

Dentro de una transacción, el orden de las escrituras no cambia la atomicidad, así que la advertencia sobre invertir las dos escrituras se aplica solo a la versión insegura de arriba. Sí cambia el costo. En esta versión, un duplicado concurrente hace todo el trabajo antes de fallar en el insert, y en algunas bases de datos una violación aborta toda la transacción, de modo que el handler no puede recuperarse dentro de ella. Reclamaría primero: inserta la fila del inbox con semántica de “on conflict do nothing”, mira si se insertó una fila y omite el efecto si no. El duplicado entonces espera en la clave, no hace trabajo y nunca lanza una excepción. El ejemplo de la sección siguiente usa esa forma.

Una palabra sobre `ack`, porque el término esconde una diferencia entre las dos familias. En un broker basado en colas confirma un mensaje. En un broker basado en log suele ser un commit de offset, lo que significa “todo hasta aquí”, y un consumidor que procesa una partición en paralelo puede

confirmar más allá de un mensaje que no ha terminado, y perderlo. El inbox protege contra duplicados, no contra eso, y el Capítulo 9 lo examina donde el orden y el paralelismo se encuentran.

Eventos de seguimiento desde un consumidor

Muchos consumidores hacen más que cambiar su propio estado: anuncian lo que hicieron. Ledger contabiliza una entrada y emite

`LedgerEntryPosted`. Ese evento de seguimiento es un efecto como cualquier otro, y tiene que sobrevivir a una caída exactamente una vez, ni cero veces ni dos.

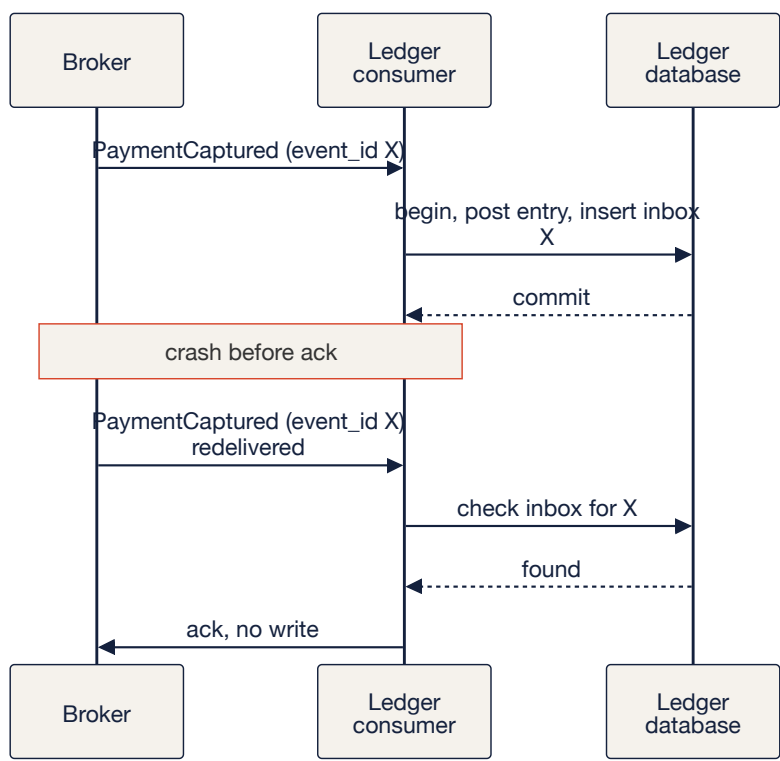
Un handler que publica el seguimiento directamente después de su commit lo pierde de forma permanente ante una caída. La reentrega toma el camino rápido del inbox, encuentra la fila, confirma y omite la publicación. El estado es correcto y el evento nunca existió. La solución es el outbox del Capítulo 4 aplicado del lado del consumo: la fila del inbox, el cambio de estado y el insert en el outbox se confirman en una sola transacción local.

```
function handlePaymentCaptured(message):
  transaction:
    claimed = inboxInsertIfAbsent("ledger-poster",
    message.event_id)
    // on conflict do nothing: a concurrent duplicate waits on the
    key,
    // then sees claimed = false and does no work
    if claimed:
      postLedgerEntry(message.payload)
      outboxInsert({
        event_type:    "LedgerEntryPosted",
        causation_id:  message.event_id,
        partition_key: message.partition_key
        // remaining envelope fields as in Chapter 4
      })
    // commit point: inbox row, ledger entry and follow-up event, or
    none of them
    ack(message)
```

Si el proceso muere antes del commit, no existe nada y la reentrega lo hace todo. Si muere después del commit, el seguimiento está en el outbox y el relay lo publica, mientras la reentrega encuentra la fila del inbox y no hace nada. El outbox del consumidor tiene la misma publicación at-least-

once que el del productor, así que quien consuma `LedgerEntryPosted` necesita su propio inbox. El Capítulo 10 se apoya en esta regla cuando un paso de una saga reacciona a un evento y emite el siguiente.

Esta es también la razón por la que el inbox vive en la *misma base de datos* que el estado que protege. Si el registro de mensajes procesados estuviera en un cache o en otra base de datos, volveríamos a tener dos escrituras sin transacción compartida, que es el dual write del Capítulo 4 del lado del consumo. El inbox no es una técnica que se pueda acoplar desde fuera del almacén de datos del servicio.



Mira la nota del medio: la ventana de caída está entre el `commit` y el `ack`, y la fila del inbox es lo que convierte la reentrega en una operación nula.

Duplicados concurrentes

La secuencia anterior supone que el duplicado llega después de que el primer intento haya terminado. La concurrencia es más difícil, y la restricción única es lo que te salva.

Imagina dos consumidores procesando el mismo mensaje a la vez. Ambos inician una transacción. Ambos hacen el trabajo. Ambos intentan insertar la fila del inbox. La base de datos deja que exactamente uno haga commit; el otro recibe una violación de la restricción única, su transacción hace rollback y su entrada en el libro mayor se desvanece con ella. El perdedor deja entonces el mensaje sin confirmar, o lo confirma tras volver a comprobar, y en cualquier caso el libro mayor contiene una sola entrada.

Ese es todo el mecanismo, y vale la pena decir lo que no requiere: ningún bloqueo distribuido, ninguna coordinación entre los dos consumidores, ninguna suposición sobre tiempos. La restricción de la base de datos es un único punto de arbitraje por el que deben pasar ambos intentos.

```
function handleSafely(message):
  try:
    transaction:
      applyChange(message)
      inboxInsert("ledger-poster", message.event_id)
      // commit point; a losing concurrent attempt rolls back here
      ack(message)
    on UniqueViolation as e:
      if e.constraint == "inbox_pkey":
        // someone else committed this message first; the work is
done
        ack(message)
      else:
        // a business constraint failed: a real error, not a
duplicate
        raise e
```

Si una violación única debe confirmarse o reintentarse es una decisión pequeña que vale la pena tomar deliberadamente. Si la violación es sobre la clave del inbox, el mensaje ha sido manejado por una transacción confirmada y confirmarlo es lo correcto. Si es sobre alguna otra restricción dentro de `applyChange`, el error significa algo muy distinto, y confirmarlo ocultaría un bug real. En mi opinión el handler debe distinguirlas, lo cual puede hacer comprobando el nombre de la restricción violada, como hace el código de arriba.

Handlers más largos y trabajo en curso

El patrón anterior mantiene todo el handler dentro de una transacción. Eso está bien cuando el trabajo son unas pocas escrituras. Es incómodo cuando el handler llama a algo lento, porque la transacción permanece

abierta, los bloqueos se mantienen y el visibility timeout puede expirar mientras esperas. El mensaje expirado se reentrega a un segundo consumidor, que inicia el mismo trabajo lento en paralelo.

Hay dos respuestas sólidas. La primera es mantener corta la transacción: hacer el trabajo externo lento *fuera* de ella, con el enfoque de efectos externos de la sección siguiente, y poner dentro de la transacción solo el cambio de estado local y la fila del inbox. La segunda, cuando el trabajo realmente debe reclamarse, es hacer explícito el reclamo con un estado.

```
table inbox:
  consumer      text not null
  event_id      uuid not null
  status        text not null    // "processing" or "done"
  lease_until   timestamp null
  lease_token   bigint not null  // increases on every takeover
  primary key (consumer, event_id)
```

Un consumidor inserta la fila como `processing` con un lease, y ese reclamo se confirma en su propia transacción corta antes de que empiece el trabajo lento. Un duplicado que encuentra un lease vigente no debe confirmar el mensaje, porque el primer intento aún puede fallar y la confirmación borraría el único retry. Lo reencola con una demora, mayor que el lease restante, y vuelve a mirar en la siguiente entrega. Uno que encuentra un lease vencido puede tomar el relevo con un lease token más nuevo, que el titular original debe comprobar antes de confirmar sus cambios locales; uno que encuentra `done` confirma.

El token protege solo lo que puede comprobarlo. La transacción local puede: bloquea la fila del inbox, compara el token y hace rollback si existe uno más nuevo. La llamada externa lenta no puede deshacerse, así que el efecto recibe la clave determinista de la sección siguiente, y un titular obsoleto que despierta y repite la llamada es reconocido por el sistema aguas abajo. Esta es la idea de fencing del lease del relay del Capítulo 5, con el mismo límite: un fence funciona solo donde el recurso protegido comprobaba el token.

El esquema funciona, y añade una máquina de estados sobre la que alguien tiene que razonar, incluida la pregunta de qué pasa cuando un worker muere sosteniendo un lease. Recurriría a él solo cuando el primer enfoque no esté disponible, y mantendría el lease corto y renovable.

Advertencia: No confíes en un `SELECT` previo ni en un bloqueo de fila para impedir duplicados. Que un segundo consumidor vea o no el trabajo sin confirmar del primero depende del nivel de aislamiento de la base de datos, y una comprobación que es correcta bajo un nivel falla bajo otro. Subir el nivel a serializable tampoco elimina la necesidad de la restricción: sustituye un duplicado silencioso por un fallo de serialización, que el handler debe tratar como reintentable y volver a ejecutar desde el principio. Los bloqueos reducen la contención; la restricción única dentro de la transacción es la garantía.

Ventanas de deduplicación y retención

Un inbox que conserva todos los ids para siempre crece para siempre. En algún momento querrás borrar filas antiguas, y borrar cambia la garantía de una manera que merece un nombre: la **ventana de deduplicación**. Un mensaje cuya fila de inbox ha sido borrada puede procesarse de nuevo si alguna vez reaparece.

Entonces, ¿cuánto debe durar la ventana? Tiene que cubrir el mayor tiempo en que un duplicado podría llegar razonablemente después del original. No conozco ese número para tu sistema, y depende de una lista corta de cosas.

- **Retención del broker y horizonte de reentrega.** Una cola que reentrega durante cuatro días, o un log que retiene una semana, significa que pueden llegar duplicados así de tarde.
- **Práctica de dead letters y redrive.** Si un equipo puede hacer redrive de dead letters una semana después de que fallaron, la ventana debe superar eso (Capítulo 8).
- **Replay.** Releer un log de eventos desde un offset antiguo crea duplicados de eventos tan antiguos como los que reproduzcas. O bien la ventana es lo bastante larga, o bien un replay se ejecuta contra un consumidor deliberadamente reiniciado, que es un procedimiento que diseñas con antelación.
- **Failover del relay.** Las publicaciones dobles tras un failover suelen estar cerca en el tiempo del original, así que quedan muy dentro de cualquier ventana que elijas.

Una forma común es un período de retención cómodamente más largo que la retención propia del broker, digamos unas semanas a modo de ilustración, con un job programado que borra por lotes. Particionar la tabla del inbox por tiempo hace barato el borrado, porque descartar una partición entera cuesta mucho menos que borrar millones de filas.

Ten presente que el inbox no es la única línea de defensa. Una restricción de clave de negocio en la tabla del dominio vive tanto como los datos, así que protege contra duplicados que llegan una vez desaparecida la fila del inbox. Esta es otra razón para usar ambas capas: el inbox da protección barata y de memoria acotada contra el caso común, y la restricción cubre la cola.

Si el sistema puede aceptarlo, hay una alternativa a una ventana de tiempo: derivar la clave de algo monótono. La regla es más estrecha de lo que parece al principio. Se aplica a los eventos que portan estado, el tipo que el Capítulo 9 distingue de los deltas, donde cada evento contiene el estado nuevo completo de un agregado. Para esos, el consumidor almacena la versión más alta del agregado aplicada y rechaza todo lo que no sea más nuevo. El offset es una posición dentro de una partición y la versión es una propiedad de un agregado, así que no son intercambiables: un offset no dice nada de los agregados que viven en otras particiones.

Para los eventos delta, “rechazar si no es más nuevo” es incorrecto, porque una versión omitida significa un cambio perdido. Un consumidor así aplica la versión `last + 1`, retiene en un buffer una posterior hasta que se llene el hueco y trata como duplicado todo lo que esté en `last` o por debajo; el Capítulo 9 describe el buffering y sus límites. En ambos casos la memoria está acotada sin TTL, y depende de garantías de orden que el Capítulo 9 examina. Conviene a los agregados con un contador de versión y no conviene a los consumidores alimentados desde varias particiones.

Efectos secundarios externos

El inbox protege los efectos que viven en tu propia base de datos. Muchos efectos no lo hacen. Un correo sale de tu sistema en el momento en que lo envías, y una llamada al proveedor de pagos cobra una tarjeta tanto si tu transacción se confirma después como si no. Ninguna transacción local puede deshacer esto, así que el truco de la atomicidad no se aplica.

Considera a Payments manejando un comando `CapturePayment`. Debe llamar al proveedor externo y luego registrar el resultado.

```
function handleCapturePayment(command):  
    // unsafe for the crash we care about: the provider call has no  
    key  
    result = provider.capture(command.payload.payment_id)  
    transaction:  
        recordCapture(command.payload.payment_id, result)  
        inboxInsert("payments-capture", command.command_id)  
    ack(command)
```

Ante una caída entre la llamada al proveedor y la transacción, el dinero se ha movido y Payments no tiene registro. En la reentrega el handler vuelve a llamar al proveedor. Que la persona compradora sea cobrada dos veces depende ahora por completo del proveedor.

Pasar la clave aguas abajo

La solución es llevar la clave de idempotencia al sistema que realiza el efecto, para que pueda reconocer la repetición. Los proveedores de pagos suelen admitirlo: envías una clave con la solicitud, y una segunda solicitud con la misma clave devuelve el primer resultado en lugar de actuar de nuevo. Los detalles del header o del campo varían según el proveedor, y yo leería la documentación de tu proveedor para conocer su comportamiento exacto y cuánto tiempo recuerda las claves.

El detalle que más importa es *cómo eliges la clave*. Debe ser **determinista a partir del trabajo, no generada por intento**. Una clave aleatoria creada dentro del handler se ve idéntica a una solicitud nueva en cada retry, y no protege nada.

```

function handleSafely(command):
    try:
        if inboxContains("payments-capture", command.command_id):
            // local memory, kept as long as the inbox row: the long-
term guard
            ack(command)
            return

            key = "capture-" + command.command_id
            // same command, same key, on every attempt and every
redelivery
            result = provider.capture(command.payload.payment_id, key)
            // an "in progress" or conflict answer for a concurrent
request with
            // this key is a retryable failure, not a result to record

            transaction:
                recordCapture(command.payload.payment_id, result)
                inboxInsert("payments-capture", command.command_id)
            // commit point
            ack(command)
        on UniqueViolation as e:
            if e.constraint == "inbox_pkey":
                // a concurrent attempt recorded it first
                ack(command)
            else:
                raise e

```

Ahora rastrea de nuevo los fallos. Una caída después de la llamada al proveedor y antes del commit deja el dinero movido y a Payments sin saberlo. El comando reentregado ejecuta el mismo código con la misma clave, el proveedor devuelve el resultado original y la transacción lo registra. Una caída durante la llamada al proveedor, donde se pierde la respuesta, es la misma situación: el retry lleva la misma clave, y el proveedor o bien realiza la captura una vez o bien devuelve el resultado anterior. Dos consumidores en carrera sobre el mismo comando envían la misma clave a la vez, y un proveedor que rechaza el segundo como “en curso” no ha hecho fallar el trabajo: el handler trata esa respuesta como reintentable y vuelve a mirar en la siguiente entrega.

La memoria del proveedor y el inbox no son la misma protección. Un proveedor recuerda las claves durante un tiempo limitado, y yo leería su documentación para saber cuánto. Una reentrega que llegue después de ese período haría una segunda captura. La fila local del inbox, conservada al

menos tanto como la ventana de deduplicación descrita antes, es lo que impide que el handler llame siquiera al proveedor, de modo que la retención de claves del proveedor nunca es la ventana en la que confías.

Si el trabajo se identifica por el negocio y no por el mensaje, usa también aquí una clave derivada del negocio, como el id del pago más la operación. Así, incluso dos comandos con ids distintos llegan al proveedor como una sola solicitud.

Cuando el sistema aguas abajo no tiene clave

No todo sistema aguas abajo acepta una. SMTP simple no tiene clave de idempotencia, y muchos proveedores de correo y SMS no ofrecen una de forma confiable. Entonces te enfrentas a una elección entre dos resultados imperfectos, y yo la haría explícitamente, por efecto.

Si envías antes de registrar, una caída puede llevar a un correo duplicado. Si registras antes de enviar, una caída puede llevar a un correo faltante. Son comportamientos at-least-once y at-most-once del efecto. Para una notificación de “tu paquete está programado”, un duplicado raro es barato y un mensaje perdido es un ticket de soporte, así que enviaría primero y aceptaría la repetición ocasional. Para un código de cupón de un solo uso que otorga valor, me inclinaría por lo contrario y añadiría una forma de detectar y reparar un envío faltante.

Algunos efectos pueden hacerse más seguros registrando la *intención* en la transacción local y realizando el efecto desde un worker aparte, que es la idea del outbox aplicada a los comandos. No elimina el problema en la frontera con el proveedor, porque el worker aún puede caerse entre la llamada y la marca. Sí mueve el problema a un solo lugar donde puedes verlo, reintentarlo deliberadamente y conciliarlo. El Capítulo 13 trata la conciliación: comparar tus registros con los del proveedor y reparar las diferencias, que es la defensa final para exactamente los efectos que no pueden hacerse idempotentes.

Idempotency-Key y las APIs HTTP

El mismo problema aparece en tu propia puerta de entrada. El teléfono de una persona compradora envía “realizar pedido”, la red pierde la respuesta y la app reintent. Orders no puede distinguir un retry de un segundo

pedido a menos que el cliente lo indique.

La convención más extendida es un header de solicitud `Idempotency-Key`: el **cliente** genera un valor único para cada operación lógica y envía el mismo valor en cada retry de ella. Esta es la otra mitad de la definición dada antes: para el consumo de eventos la clave es normalmente el `event_id`, que asignó el productor, y para las solicitudes de cara al cliente la suministra quien llama. El trabajo del servidor es hacer que una clave repetida se comporte como la primera solicitud.

Un servidor necesita más que la clave. Almacena la clave junto con un fingerprint de la solicitud y la respuesta que produjo.

```

function handleCreateOrder(request):
    key = request.header("Idempotency-Key")
    if key is missing:
        return error(400, "Idempotency-Key required")
    try:
        transaction:
            record = idempotencyFindForUpdate(request.caller, key)
            if record is none:
                token = idempotencyInsert(request.caller, key,
fingerprint(request.body),
                                                    "in_progress", lease_until:
dbNow() + LEASE)
            else if record.fingerprint != fingerprint(request.body):
                return error(422, "key reused with a different
request")
            else if record.status == "completed":
                return record.stored_response
            else if record.lease_until > dbNow():
                return error(409, "request in progress")
            else:
                // the holder crashed or stalled: take over with a
newer token
                token = idempotencyTakeOver(request.caller, key,
lease_until: dbNow() + LEASE)
                // commit point: the key is claimed, fenced by an increasing
token
                on UniqueViolation as e:
                    if e.constraint == "idempotency_pkey":
                        // a concurrent first claim won
                        return error(409, "request in progress")
                    raise e
                transaction:
                    // locks the row; raises LeaseLost
                    requireLeaseToken(request.caller, key, token)
                    response = createOrderWithOutbox(request.body)
                    idempotencyComplete(request.caller, key, response)
                // commit point: on LeaseLost the order and its outbox rows roll
back
                return response

```

La comprobación del fingerprint importa. Un cliente que reutiliza una clave para un cuerpo distinto tiene un bug, y devolver en silencio la primera respuesta lo ocultaría. El estado en curso importa para el caso de concurrencia: dos solicitudes simultáneas con una misma clave no deben ejecutarse ambas. El bloqueo de fila solo ordena el relevo de una fila existente; el primer reclamo lo arbitra la restricción única, que es la razón por la que el insert perdedor se captura y se responde con 409.

El lease importa para el caso de caída, y su token importa para el caso lento. Un titular pausado más allá de su lease puede despertar e intentar terminar, pero `requireLeaseToken` ve el token más nuevo y toda su transacción, pedido y filas del outbox incluidos, hace rollback. Aquí el token funciona como fence porque el recurso protegido, la base de datos local, lo comprueba dentro de la transacción, la misma idea que el lease del relay del Capítulo 5. La duración del lease solo acorta el solapamiento; el token es lo que hace fallar el commit del titular obsoleto. Quien llama como titular obsoleto recibe un error, y su retry recibe la respuesta almacenada. Ambas comprobaciones usan el reloj de la base de datos, `dbNow()`, y no el del servidor de aplicación, por las razones que da el Capítulo 3 sobre los relojes. Acotar la clave a quien llama evita que un cliente colisione con otro.

El pseudocódigo muestra la ruta de éxito, y la ruta de fallo también necesita una regla. Si la solicitud falla por una razón determinista, como un error de validación que devuelve un 4xx, el servidor almacena esa respuesta como la completada. Un retry con la misma clave recibe entonces el mismo 4xx y no vuelve a ejecutar la validación contra datos que pueden haber cambiado. Si falla por una razón transitoria, como un timeout o una dependencia no disponible, el reclamo se libera o se deja expirar, para que el retry del cliente pueda ejecutar el trabajo. Almacenar un fallo transitorio como la respuesta final convertiría un parpadeo en un error permanente para esa clave.

La restricción única sobre `(caller, key)` cumple el mismo papel de arbitraje que la clave primaria del inbox. Es el mismo patrón: un reclamo atómico, un resultado almacenado y una respuesta determinista ante una repetición. Cuando la API produce eventos, la fila del pedido, las filas del outbox y la finalización del registro de idempotencia pueden escribirse en una sola transacción local, de modo que una solicitud reintentada ni crea un segundo pedido ni emite un segundo `OrderPlaced`. Conserva la clave almacenada al menos tanto como los clientes puedan reintentar, lo cual es una decisión sobre el comportamiento de los clientes y no sobre los brokers.

Una conexión que vale la pena mencionar: la clave que suministró el cliente puede fluir hacia el evento. `Orders` puede ponerla en los headers del evento o derivar de ella el `event_id` de `OrderPlaced`, para que los consu-

midores aguas abajo vean una sola identidad desde el toque de la persona compradora hasta la entrada del libro mayor. Si el `event_id` se deriva, debe provenir de forma determinista de quien llama y de la clave juntos, por ejemplo como un UUID basado en nombre sobre ambos, para que dos clientes que casualmente elijan la misma clave nunca compartan identidad. El Capítulo 11 trata qué pertenece a un evento; que la clave pertenezca ahí depende de hasta dónde quieras que la idempotencia siga a la solicitud.

Trade-offs y costos

El inbox añade una escritura a cada mensaje. Cada mensaje consumido cuesta un insert extra, dentro de la misma transacción, en la ruta crítica. Para un consumidor que hace un puñado de escrituras, el sobrecosto es pequeño. Para un consumidor de muy alto volumen y muy barato, como uno que actualiza el cache de un modelo de lectura, puede ser el costo dominante, y la idempotencia natural suele ser la mejor opción ahí.

El almacenamiento y la limpieza del inbox son un trabajo continuo. La tabla crece con el tráfico, y la política de retención es una decisión que te corresponde y que revisas. Una ventana demasiado corta readmite duplicados antiguos; una demasiado larga cuesta almacenamiento y tamaño de índice. Alguien debe vigilar el job de limpieza, porque cuando se detiene, la tabla crece en silencio hasta convertirse en un problema de rendimiento.

Una restricción única en la ruta crítica puede convertirse en un punto de contención. Con una clave bien distribuida es barata. Con una clave que se agrupa, o con muchos consumidores golpeando un agregado caliente, puede formar cola. Indexar la tabla por consumidor e id de evento, y mantener corta la transacción del handler, lo mantiene manejable.

Las claves de negocio son una decisión de modelado, y una equivocada es difícil de ver. Demasiado finas, y los duplicados lógicos pasan; demasiado gruesas, y se rechaza trabajo legítimo. El fallo es silencioso en ambas direcciones. Probaría la clave con casos deliberadamente incómodos, como dos operaciones legítimas sobre un agregado, antes de confiar en ella.

La idempotencia no lo cubre todo. Hace inofensiva una repetición; no arregla un handler que está mal la primera vez, no ordena los mensajes y no deshace un efecto para el que el proveedor no tiene clave. Un inbox tampoco protege contra un duplicado que ha *cambiado*: si un productor republica “el mismo” hecho con un `event_id` distinto y no existe ninguna clave de negocio, nada lo reconoce.

Cuándo no usar el inbox. Si todos los efectos del consumidor son naturalmente idempotentes y no pueden quedar obsoletos, como sobrescribir un campo desnormalizado con el último estado completo, el inbox es costo sin beneficio, y yo lo omitiría y dejaría el razonamiento anotado en el código. Tampoco es adecuado un consumidor que no puede escribir en una base de datos transaccional propia, como un forwarder sin estado. Ese consumidor necesita que su sistema aguas abajo deduplique, con la clave transmitida, o necesita rediseñarse para que posea un almacén.

Errores comunes

- **Comprobar y actuar en pasos separados.** Es tentador porque el código se lee con naturalidad: “¿lo he visto? no, entonces hazlo”. Falla ante una caída entre ambos pasos y ante la concurrencia. Pon la escritura del inbox en la misma transacción que el efecto y deja que la restricción única arbitre.
- **Escribir la fila del inbox después de confirmar, o antes de que el efecto se confirme.** Confirmar primero pierde el mensaje si el trabajo falla después. Registrar antes de que el efecto se confirme marca como hecho un trabajo que nunca ocurrió. El efecto, la fila del inbox y el commit son una unidad; el ack sigue al commit.
- **Mantener el inbox en un almacén distinto del efecto.** Un cache o una base de datos aparte parece más barato, y recrea el dual write. Manténlo en la misma base de datos que el estado que protege.
- **Generar una clave nueva por intento.** Un valor aleatorio creado dentro del handler o del bucle de retries del cliente hace que cada retry parezca nuevo. Deriva la clave del comando, del evento o de la operación de negocio, para que sea idéntica en cada intento.
- **Usar solo `event_id` y suponer que cubre todos los duplicados.** No atrapa dos mensajes distintos con la misma intención. Añade una res-

tricción de clave de negocio donde el dominio tenga ese riesgo.

- **Elegir una ventana de deduplicación a ojo.** Una ventana más corta que la retención del broker, o que tu práctica de redrive, readmite en silencio duplicados antiguos. Deriva la ventana de esos números y deja escrita la derivación.
- **Tratar una violación única como un error genérico.** Confirmar todas las violaciones oculta bugs reales; reintentar todas las violaciones entra en bucle infinito con un mensaje ya manejado. Captura la violación del inbox por separado.
- **Olvidar el efecto externo.** El inbox no puede deshacer un correo ni un cargo a una tarjeta. Pasa una clave determinista al proveedor, o decide deliberadamente entre un posible duplicado y una posible pérdida.
- **Publicar eventos de seguimiento directamente desde el handler.** Una caída entre el commit y la publicación pierde el evento, y la reentrega toma el camino rápido del inbox y nunca lo repite. Escribe el seguimiento en el outbox en la misma transacción que la fila del inbox y el cambio de estado.
- **Confirmar un mensaje cuyo lease pertenece a otro.** Un duplicado que ve un lease vigente y confirma ha eliminado el único retry si el primer intento falla después. Reencólalo con una demora en su lugar.

Ideas clave

- Los duplicados vienen del broker, del relay, de los retries, del redrive y del replay, y de los clientes, así que todo consumidor debe escribirse como si cualquier mensaje pudiera llegar dos veces.
- Check-then-act es inseguro porque la comprobación y el efecto son pasos separados, y una caída o un intento concurrente pueden colarse entre ellos.
- La fila del inbox y el cambio de negocio deben confirmarse en una sola transacción local en la base de datos propia del consumidor, y la restricción única sobre la clave es lo que realmente arbitra los duplicados concurrentes. Un consumidor que anuncia lo que hizo añade una fila de outbox a esa misma transacción.

- Usa el `event_id` como clave de idempotencia base para el consumo de eventos, y añade una clave de negocio dondequiera que dos mensajes distintos puedan expresar la misma intención.
- La ventana de deduplicación debe superar la retención del broker, tu práctica de redrive y cualquier replay que planees, y una restricción de clave de negocio en la tabla del dominio te protege después de que se borren las filas del inbox.
- Para los efectos secundarios externos, envía una clave de idempotencia determinista derivada del trabajo y no del intento, conserva la fila local del inbox como guardia a largo plazo porque el proveedor olvida las claves y, donde el sistema aguas abajo no tiene clave, elige conscientemente entre un posible duplicado y una posible pérdida.
- En la frontera de la API, quien llama suministra la clave de idempotencia, y el servidor almacena la clave, un fingerprint de la solicitud y la respuesta, para que una repetición devuelva el primer resultado.

La idempotencia hace inofensivo un mensaje repetido, pero supone que el procesamiento acaba teniendo éxito. Algunos mensajes nunca lo tendrán: fallan en cada intento, y los retries y las reentregas se acumulan detrás de ellos. El Capítulo 8 trata esos poison messages y los dead letters que dejan.

Capítulo 8 — Poison messages y dead letters

Imagina Parcelly la mañana siguiente a que un campo nuevo, `gift_message`, llegara a producción en `OrderPlaced`. Notifications lo lee para construir el correo de confirmación, y el pedido de prueba de un vendedor trae un mensaje de regalo con un carácter que la plantilla del correo no sabe codificar. El consumidor lanza una excepción con ese único mensaje. El mensaje vuelve. El consumidor lanza la excepción otra vez.

Al mediodía el mensaje se ha entregado varios miles de veces. Cada intento falla en el mismo punto, de la misma manera, en cuestión de milisegundos. El consumidor está ocupado, sano según todas las métricas de infraestructura, y lleva horas sin hacer ack de nada útil. En un broker basado en colas, los demás workers siguen atendiendo el resto de los mensajes, así que el daño parece pequeño. En un topic basado en log, una partición ha dejado de avanzar porque el consumidor vuelve una y otra vez al registro que falla en lugar de saltarlo, y todos los pedidos de esa partición esperan detrás de él.

Alguien acaba por darse cuenta, y alguien sugiere mover el mensaje a otro sitio para que el trabajo continúe. Esa es la mitad corta de la respuesta. La mitad difícil es lo que ocurre después: quién mira ese mensaje, cómo decide qué significa y cómo vuelve al sistema sin provocar un segundo incidente. Este capítulo trata de las dos mitades. ¿Qué debe hacer un sistema con un mensaje que no puede procesar, y cómo evita un equipo que esa respuesta se convierta en un vertedero?

El problema y por qué es difícil

Un **poison message** es un mensaje que falla cada vez que se procesa. La palabra “cada” lleva la definición. Un mensaje que falla porque una base de datos estuvo inaccesible un momento no es poison; tendrá éxito cuando la base de datos vuelva. Un poison message falla porque algo del men-

saje, o del código que lo lee, hace imposible el éxito hasta que una persona o un deploy cambien algo. En el vocabulario del Capítulo 3, la semántica del fallo dice que volver a intentarlo no puede ayudar, y por eso mismo reintentarlo es esfuerzo desperdiciado.

Las causas habituales se agrupan en unas pocas categorías:

- **Contenido mal formado o inesperado.** Falta un campo obligatorio, un valor está fuera de rango o un payload no se puede parsear. Los cambios de esquema que no eran compatibles con un consumidor antiguo producen esto (el Capítulo 12 explica cómo evitarlo).
- **Un bug en el consumidor.** El mensaje es válido, pero el código maneja mal un caso que nadie probó, como un null donde nunca se esperaba uno.
- **Una suposición de negocio violada.** Un evento `PaymentCaptured` llega a un consumidor que no tiene registro de la autorización correspondiente, quizá porque se borraron datos, o porque los eventos se cruzaron en tránsito y los retries ya han superado cualquier ventana razonable de reordenamiento. Dentro de esa ventana el mismo síntoma es transitorio, como señala el Capítulo 6 y desarrolla a fondo el Capítulo 9, y corresponde a los retries y no a la DLQ. Pasada la ventana, el consumidor no puede continuar y no puede inventar el registro que falta.
- **Un rechazo permanente aguas abajo.** Un transportista externo rechaza una dirección por inválida. Intentarlo de nuevo produce el mismo rechazo.

El Capítulo 6 introdujo la distinción entre fallos transitorios y permanentes, y este capítulo continúa donde aquel termina. Una política de retries que funciona correctamente acabará por rendirse con un mensaje. Los retries están acotados por intentos, por tiempo o por presupuesto. Lo que ocurre en ese punto es una decisión de diseño, y si nadie la toma, la toma el broker por ti.

El bucle infinito de reentrega

Si el consumidor no hace nada especial, entra en juego el comportamiento por defecto del broker. Un broker basado en colas reentrega un mensaje sin ack, ya sea inmediatamente después de un nack o cuando vence el visi-

bility timeout, y por lo general lleva un contador de reentregas que crece con cada intento.

Un broker basado en log funciona de otra manera, y la diferencia importa. No hay reentrega por mensaje: el consumidor mantiene un offset y lee hacia adelante. No hacer commit de un offset no consigue que el broker devuelva el registro; la posición en memoria del consumidor sigue avanzando, y el registro fallido se omite en silencio hasta que el siguiente reinicio o rebalanceo devuelve al consumidor al último offset confirmado. Un bucle en un log existe solo porque el consumidor lo construye: vuelve al offset fallido y lo lee otra vez, o deja de avanzar la partición mientras reintentando en el sitio. Un consumidor que no hace ninguna de las dos cosas y hace commit más allá del fallo pierde el registro sin ninguna señal.

En ambas familias el resultado puede ser el mismo: el mensaje vuelve, falla y vuelve otra vez, o, en un log con un manejo descuidado de los offsets, desaparece. Las herramientas correctas de la familia de logs son un seek o una pausa explícitos, un retry topic que reciba el registro fallido para que la partición principal pueda avanzar, y un dead-letter topic para los registros de los que se ha desistido.

El bucle tiene tres costos. El primero es la capacidad desperdiciada: los workers dedican su tiempo a un trabajo que no puede tener éxito. El segundo es el ruido: los logs se llenan de stack traces idénticos y los problemas reales se esconden entre ellos. El tercero es el que más importa, el **head-of-line blocking**. En un log, un consumidor procesa una partición en orden, así que un registro que no se salta bloquea a todos los que vienen detrás. En una cola simple, un poison message no bloquea a los demás por sí solo, pero un puñado de ellos puede ocupar a todos los workers. Si la cola agrupa los mensajes para conservar el orden (colas FIFO, message groups, sesiones), un poison message bloquea a su grupo, que es el mismo radio de impacto que una partición de un log, solo que más estrecho.

Las dos malas salidas

Los equipos bajo presión suelen recurrir a una de dos salidas, y ambas son incorrectas por la misma razón: esconden el fallo en lugar de manejarlo.

La primera es **descartar el mensaje**: capturar la excepción, registrarla, hacer ack y seguir. El pipeline vuelve a fluir, pero el evento de negocio se ha perdido. Para `OrderPlaced` eso significa un pedido que existe en `Orders` y en ningún otro servicio. Para `PaymentCaptured` significa un libro mayor que no coincide con la realidad. Descartar es la decisión correcta solo cuando el mensaje no tiene valor, y un consumidor no puede saberlo en el momento del fallo.

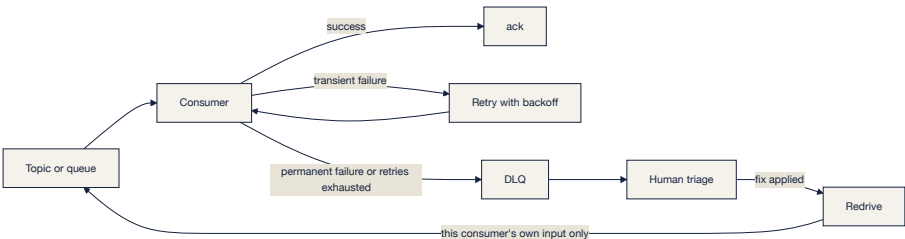
La segunda es **reintentar para siempre** y confiar en que alguien arreglará la causa. Es el bucle infinito de arriba, y convierte un mensaje malo en una caída para todo lo que hay detrás.

La tercera opción está entre las dos. Apartar el mensaje, con el contexto suficiente para que una persona pueda entenderlo más tarde, y dejar que el resto del stream continúe. Esa opción es la dead-letter queue.

El patrón de la dead-letter queue

Una **dead-letter queue (DLQ)** es un destino aparte donde un consumidor o un broker colocan los mensajes de los que ha desistido. En un broker basado en log, la misma idea suele ser un **dead-letter topic**: un topic ordinario que recibe los registros fallidos. Cuando una afirmación vale para las dos familias, este capítulo dice DLQ.

El patrón tiene un núcleo simple. Cuando un mensaje agota sus retries, o falla de una manera que el consumidor clasifica como permanente, el consumidor lo mueve a la DLQ y hace ack del original. El stream continúa. El mensaje fallido no se pierde; queda aparcado donde se puede examinar.



Fíjate en las dos últimas flechas del diagrama. La DLQ no es una salida. Es una sala de espera con una puerta de vuelta a la propia entrada del consumidor que falló, y toda la dificultad de este capítulo está en la distancia

entre esas dos puertas. La puerta no lleva a un topic compartido: los demás consumidores ya manejaron el evento y no deberían volver a verlo.

Dónde se hace el movimiento

Hay dos maneras de implementar el movimiento, y la elección afecta a lo que puedes almacenar.

Dead-lettering gestionado por el broker. Muchos brokers basados en colas se pueden configurar para que un mensaje entregado más de cierto número de veces se mueva automáticamente a una cola designada. Cuesta poco configurarlo y funciona incluso cuando el consumidor se cae antes de poder hacer nada. Te da un contador de entregas, pero normalmente no el motivo por el que falló el consumidor, porque el broker no lo conoce.

Dead-lettering gestionado por el consumidor. El consumidor captura el fallo, construye un registro que incluye el motivo, lo publica en la DLQ y solo entonces hace ack del original. Esto permite clasificar los fallos y enviar un error permanente a dead letter en el primer intento en lugar de en el quinto. Cuesta código en cada consumidor, y hay que escribirlo con cuidado: publicar primero en la DLQ y hacer ack después, de modo que una caída entre ambos pasos produzca un duplicado en la DLQ y nunca una pérdida.

Usaría los dos donde el broker ofrezca el primero. La regla gestionada por el broker existe solo en algunos brokers basados en colas, y no todos los tipos de cola limitan las entregas. Donde existe, es la red de seguridad para los consumidores que se caen o se cuelgan de maneras que el código no puede capturar, y junto con la vía gestionada por el consumidor significa que un poison message no puede girar para siempre, ni siquiera si tu propio manejo tiene un bug.

Los brokers basados en log, en general, no tienen ni un contador de entregas por mensaje ni una DLQ del lado del broker, y eso deja un hueco. Un mensaje que tumba el proceso, como un fallo del deserializador antes de que corra el handler, un error de memoria agotada o una caída nativa, nunca llega a tu bloque `try`. El consumidor se reinicia en el offset confirmado, lee el mismo registro y se cae de nuevo. El Capítulo 6 señaló que el contador de intentos vuelve a empezar tras una caída. Para un registro poison, eso es todo el problema.

El remedio es contar antes de procesar, de modo que la cuenta sobreviva a la caída:

```
function pollLoopStep(record):
    key = record.partition + ":" + record.offset
    // the counter is written before any work, so a crash still counts
    attempts = incrementDurableCounter(key)
    if attempts > MAX_PROCESS_STARTS:
        // already tried too many times: do not run the code that
        crashes
        deadLetterRaw(record, "exceeded process starts")
        clearDurableCounter(key)
        return
    // deserialization is inside the guarded path, not before it
    handleMessage(deserializeGuarded(record))
    clearDurableCounter(key)
```

El contador se indexa por partición y offset (o por `event_id` si puedes leerlo sin deserializar). El precio es una escritura durable por registro en el camino crítico, y un límite que tiene que ser lo bastante alto para que el reinicio de un deploy ordinario no empuje a los registros sanos por encima de él. La alternativa es aceptar que un registro poison que provoca un bucle de caídas detiene su partición hasta que alguien lo note, y por eso importa la alarma de retraso que se describe más abajo.

```
function handleMessage(message):
    try:
        processBusinessLogic(message)
        ack(message)
    on PermanentFailure as failure:
        // classified permanent: retrying cannot help, so skip the
        retry budget
        deadLetter(message, failure)
    on error as failure:
        // Chapter 6's classifier: anything unclassified is "unknown"
        // and gets the stricter attempt limit, never an escaped
        exception
        limit = attemptLimitFor(classifyFailure(failure))
        if message.attempt_count >= limit:
            deadLetter(message, failure)
        else:
            // nack with backoff on a queue; retry topic on a log
            (Chapter 6)
            retryLater(message)
```

El paso `deadLetter` es donde se captura el contexto necesario para el triaje, y el orden de sus dos últimas líneas es lo esencial.

```
function deadLetter(message, failure):
    record = {
        original_message: message.body,
        original_headers: message.headers,
        original_key:      message.key,
        original_time:     message.timestamp,
        source:             message.source_name,
        source_position:    message.position,
        consumer_input:     CONSUMER_INPUT,
        redrive_count:      message.redrive_count,
        error_class:        failure.class,
        error_message:      truncate(failure.message, 1000),
        attempt_count:      message.attempt_count,
        consumer:           CONSUMER_NAME,
        consumer_version:   BUILD_VERSION,
        first_failed_at:    message.first_failed_at,
        failed_at:          now(),
        correlation_id:     message.correlation_id,
        stack_trace_ref:    failure.trace_pointer
    }
    // publish first, ack second: a crash between them yields a
duplicate
    // dead letter, which is recoverable; the reverse order loses the
message
    publish(DLQ_NAME, record)
    ack(message)
```

Si el propio `publish` falla, `deadLetter` lanza una excepción y el mensaje no recibe ack. En un broker de la familia de colas eso basta: el mensaje vuelve mediante la reentrega normal. En un broker de la familia de logs no basta, porque un offset sin commit no hace volver el registro; el consumidor debe volver con un seek a la posición del registro fallido, o pausar la partición, y reintentar la publicación en la dead letter antes de seguir leyendo. En cualquier caso el objetivo es el mismo modo de fallo: el mensaje permanece en el sistema hasta que tiene un nuevo hogar seguro.

Qué almacenar con una dead letter

Una dead letter sin contexto es un acertijo. Un equipo que encuentra un mensaje en una DLQ seis semanas después necesita responder cuatro preguntas: qué era, de dónde venía, por qué falló y si sigue siendo seguro procesarlo. El registro debería responderlas sin abrir un repositorio de código fuente.

Campo	Por qué está ahí
Cuerpo y headers originales, byte por byte	El redrive debe reproducir el mensaje exactamente, incluido <code>event_id</code>
Nombre y posición de origen (topic, partición y offset, o nombre de la cola)	Ubica el mensaje en el historial y respalda las decisiones de orden
Clave original (clave del registro o routing key) y timestamp original	El redrive debe publicar con la misma clave; si no, el mensaje cae en otra partición y pierde su orden por clave
Entrada del consumidor (la cola propia del consumidor que falla o su topic de redrive)	El único lugar donde un redrive puede publicar
<code>redrive_count</code>	Separado de <code>attempt_count</code> : cuántas veces ha vuelto ya este mensaje
<code>error_class</code> y <code>error_message</code>	Agrupar los fallos por causa; trunca los mensajes largos
Stack trace o un puntero a la línea del log	Permite a un ingeniero ver dónde falló sin reproducirlo
<code>attempt_count</code>	Distingue “falló una vez, permanente” de “falló tras muchos retries”
<code>first_failed_at</code> y <code>failed_at</code>	Muestra cuánto tiempo lleva esperando el mensaje y cuándo empezó el problema
Nombre y versión del consumidor	Vincula el fallo con un deploy; suele ser el camino más rápido hacia la causa
<code>correlation_id</code>	Conecta el mensaje con el resto de su flujo de negocio

Hay que tener en cuenta dos precauciones. La primera, conserva el payload original sin modificar. Si lo envuelves en un envelope propio, la herramienta de redrive debe desenvolverlo exactamente, o `event_id` y `causation_id` se desviarán y se perderán las garantías de idempotencia del Capítulo 7. La segunda, recuerda que una DLQ es una copia de datos de producción, a menudo conservada durante mucho tiempo. Hereda los controles de acceso y las obligaciones de retención del original, y los payloads con datos personales necesitan en la DLQ el mismo cuidado que en cualquier otro sitio.

Parking lot frente a DLQ

La palabra DLQ se usa para dos cosas distintas, y separarlas facilita razonar sobre la operación.

Un **parking lot** es un lugar de retención para las dead letters que necesitan una decisión humana. La DLQ es donde el sistema pone un mensaje automáticamente; el parking lot es donde una persona pone un mensaje de manera deliberada después de mirarlo y decidir que un redrive no lo arreglará. Es un lugar distinto con un propósito distinto.

Piensa en qué contiene cada uno. La DLQ contiene mensajes que fallaron y no se han examinado. Debería vaciarse: su estado esperado es vacía o casi vacía, y cualquier cosa antigua en ella es un síntoma. El parking lot contiene mensajes que alguien examinó y apartó: un pedido que se canceló antes de que llegara su evento, un payload que necesita corrección manual, un registro que debe conciliarse a mano con otro equipo. Esos mensajes no esperan a que se despliegue un arreglo. Esperan una decisión.

Mezclarlos produce un fallo previsible. Si los mensajes ya decididos y aparcados se quedan en la DLQ, la profundidad de la DLQ nunca vuelve a cero, se sube el umbral de la alerta para acallar el ruido, y el siguiente poison message real llega sin que nadie lo note. Mantenerlos separados deja afilada la alarma de la DLQ.

Un parking lot puede ser tan modesto como una tabla de base de datos con un estado, un responsable, un motivo y una fecha de revisión. No necesita ser un destino del broker en absoluto, porque nada lo va a consumir automáticamente. La fecha de revisión es lo que más importa: un mensaje aparcado sin fecha para su próxima revisión ya está olvidado.

Responsabilidad y alarmas

Una DLQ sin responsable es un sumidero que se llena en silencio. La primera decisión, antes de cualquier herramienta, es quién responde por cada una.

Mi criterio por defecto es que el equipo que es responsable del consumidor es responsable de la DLQ. Escribieron el código que falló, saben qué debía significar el mensaje y son quienes pueden arreglarlo y desplegar. Una DLQ por consumidor, o por grupo de consumidores y origen, hace eviden-

te esa responsabilidad. Una sola DLQ compartida para toda la empresa la vuelve ambigua, y la ambigüedad significa que nadie mira. Cuando un equipo de plataforma proporciona las herramientas, el equipo de plataforma es responsable del mecanismo y el equipo consumidor es responsable del contenido.

Sobre qué alarmar

Alarma sobre la **llegada**, no solo sobre la profundidad. Un solo mensaje que aparece en una DLQ normalmente vacía merece una notificación, porque la primera dead letter es el momento más barato para encontrar un bug. Un umbral de profundidad por sí solo no te dice nada hasta que la cola ya es profunda.

Tres señales juntas cubren la mayoría de los casos:

- **Cualquier llegada nueva** a una DLQ que debería estar vacía. Dirígela a un canal que una persona lea durante el horario laboral.
- **Tasa de llegada.** Una ráfaga repentina suele significar un mal deploy o un cambio de esquema incompatible, y justifica enviar un page a alguien.
- **Antigüedad del mensaje más viejo.** Esto detecta el fallo lento: la DLQ que recibe un mensaje al día, sobre el que nadie actúa, hasta que el más antiguo tiene un mes y el contexto de negocio ha desaparecido.

No sé cuál es el umbral correcto de paging para tu sistema, y nadie puede decírtelo sin conocer el valor de negocio de los mensajes en cuestión. Una DLQ asociada a los consumidores de `PaymentCaptured` merece un umbral más bajo que una asociada a `Notifications`, porque un correo perdido es una molestia y una captura perdida es una discrepancia en el libro mayor. Los umbrales deberían reflejar esa diferencia, y la diferencia debería quedar escrita junto a la alarma.

Nota: Alarma también sobre el retraso del consumidor. Un consumidor atascado en un registro poison de una partición de log no produce necesariamente dead letters, porque nunca se rinde. Un retraso creciente sin throughput es la señal de que está corriendo el bucle descrito al comienzo de este capítulo.

Redrive: devolver los mensajes

Redrive significa devolver las dead letters al consumidor que falló con ellas, para que las reprocese. El destino es la propia entrada de ese consumidor: su retry topic o de redrive, o su propia cola. Nunca es el topic compartido del que vino originalmente el mensaje, porque todos los demás consumidores de ese topic ya procesaron el evento, y republicarlo ahí se lo entregaría a todos por segunda vez. La clave original viaja con el mensaje, de modo que se queda en la misma partición y conserva su posición por clave. El contador de intentos vuelve a empezar en cero, y el `redrive_count`, que es aparte, sube en uno, así que un mensaje que no deja de volver se hace visible en lugar de parecer nuevo cada vez. El redrive es la operación que hace valiosa a una DLQ y también la que con más probabilidad causa un segundo incidente. El Capítulo 3 describió al ingeniero que pulsa un botón y hace un redrive de 40,000 mensajes de golpe. El remedio es tratar el redrive como un procedimiento diseñado, no como un botón de comodidad.

Arregla la causa antes del redrive

Un mensaje que falló por un bug del consumidor volverá a fallar hasta que el bug se arregle y se despliegue. Hacer el redrive antes del arreglo mueve los mensajes en círculo y vuelve a llenar la DLQ con las mismas entradas, ahora con contadores de intentos más altos y un historial confuso. El orden de trabajo es: entender la causa, desplegar el arreglo, hacer el redrive de una muestra pequeña, ver que tiene éxito y luego hacer el redrive del resto.

El redrive debe ser seguro de repetir

Un mensaje con redrive es un duplicado por definición: el sistema ya ha visto este `event_id` y es posible que el consumidor también. Si el procesamiento original tuvo éxito en parte antes de fallar, puede que algunos efectos ya existan. El Capítulo 7 es la razón por la que esto es sobrevivible. Un consumidor idempotente con un inbox trata el mensaje del redrive como cualquier otra reentrega y aplica sus efectos una sola vez.

Vale la pena enunciar dos puntos. El redrive debe conservar el `event_id` original, ya que esa es la clave de idempotencia. Una herramienta de redrive que genera un identificador nuevo para cada mensaje anula el inbox,

y el efecto puede aplicarse dos veces. Y un consumidor que no es idempotente no debería ser el destino de un redrive masivo en absoluto; arregla eso primero, o acepta que cada mensaje con redrive es una evaluación de riesgo manual.

Redrive en lotes controlados

Una herramienta de redrive segura tiene unas pocas propiedades, ninguna de ellas exótica:

1. **Un filtro de selección.** Haz el redrive por clase de error, ventana de tiempo o `correlation_id`, y no solo de “todo”.
2. **Un rate limit.** Los mensajes vuelven a la entrada del consumidor a un ritmo que él y sus dependencias puedan absorber. Una avalancha de mensajes con redrive es un retry storm autoinfligido, y los límites de trabajo en curso y el load shedding de “Retries bajo sobrecarga: back-pressure” (Capítulo 6) se aplican al tráfico del redrive exactamente igual que al tráfico en vivo.
3. **Un interruptor de pausa.** Un operador puede detener el redrive a mitad si reaparecen los fallos.
4. **Un dry run.** La herramienta informa de lo que movería antes de mover nada.
5. **Un registro de auditoría.** Quién hizo el redrive de qué, cuándo y por qué, guardado en un lugar durable.

```

function redriveBatch(dlqName, filter, batchSize, maxPerSecond):
    // an operator procedure, not a consumer: run it with a human
    watching
    batch = readFromDlq(dlqName, filter, batchSize)
    for each record in batch:
        if redriveIsPaused():
            return "paused"
        // the failing consumer's own input, never the shared source
        topic;
        // the original key keeps the message on its partition, and
        the
        // original headers and event_id keep the consumer's inbox
        effective
        publish(record.consumer_input, record.original_message,
            key: record.original_key,
            headers: record.original_headers
                + { attempt: 0, redrive_count:
record.redrive_count + 1 })
        // mark this record as redriven, one by one; a crash leaves a
        few
        // already-published records unmarked, which is a duplicate,
        not a loss
        markRedriven(record, operator(), now())
        sleep(1 second / maxPerSecond)
    return "batch complete"

```

Fíjate en el orden dentro del bucle: un registro se marca como con `redrive` después de que la publicación tenga éxito, no antes. Si la herramienta se cae a mitad, algunos mensajes han recibido el `redrive` y siguen sin marcar en la DLQ, lo que lleva a duplicados, no a pérdidas. Un consumidor con inbox absorbe esos duplicados; uno sin él convierte la caída de una herramienta en un problema de datos. El registro se marca, no se borra, de modo que la pista de auditoría y el `redrive_count` sobreviven; la limpieza sigue la regla de retención de “Cómo evitar el vertedero” más abajo.

Redrive y orden

El `redrive` rompe el orden a menos que lo diseñes con ese fin. Un mensaje enviado a dead letter fue saltado; los mensajes posteriores del mismo agregado se procesaron mientras tanto. Cuando llega el mensaje con `redrive`, llega después de ellos. Es el ejemplo de Inventory del Capítulo 6 con un intervalo más largo: si `OrderPlaced` se envía a dead letter en lugar de reintentarse, `OrderCancelled` sigue sin encontrar una reserva que liberar,

y la colocación con redrive, horas después, sigue creando una que nada liberará. Conservar la clave no lo repara: la clave pone el mensaje en la partición correcta, pero la partición ya ha avanzado.

Que esto importe depende del consumidor. Un consumidor **conmutativo**, es decir, uno cuyo estado final es el mismo sea cual sea el orden en que lleguen los eventos, no se ve afectado. Uno que depende de transiciones de estado sí. El Capítulo 9 explica cómo detectar y manejar la llegada desordenada con números de versión o comprobaciones de secuencia, y un consumidor basado en estado debería usarlas antes de que nadie haga un redrive hacia él. Si no lo hace, el procedimiento de redrive debe incluir una comprobación de que el mundo no ha seguido su curso desde que el mensaje se envió a dead letter.

Qué le hace una dead letter a un stream ordenado

Los streams ordenados fuerzan la decisión más difícil de este capítulo, porque la opción que protege el orden es la que detiene el negocio.

Considera un topic basado en log con clave `order_id`, donde Ledger registra asientos. Los consumidores procesan cada partición en secuencia. El mensaje 41 de `order-81724` falla de forma permanente. El consumidor tiene dos opciones:

- **Detenerse y esperar.** Mantener la partición en el mensaje 41 hasta que alguien arregle la causa. El orden se conserva exactamente. Todos los demás pedidos que caen en esa partición por el hash también se detienen, aunque sus mensajes estén bien. El radio de impacto es la partición entera.
- **Enviar a dead letter y continuar.** Aparcar el mensaje 41, procesar el 42 y los siguientes. La partición fluye. Pero el 42 puede haber dependido del 41, y entonces el consumidor lo ha procesado contra un estado que no incluye al 41.

Ninguna de las dos opciones es gratis. Detenerse cambia disponibilidad por corrección; continuar cambia corrección por disponibilidad. No hay ningún ajuste que dé ambas cosas, y es mejor elegir a propósito para cada consumidor que heredar lo que resulte ser el valor por defecto del framework.

Mi regla práctica tiene tres partes:

1. **Si el consumidor es insensible al orden o conmutativo**, envía a dead letter y continúa. Notifications es el caso típico: un correo fallido en un pedido no influye en el siguiente.
2. **Si los mensajes posteriores de la misma clave dependen del que falló**, no continúes para esa clave. Envía a dead letter el mensaje fallido y desvía además los siguientes mensajes del mismo `partition_key` a una zona de retención hasta que el primero se resuelva. Las demás claves continúan. Así el radio de impacto se queda en un agregado en lugar de en una partición, al precio de un mecanismo adicional: el seguimiento por clave de las claves “bloqueadas”.
3. **Si el consumidor es un libro mayor financiero, detente**. El costo de procesar fuera de orden es un registro que está mal de una manera difícil de detectar y cara de reparar, mientras que una partición en pausa es ruidosa y obvia. Corrección por encima de velocidad fue la regla declarada para Ledger, y aquí se aplica en su forma más simple.

La variante por clave de la segunda regla merece un esbozo, porque es la más útil y la menos común. Supone una pequeña tabla `key_block` con una fila por clave bloqueada (`partition_key`, `blocked_event_id`, `state`), y que un solo consumidor a la vez maneja el tráfico en vivo de una clave dada, que es lo que te da una partición o un message group.

```

function handleOrderedMessage(message):
    key = message.partition_key
    parked = false
    transaction:
        // select ... for update; none if the key is open
        block = lockBlockRow(key)
        if block exists and message.event_id !=
block.blocked_event_id:
            // park under the same lock that resolveBlockedKey takes,
so a
            // message cannot be parked after the block was cleared

            // held messages are unique on event_id; a repeat is
ignored
            appendToHeldMessages(key, message)
            parked = true
    if parked:
        ack(message)
        return
    try:
        // the inbox absorbs a repeat
        processBusinessLogic(message)
        // this was the redriven message
        if block exists:
            // durable: a crash from here on is recoverable
            markResolving(key)
        ack(message)
        if block exists:
            resolveBlockedKey(key)
    on PermanentFailure as failure:
        transaction:
            // state = "blocked"
            markBlocked(key, message.event_id)
            deadLetterRecord(message, failure)
        // blocked key and dead letter commit together
        ack(message)

```

La liberación es un procedimiento aparte, y un barrido periódico la hace reinicialable.


```

function resolveBlockedKey(key):
  loop:
    transaction:
      block = lockBlockRow(key)
      if block is none or block.state != "resolving":
        // another resolver cleared or re-blocked it
        return
      next = oldestHeldMessage(key)
      if next is none:
        // emptiness check and clear under the lock
        clearBlocked(key)
        return
    try:
      // the inbox absorbs a repeat after a crash
      processBusinessLogic(next)
      transaction:
        removeFromHeld(key, next)
    on PermanentFailure as failure:
      transaction:
        block = lockBlockRow(key)
        if block is none or block.state != "resolving":
          // the key moved on while next was in flight
          return
        // state = "blocked", anchored on next
        markBlocked(key, next.event_id)
        deadLetterRecord(next, failure)
        removeFromHeld(key, next)
      return

function sweepResolvingKeys():
  // runs at worker start and on a timer
  for each key in keysInState("resolving"):
    resolveBlockedKey(key)

```

Tres detalles de este esbozo deciden si funciona. El primero es el propio `redrive`. El mensaje con `redrive` es justamente aquel en el que está bloqueada la clave, así que tiene que saltarse el bloqueo: la comprobación compara su `event_id` con el bloqueado. Sin esa excepción, el `redrive` queda aparcado detrás de los mensajes a los que debía preceder, y la clave nunca se desbloquea.

El segundo es la carrera con el tráfico en vivo. Mientras `resolveBlockedKey` vacía los mensajes retenidos, siguen llegando mensajes nuevos de la misma clave. Aun así deben retenerse, no procesarse, hasta que el conjunto retenido esté vacío. Eso solo se sostiene si la decisión de aparcar y la decisión de despejar están serializadas: ambas toman el bloqueo de fila de la clave, y el aparcado se escribe en la misma transacción que la comprobación. Si el aparcado fuera un paso aparte, un mensaje en

vivo podría ver una clave bloqueada, encontrar luego el conjunto retenido ya vaciado y despejado, y quedar aparcado detrás de un bloqueo que ya no existe. Nadie lo vaciaría, y los mensajes posteriores se procesarían antes que él.

El tercero es una caída en mitad de la liberación. Una vez procesado el mensaje con redrive, el estado `resolving` se escribe antes del ack, de modo que el trabajo pendiente queda registrado en la base de datos y no solo en un mensaje que nunca volverá. El barrido recoge cualquier clave que haya quedado en ese estado y continúa; un mensaje procesado poco antes de la caída lo absorbe el inbox. Si un mensaje retenido falla de forma permanente durante la liberación, la clave se bloquea de nuevo, anclada en ese mensaje, con su propia dead letter, y el resto del conjunto retenido espera a su redrive. El bloqueo por clave es un bucle, no un evento único.

El esbozo no está completo. Un barrido puede solaparse con una liberación en vivo, de modo que dos resolvedores trabajen sobre la misma clave. La comprobación de estado bajo el bloqueo impide que cualquiera de ellos adelante a un nuevo bloqueo: cuando un resolvedor ancla el bloqueo en un mensaje fallido, el otro encuentra que la clave ya no está en `resolving` y se detiene. Lo que queda es trabajo duplicado sobre el mismo mensaje retenido, que el inbox absorbe; un lease sobre el estado `resolving` ahorraría ese trabajo, no la corrección. Tampoco pone ningún límite al conjunto retenido, y deja fuera cómo descarta un operador un mensaje retenido que ya no tiene sentido. Tómallo como la forma del mecanismo, no como uno terminado.

Esto es maquinaria real con sus propios modos de fallo, como que el almacén de mensajes retenidos crezca sin límite, así que yo lo construiría solo para streams donde el orden por clave importe y la tasa de dead letters sea lo bastante baja para que el conjunto retenido siga siendo pequeño. Además, es un mecanismo que no sabe si los mensajes retenidos siguen teniendo sentido una vez resuelto el que falló. Cuando hay que validar las propias transiciones de estado, y no solo ponerlas en secuencia, el version buffer del Capítulo 9 es la herramienta más fuerte: retiene un evento hasta que su predecesor se ha aplicado de verdad, por número de versión, y funciona sin un registro de dead letter que le sirva de ancla.

Cómo evitar el vertedero

Una DLQ se convierte en un vertedero cuando los mensajes entran más rápido de lo que nadie decide qué significan. El síntoma es conocido: miles de entradas, muchos errores distintos, nadie seguro de cuáles importan y una regla no escrita de que no se toca. En ese punto la DLQ ya no es un mecanismo de seguridad; es un lugar donde los datos perdidos se almacenan con un nombre educado.

Los vertederos tienen causas, y cada una tiene su contramedida.

- **Sin responsable.** La contramedida es la regla de responsabilidad de arriba, puesta por escrito.
- **Sin cadencia de triaje.** Las dead letters deberían revisarse según un calendario, y no solo cuando salta una alarma. Para una DLQ de poco volumen, una revisión semanal es un punto de partida razonable.
- **Una retención que sobrevive a la utilidad, o que vence antes del triaje.** Una dead letter de una reserva de stock que venció el mes pasado no puede someterse a un redrive con sentido. Fija un período de retención basado en cuánto tiempo sigue siendo válido el mensaje para el negocio, y decide de antemano qué ocurre al vencimiento: archivar, descartar con una entrada de auditoría o escalar. El error opuesto es igual de común: una retención más corta que la cadencia de triaje borra los mensajes antes de que nadie los haya mirado. Si la DLQ es una cola del broker cuya retención es más corta que tu intervalo de revisión, el broker elimina la evidencia primero. Fija la retención de la DLQ por encima del intervalo más largo entre revisiones, y lee la documentación de tu plataforma para saber desde qué momento empieza a correr el reloj de la retención. Si empieza en el momento original de encolado y no en el de la dead letter, un mensaje que esperó mucho antes de fallar llega ya cerca de su vencimiento. Es una razón más para que el registro guarde el timestamp original.
- **Todo va a dead letter.** Si los resultados de negocio rutinarios, como un rechazo de validación que es parte normal del flujo, se envían a la DLQ, los fallos reales se ahogan. Un pedido rechazado es un evento de negocio con su propio manejo, no una dead letter.
- **Fallos transitorios enviados a dead letter demasiado pronto.** Si la política de retries se rinde en segundos durante una caída breve, todos

los mensajes en vuelo durante esa caída terminan en la DLQ. El remedio está en el Capítulo 6: una política de retries cuyos intentos y deadline estén dimensionados para la caída que esperas sobrevivir.

También daría a cada dead letter un destino final. El triaje termina con uno de una lista corta de resultados: redrive tras un arreglo, corregida y con redrive, aparcada para una decisión manual o descartada con un motivo registrado. “Sigue ahí” no es un destino final.

Trade-offs y costos

Una DLQ mueve un fallo de un lugar donde es ruidoso, el consumidor bloqueado, a un lugar donde es silencioso. Esa es a la vez la ventaja y el peligro. El stream fluye, y el mensaje fallido puede olvidarse. Todo en este capítulo que no es la DLQ en sí, las alarmas, el responsable, la cadencia de revisión y la retención, existe para contrarrestar ese silencio.

Costos concretos que hay que sopesar:

- **Carga operativa.** Alguien tiene que hacer el triaje, decidir y hacer el redrive. Una DLQ para la que nadie tiene tiempo es peor que ninguna, porque esconde el problema detrás de la apariencia de un mecanismo.
- **Gobierno de datos.** Las dead letters contienen payloads de producción, a veces con datos personales, fuera del ciclo de vida habitual. Necesitan controles de acceso, reglas de retención y vías de borrado.
- **Riesgo de orden.** Enviar a dead letter un mensaje de un stream ordenado lo salta. A menos que construyas un bloqueo por clave o detengas la partición, los eventos posteriores se aplican sin él. El propio bloqueo por clave añade un almacén de mensajes retenidos, un bypass del redrive, un bucle de liberación que debe competir de forma segura con el tráfico en vivo y un barrido que reinicia una liberación interrumpida.
- **Identidad almacenada.** Un redrive seguro necesita la clave original, la partición y la entrada del consumidor en cada registro, y un `redrive_count` junto al contador de intentos. Es más para almacenar y más para mantener consistente, y sin ello el redrive solo puede ser una conjetura.

- **Inversión en herramientas.** Una herramienta de redrive segura, con filtros, rate limits, destino final por registro y pista de auditoría, es un proyecto pequeño. Comprarla o construirla forma parte del costo de tener una DLQ.

Cuándo no usar una: si un mensaje se puede regenerar o su pérdida es inofensiva, una DLQ añade más proceso que valor. Un evento de calentamiento de cache o una actualización puramente cosmética se pueden descartar con una métrica que cuente los descartes. Y donde la respuesta correcta al fallo es detenerse, como en el ejemplo del libro mayor, una DLQ que deja continuar el stream juega en tu contra. En ese caso, detén el consumidor, alerta con fuerza y arregla la causa.

Hay además un límite a lo que puede hacer el patrón. Una DLQ contiene los mensajes que fallaron de forma visible. No hace nada por los mensajes que se procesaron mal sin lanzar un error, que es el territorio del Capítulo 13 y de su trabajo de conciliación.

Advertencia: Nunca hagas ack del mensaje original antes de que la dead letter esté almacenada de forma durable. Si la publicación en la DLQ falla después del ack, el mensaje no existe en ninguna parte, y ninguna alarma te lo dirá.

Errores comunes

- **Enviar a dead letter sin contexto.** Es tentador porque mover el cuerpo en bruto es el código más corto. La solución es el registro de metadatos descrito arriba, incluidos los headers originales exactos, de modo que sean posibles tanto el redrive como el diagnóstico.
- **Reintentar fallos permanentes.** Un error de parseo no mejora con el tiempo, y aun así una política de retries genérica lo trata como un timeout. Clasifica los fallos y envía los permanentes directamente a la DLQ en lugar de gastar el presupuesto de retries.
- **Hacer el redrive antes de arreglar la causa.** La presión por “poner los mensajes en marcha” es real y el resultado son los mismos mensajes de vuelta en la DLQ. Despliega el arreglo, haz el redrive de una muestra y luego del resto.

- **Hacer el redrive de todo a la vez.** Una avalancha de mensajes viejos compite con el tráfico en vivo y puede iniciar un incidente nuevo. Aplica rate limit, usa lotes y ten un interruptor de pausa.
- **Alarmar solo sobre la profundidad.** Parece la métrica obvia, pero se dispara tarde. Alarma sobre la primera llegada, sobre la tasa y sobre la antigüedad del mensaje más viejo.
- **Una DLQ compartida para todo.** Es barata de crear e imposible de asumir como propia. Da a cada consumidor la suya y asigna al responsable por nombre.
- **Dejar que se mezclen los mensajes aparcados y los no examinados.** La DLQ nunca se vacía, la alerta se afloja y el poison real pasa inadvertido. Mantén un parking lot aparte, con responsables y fechas de revisión.
- **Ignorar el orden en el redrive.** Un mensaje con redrive llega después de eventos posteriores del mismo agregado. Usa las comprobaciones de versión del Capítulo 9, o confirma que el estado no ha seguido su curso.
- **Hacer el redrive al topic compartido.** Parece “devolver el mensaje a donde estaba”, y entrega el evento por segunda vez a todos los demás consumidores de ese topic. Haz el redrive a la propia entrada del consumidor que falla, con la clave original.
- **Esperar para siempre en un registro que provoca un bucle de caídas.** En un broker de log, un consumidor que nunca se rinde no produce dead letters ni más alarma que el retraso. Cuenta los intentos de forma durable por partición y offset, para que un reinicio no reinicie la cuenta.

Ideas clave

- Un poison message falla cada vez que se procesa, así que reintentarlo para siempre convierte un mensaje malo en una caída para todo lo que hay en cola detrás de él.
- Descartar un mensaje fallido y reintentarlo para siempre son dos maneras de esconder un fallo; una dead-letter queue aparta el mensaje con su contexto para que el resto del stream pueda continuar.

- Una dead letter debe llevar el cuerpo y los headers originales sin cambios, su posición de origen, el error, el contador de intentos y la versión del consumidor, porque la persona que la lea semanas después solo tiene ese registro.
- Publica en la DLQ antes de hacer ack del original, para que una caída produzca un duplicado recuperable y nunca un mensaje perdido.
- Toda DLQ necesita un equipo responsable con nombre, una alarma sobre la primera llegada, la tasa y la antigüedad, y una cadencia regular de triaje; si no, se convierte en un vertedero.
- Un redrive solo es seguro cuando la causa está arreglada, se conservan el `event_id` y la clave originales, el mensaje va a la propia entrada del consumidor que falló y nunca a un topic compartido, el consumidor es idempotente y los mensajes vuelven en lotes con rate limit y un interruptor de pausa.
- En un stream ordenado, enviar a dead letter un mensaje obliga a elegir entre detener la partición y continuar pasando por encima de un hueco; elige por consumidor y a propósito, detente en los libros mayores y bloquea por clave solo donde la maquinaria compense su costo.

El dead-lettering y el redrive se apoyan los dos en una pregunta que este capítulo ha ido aplazando: qué significa que los mensajes lleguen en el orden correcto, y qué se puede construir cuando no lo hacen. El Capítulo 9 la aborda de frente.

Capítulo 9 — Orden

Imagina un jueves por la tarde en Parcelly. Una persona compradora hace un pedido y a los pocos segundos cambia de idea y lo cancela. Orders emite `OrderPlaced`, luego `OrderConfirmed` y luego `OrderCancelled`, todos para `order-81724`, y todos en el orden correcto. El relay también los publica en ese orden.

El topic que alimenta a Shipping se creó con un ajuste por defecto: los mensajes se reparten entre seis particiones sin tener en cuenta su contenido. `OrderConfirmed` cae en la partición 2 y `OrderCancelled` en la partición 5. Shipping ejecuta un consumidor por partición. El consumidor de la partición 5 está libre y es rápido, así que lee primero la cancelación y se encuentra con un envío que todavía no existe. Se encoge de hombros y hace ack. Un momento después, el consumidor de la partición 2, que estaba ocupado con un lote lento, lee la confirmación y programa un envío para un pedido que ya no existe.

Nada se cayó, ningún mensaje se perdió y ningún mensaje se duplicó. Cada componente hizo lo que se le había configurado. Un transportista llegará el lunes a recoger un paquete que nadie pidió. ¿Cuánto orden necesita un sistema como este, dónde se pierde y qué se puede construir cuando no se puede tener?

El problema y por qué es difícil

El orden no es una propiedad única que un sistema tiene o no tiene. Es una cadena, y cada eslabón puede romperlo por su cuenta. El relay puede publicar las filas en una secuencia distinta de la que confirmó la base de datos (el Capítulo 4 y el Capítulo 5 tratan ese eslabón). El broker puede repartir mensajes relacionados entre particiones o colas. Los consumidores pueden procesar en paralelo. Un retry puede poner un mensaje detrás de sus sucesores (Capítulo 6), y una dead letter puede saltárselo por completo (Capítulo 8).

Eslabón	Cómo se pierde el orden	Dónde mirar
Productor y relay	Relays en paralelo, commits concurrentes, ids que no coinciden con el orden de commit	Capítulos 4 y 5
Broker	Mensajes relacionados en distintas particiones o colas	Este capítulo
Consumidor	Varios workers manejando mensajes de una misma partición o cola	Este capítulo
Manejo de fallos	Retry topics, nacks, dead letters, redrive	Capítulos 6 y 8
Cambio de topología	Cambios en el número de particiones, rebalanceos de consumidores	Este capítulo

La respuesta obvia es exigir un orden global: todos los eventos del sistema, en una sola secuencia, vistos por todos los consumidores. Suena como el valor por defecto seguro. Es caro y, para la mayoría de los sistemas, innecesario.

Por qué el orden global es caro

Un único orden total significa un único punto de secuencia. En un broker basado en log, eso es una partición, que como máximo lee un consumidor de un grupo de consumidores a la vez. Tu techo de throughput pasa a ser la velocidad de un escritor y un lector. Añadir capacidad no sirve de nada, porque la capacidad viene de ejecutar cosas en paralelo y el paralelismo es justo lo que un orden total prohíbe.

Un sistema que abarca varios servicios lo empeora. Orders, Payments e Inventory tienen cada uno su propia base de datos y su propio outbox. Un orden global entre todos ellos necesitaría un coordinador que marcara cada evento, y ese coordinador estaría en el camino de escritura de todos los servicios. Estarías recomprando el acoplamiento que la arquitectura se diseñó para eliminar.

Por qué el orden global casi nunca hace falta

Pregunta qué necesita realmente el negocio. A Parcelly no le importa si `PaymentCaptured` de `order-81724` llegó antes que `OrderPlaced` de `order-90311`. Esos pedidos no se influyen entre sí. Lo que importa es que,

para un pedido, `OrderPlaced` se vea antes que `OrderConfirmed`, y `OrderConfirmed` antes que `OrderCancelled`.

Eso es el **orden por agregado**. Un agregado es el límite de consistencia cuyo identificador sirve como clave de partición. Si todos los eventos de un agregado se ven en el orden en que se produjeron, y los eventos de agregados distintos pueden intercalarse libremente, la mayoría de las máquinas de estados funcionan correctamente. Este es el orden que te puedes permitir, y en mi opinión es el orden alrededor del cual debes diseñar.

Hay un límite que ni siquiera el orden por agregado cruza. Los eventos del mismo pedido, producidos por servicios distintos y transportados en topics distintos, no tienen orden relativo entre sí. `PaymentAuthorized` y `StockReserved` pueden llegar a `Orders` en cualquiera de las dos secuencias, que es exactamente la razón por la que el flujo de pedidos que se usa a lo largo de este libro solo solicita la captura después de que hayan llegado los dos. No trates una clave de partición compartida como una promesa entre topics.

Orden por agregado

El mecanismo es sencillo de enunciar y exigente de mantener: elige una clave, envía todo lo que tenga la misma clave por un mismo camino y procesa ese camino de un mensaje a la vez.

La clave de partición

En un broker basado en log, el productor aplica un hash a la `partition_key` para elegir una partición. Todos los mensajes con la misma clave caen en la misma partición, y una partición es una secuencia ordenada. El envelope de este libro lleva `partition_key` por esa razón, y el outbox lo almacena para que el relay pueda fijarlo sin preguntar al servicio productor (Capítulo 4).

La garantía tiene una condición: **misma clave, mismo partitioner, mismo número de particiones**. El partitioner es la función hash más la manera en que la clave se serializa a bytes antes del hash. Si dos productores usan distintos, un mismo `order_id` va a dos particiones y el orden por clave se pierde, sin ningún error en ninguna parte. Es el fallo políglota más concreto del orden. Las bibliotecas cliente de distintos lenguajes no com-

parten todas una misma función hash por defecto, así que un servicio escrito en un lenguaje y un relay escrito en otro pueden enviar el mismo `order_id` a particiones diferentes. Funcionan dos defensas. Fijar el `partitioner` de forma explícita en cada productor, con el mismo algoritmo y la misma codificación de la clave. O quitarles la decisión a los productores: que el relay calcule la partición él mismo a partir de la `partition_key` almacenada, de modo que un único fragmento de código sea responsable de la correspondencia. La segunda es natural cuando todos los eventos ya pasan por el relay del outbox del Capítulo 5.

Los brokers basados en colas son distintos. Los consumidores competidores sobre una cola no te dan ninguna garantía de orden en cuanto hay más de uno, porque el broker entrega el siguiente mensaje a cualquier consumidor que esté libre. Algunos brokers basados en colas ofrecen una función de agrupación que serializa los mensajes que comparten un identificador de grupo, por ejemplo los message groups de las colas FIFO de Amazon SQS, y otros ofrecen un modo en el que solo hay un consumidor activo en una cola a la vez. Estas funciones sirven, y cada una cuesta paralelismo de una manera distinta. Comprueba qué promete tu broker sobre reentrega y visibility timeouts bajo esa función, porque la garantía suele tener letra pequeña. Di en qué familia te apoyas y qué promete realmente esa familia.

Elegir la clave

La clave es el identificador del agregado, y la elección es una declaración sobre dónde vive la consistencia. Para el ciclo de vida del pedido de Parcelly, la clave es `order_id`. Todos los eventos que llevan a un pedido por sus estados la incluyen, lo que da a Shipping, Orders y Ledger una secuencia coherente por pedido.

Unas cuantas preguntas ayudan cuando la respuesta no es obvia:

- ¿El estado de qué entidad cambia el consumidor? Usa esa entidad como clave.
- ¿Cuál es la unidad más pequeña dentro de la cual dos eventos pueden entrar en conflicto? Ese es el agregado.
- ¿El mismo stream lo consumen varios servicios con agregados distintos? Entonces o bien el productor elige el más fino de los comunes, o

bien cada consumidor necesita su propia vista.

Inventory muestra la tensión. Los niveles de stock pertenecen a un SKU, así que los propios comandos de Inventory se indexan naturalmente por `sku`. Los eventos que fluyen de Orders a Inventory se indexan por `order_id`. Un evento de reserva lleva ambos identificadores, e Inventory vuelve a asignar la clave internamente si necesita orden por SKU. Elegir una sola clave para todos los topics del sistema es tentador, y desperdicia el sentido de tener agregados.

Particiones calientes

Una clave de partición reparte la carga de manera uniforme solo si las claves están igual de ocupadas. No lo están. Imagina una venta relámpago en Parcelly en la que el único producto de un vendedor recibe la mayor parte de los pedidos de un día en una hora. Si Inventory indexa su propio topic por `sku`, esa clave produce la mayor parte del tráfico, todo cae en una partición y a un consumidor se le pide el trabajo de todo el grupo mientras los demás están ociosos.

Añadir particiones no ayuda. Una clave siempre se asigna a una única partición, y añadir particiones mueve las demás claves a su alrededor. Hay cuatro respuestas honestas:

- **Hacer más pequeño el agregado.** Si la regla de negocio lo permite, indexa por algo más fino, como un id de reserva en lugar del SKU. Se renuncia al orden dentro del SKU a cambio de reparto, y esto solo es aceptable si ninguna regla exige orden por SKU.
- **Gastar en el consumidor, no en las particiones.** Haz barato el manejo de cada mensaje, de modo que un consumidor pueda seguir el ritmo de la clave caliente.
- **Aislar.** Dirige las claves calientes conocidas a un topic o una partición dedicados, para que no ahoguen el tráfico ordinario que comparte la partición.
- **Aceptar un orden relajado para ese stream.** Añade un sufijo aleatorio a la clave para repartir la carga, y apóyate en las comprobaciones de versión (más adelante en este capítulo) para reparar el orden en el consumidor. Es lo más flexible y lo que más trabajo da. Fíjate en lo que el salting compra y en lo que no: reparte la carga en el broker, pero la

comprobación de versión vuelve a serializar la clave sobre el bloqueo de fila del agregado en la base de datos del consumidor. El throughput solo mejora si el trabajo costoso ocurre antes del paso serializado, o si las operaciones conmutan.

Mediría el sesgo de las claves antes de elegir. Una distribución de claves que parece pareja en una semana tranquila puede estar muy desequilibrada en un día ajetreado, y la primera señal suele ser el retraso del consumidor en exactamente una partición (Capítulo 13).

El paralelismo y el orden tiran en direcciones opuestas

Un grupo de consumidores escala dando cada partición a un consumidor, de modo que el número de particiones es el techo del paralelismo. Si Shipping tiene seis particiones, seis consumidores pueden trabajar en paralelo. Un séptimo estaría ocioso.

Los equipos suelen intentar subir el throughput dentro de un consumidor pasando los mensajes a un thread pool. Parece inofensivo y rompe el orden. Dos mensajes del mismo pedido, leídos uno tras otro, se pasan a dos workers, y el más lento termina segundo. La partición los entregó en orden y el procesamiento no lo preservó.

La forma de conseguir ambas cosas es paralelizar entre claves y serializar dentro de cada clave. Dentro del consumidor, aplica un hash a la clave para asignarla a un lane fijo, y da a cada lane exactamente un worker.

```

function pollLoop():
    while running:
        // bounded lanes: a slow lane pauses intake instead of filling
        memory
        if anyLaneAbove(HIGH_WATER):
            pause(assignedPartitions)
        else if allLanesBelow(LOW_WATER):
            resume(assignedPartitions)
        for message in poll(maxRecords: POLL_LIMIT):
            // the same key always maps to the same lane, so per-key
            order survives
            lanes[hash(message.partition_key) mod
            LANE_COUNT].enqueue(message)
            // commits are per partition and happen here, on the polling
            thread
            for partition in assignedPartitions:
                commit(partition, contiguousDoneOffset(partition) + 1)

function runLane(lane):
    while running:
        message = lane.dequeue()
        try:
            handleMessage(message)
        on error as failure:
            // in place or aside, as decided in Chapter 6
            decideInPlaceOrAside(message)
            markDone(message.partition, message.offset)

```

Tres detalles sostienen el diseño. El primero es el offset. Los commits son por partición, y `contiguousDoneOffset` es el offset más alto por debajo del cual todos los mensajes de esa partición están terminados. Cuando los lanes terminan a distinta velocidad, hacer commit del offset del lane más rápido haría, tras una caída, que se saltaran mensajes que un lane más lento no había terminado. El commit ocurre en el poll loop porque muchos clientes de log no son seguros para llamarse desde varios threads. El precio del paralelismo dentro del consumidor es esa contabilidad, y los duplicados tras una caída, que maneja un inbox del Capítulo 7.

El segundo es el límite. Sin uno, un lane lento mantiene quieto el punto de commit mientras el poll loop sigue llenando la memoria, y la brecha entre el offset confirmado y el trabajo en curso es exactamente el número de mensajes que se reentregan tras una caída. Pausar las particiones cuando un lane pasa su marca de nivel alto convierte la sobrecarga en retraso, que es visible y seguro, en lugar de crecimiento de memoria, que no es ningun-

na de las dos cosas. El mecanismo, y la elección entre pausar y descartar carga, es el que se describe en “Retries bajo sobrecarga: backpressure” en el Capítulo 6.

El tercero es el fallo. Si `handleMessage` falla y el lane reintentará en el sitio, todas las claves que comparten el lane esperan detrás y el offset confirmado deja de avanzar. Enviar el mensaje aparte deja que el lane siga y reordena esa clave. Es la misma decisión que el Capítulo 6 tomó para particiones enteras, ahora tomada por lane, y la respuesta depende del consumidor de la misma manera.

En un broker basado en colas, la misma idea aparece un nivel más arriba. Si necesitas orden por clave entre consumidores competidores, o bien usas la función de agrupación del broker, o bien aceptas que la cola entregará fuera de orden y lo manejas en el consumidor. No supongas que una cola con varios consumidores preserva el orden porque los mensajes entraron ordenados. No lo hace.

Detectar la llegada desordenada con versiones

A veces el broker no puede darte el orden que necesitas, por un retry topic, un redrive, consumidores en paralelo o un cambio en el número de particiones. En esos casos el consumidor tiene que darse cuenta por sí mismo. Para eso, los eventos deben llevar algo que un consumidor pueda comparar.

Los timestamps son lo que no hay que comparar. El Capítulo 3 explicó que `occurred_at` refleja el reloj de una sola máquina, y que usarlo para decidir qué actualización gana permite que un servicio con el reloj adelantado gane todos los conflictos que debería perder. Usa `occurred_at` para las personas y para la analítica.

Lo que funciona es un **número de versión** (también llamado número de secuencia): un contador que el agregado incrementa en cada cambio, escrito en la misma transacción que el cambio de estado y transportado con el evento. Como solo el servicio responsable cambia el agregado, y lo hace dentro de una transacción, las versiones forman una secuencia sin huecos por agregado: 1, 2, 3. Sin huecos exige también una regla en el otro sentido: **cada incremento de versión produce exactamente un evento pu-**

blicado. Una actualización interna que sube la versión sin publicar nada deja un hueco permanente, y todos los consumidores esperarán un evento que no existe.

El envelope estándar no tiene un campo para esto, y yo no cambiaría el envelope por un solo capítulo. Lo llevaría en el payload, como `order_version` en los eventos de Orders. Es la versión del agregado (Capítulo 4), el mismo número que almacena la fila del outbox. Dejo la discusión más profunda sobre el diseño del envelope para el Capítulo 11. El contador pertenece a un productor y a un agregado. Un consumidor compara valores de `order_version` solo entre eventos de Orders del mismo pedido, y nunca con un evento de Payments o Inventory, que necesitaría un contador propio. Una dependencia entre servicios no se resuelve con una secuencia compartida. Se resuelve con una comprobación de estado o de precondition, como hace el paso de captura cuando espera tanto a `PaymentAuthorized` como a `StockReserved`. La fila del outbox lo obtiene gratis, ya que el servicio escribe la nueva versión, el cambio de estado y la fila del outbox en una sola transacción.

La decisión del consumidor

Un consumidor que mantiene una vista de un agregado mantiene también la última versión que aplicó. Para cada evento que llega compara versiones y toma uno de tres caminos.

Versión que llega, comparada con la última aplicada	Significado	Acción
Igual o menor	Duplicado o obsoleta	Ignorar, contar, ack
Exactamente una más	El evento siguiente	Aplicar, registrar la nueva versión
Más de una más	Un hueco: falta algo o llega tarde	Poner el evento en el buffer, no aplicarlo


```

function handleOrderEvent(message):
    key = message.partition_key
    transaction:
        view = loadOrderView(key)
        // loadOrderView locks the row; a concurrent worker waits
here.
        // For the first event there is no row yet: last_version
defaults
        // to 0 and the key's unique constraint settles a concurrent
insert.
        version = message.payload.order_version
        if version <= view.last_version:
            countIgnored(message)
            // a lower version was already applied in sequence: a
duplicate
            else if version == view.last_version + 1:
                applyToView(view, message)
                view.last_version = version
                saveOrderView(view)
                // applying may unblock held events (drain is defined
below)
                drainHeldEvents(key)
            else:
                // insert-if-absent: a redelivered held version must not
fail
                holdEvent(key, version, message)
                // the commit covers the view and the held event together
                ack(message)

```

Dos propiedades lo hacen seguro. La comprobación de versión y la actualización de estado se confirman de forma atómica, de modo que dos workers que manejan la misma clave no pueden aplicar ambos la versión 4. Y la comprobación es idempotente: un duplicado es una versión obsoleta como cualquier otra y se ignora, así que complementa al inbox del Capítulo 7 en lugar de sustituirlo. `holdEvent` también tiene que ser idempotente. Una entrega duplicada de una versión ya retenida chocaría de otro modo con la restricción de unicidad del buffer, haría fallar la transacción, dejaría el mensaje sin ack y se convertiría en un bucle poison.

Con frecuencia se pasan por alto dos precondiciones. La primera es el inicio de la secuencia. Un consumidor que empieza a mitad del stream, como un servicio nuevo o un replay desde un offset posterior, ve primero la versión 5 y esperaría para siempre las versiones 1 a 4. Siembra su vista a partir de un snapshot o de la API del servicio responsable, junto con la versión que refleja ese snapshot, y empieza a comparar desde ahí.

La segunda es que la detección de huecos solo funciona si el consumidor ve todas las versiones. Un consumidor que se suscribe a un subconjunto de los eventos de un agregado, digamos solo `OrderCancelled`, verá las versiones 3 y 7 y concluirá, erróneamente, que faltan de la 4 a la 6. Para esos consumidores la regla se debilita a “nunca aplicar nada que no sea más nuevo que lo que tengo”, que sigue protegiendo contra sobrescrituras obsoletas pero no puede detectar huecos.

Las trampas de last-writer-wins

La política de conflictos más simple es que gane la última escritura. Es simple y esconde tres trampas distintas, según lo que signifique “última”.

La última en llegar. La vista se sobrescribe con el mensaje que el consumidor manejó más recientemente. Un `OrderPlaced` reentregado que llega después de `OrderConfirmed` devuelve el estado a `PLACED`, que es el fallo al que apuntó el Capítulo 7. El orden de llegada es lo único que la topología del broker puede no prometer.

La última por timestamp. Mejor en apariencia, peor en la práctica. El desfase de relojes entre servicios hace que el evento ganador pueda ser el del reloj más adelantado. Dos eventos escritos con milisegundos de diferencia en máquinas distintas tienen un orden de timestamps sin significado.

La última por versión. Esto funciona, y tiene su propio límite. Descartar un evento obsoleto solo es correcto si el evento más nuevo ya contiene todo lo que decía el obsoleto. Eso es cierto para los eventos que llevan estado (“el pedido ahora es `CANCELLED`”). Es falso para los eventos que llevan cambios, como “suma 4,990 al saldo”. Ignorar un evento de cambio obsoleto pierde el cambio.

Una aclaración evita que esto contradiga la tabla de arriba. Con versiones sin huecos aplicadas estrictamente en secuencia, una versión menor es siempre un duplicado, porque ya se aplicó; ignorarla es correcto incluso para los deltas. La trampa de los deltas aparece cuando un consumidor salta a la versión más nueva y se salta un hueco, porque entonces los cambios omitidos se pierden.

Esta distinción decide qué haces ante una versión obsoleta o desordenada:

- **Eventos que llevan estado** (el estado completo, o un estado que sobrescribe al anterior): ignorar el evento obsoleto es correcto, porque el posterior lo reemplaza.
- **Eventos de delta o de transición** (incrementos, asientos del libro mayor, cualquier cosa que no reemplace): nunca los descartes. Ponlos en un buffer y reordénalos, o procésalos en cualquier orden si la operación es conmutativa. Ledger en particular no debe descartar un asiento porque sea más antiguo.

Una trampa relacionada es el agregado borrado. Si un consumidor borra su fila al ver `OrderCancelled`, un `OrderPlaced` tardío no encuentra fila, supera la comprobación de versión sin nada con qué comparar y crea un pedido nuevo. La resurrección es el bug característico de ignorar el orden alrededor de los borrados. Conserva un tombstone: la clave, la última versión y una marca de borrado, al menos mientras puedan seguir llegando eventos obsoletos. El Capítulo 8 mostró la misma resurrección cuando un `OrderPlaced` con redrive llega después de una cancelación, y un tombstone con una comprobación de versión es lo que la detiene.

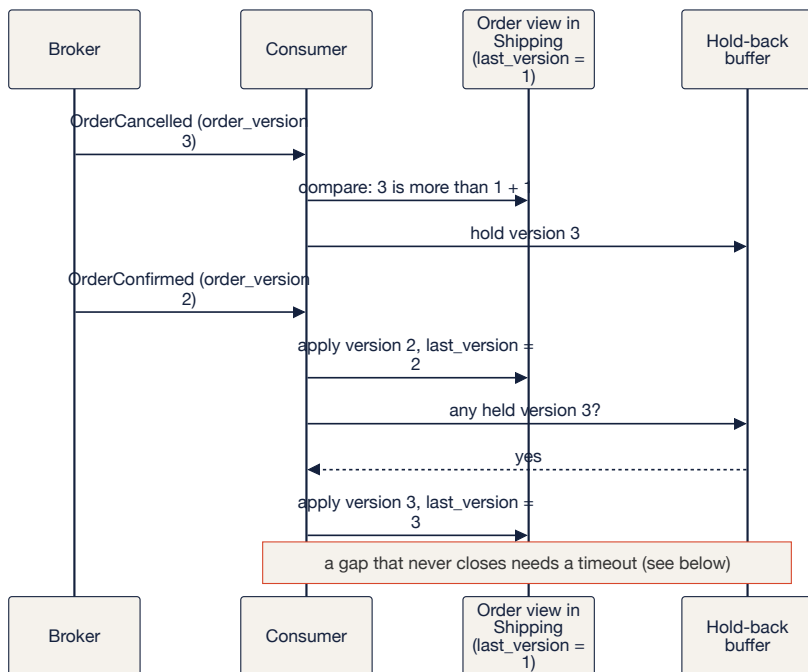
Por último, ignorar es una decisión que merece dejar rastro. Un evento obsoleto descartado en silencio es un bug que nadie puede ver. Cuenta los eventos ignorados por consumidor y guarda la clave y las versiones en una línea de log, para que un pico en la cuenta se vea antes de que lo note un cliente (Capítulo 13).

Buffering y reordenamiento

Cuando el consumidor ve un hueco, tiene dos caminos. Puede aplicar el evento de todos modos y arriesgarse a un estado erróneo, o puede retener el evento hasta que llegue el que falta. La segunda opción es un **hold-back buffer**.

El diagrama de secuencia siguiente muestra a Shipping recibiendo de Orders dos eventos de `order-81724`, la versión 3 antes que la 2, como en el escenario inicial. Los dos vienen de un productor y de un agregado, que es lo que hace significativo comparar sus versiones. Shipping solo actúa sobre `OrderConfirmed`, pero para que esto funcione lee todo el stream de Orders de ese pedido y registra la versión de cada evento, incluidos los tipos

que de otro modo ignora. Eso es lo que hace real el `last_version = 1` de la vista, y lo que hace posible la detección de huecos. Observa dónde espera la versión 3 y qué la libera.



El paso de liberación es un bucle. Después de aplicar cualquier evento, el consumidor comprueba si el buffer contiene la versión siguiente, la aplica y repite hasta que el buffer no tenga sucesor.

```

function drainHeldEvents(key):
    // runs inside the same transaction that applied the triggering
    event
    while true:
        view = loadOrderView(key)
        next = findHeldEvent(key, view.last_version + 1)
        if next is null:
            return
        applyToView(view, next.message)
        view.last_version = next.version
        saveOrderView(view)
        deleteHeldEvent(key, next.version)
  
```

El propio buffer es una tabla indexada por el agregado y la versión, de modo que sobrevive a un reinicio. Un buffer en memoria desaparece con el proceso, y el broker ya ha considerado esos mensajes como

confirmados.

La limpieza necesita una regla más. Cada vez que `last_version` da un salto, por ejemplo tras reemplazar la vista con un estado obtenido de otro lado, las filas retenidas con `version <= last_version` quedan obsoletas, y el consumidor las borra en la misma transacción. De lo contrario se quedan en la tabla para siempre, y el bucle de drenaje nunca las alcanza.

El hueco que nunca se cierra

El buffering es optimista: supone que el evento que falta simplemente llega tarde. A veces se perdió, se envió a dead letter o nunca se envió a este consumidor. Sin un límite, el agregado queda bloqueado para siempre y el conjunto retenido crece.

Todo buffer necesita un timeout y una decisión sobre qué ocurre cuando vence. Hay tres respuestas razonables, y yo elegiría según el costo de equivocarse:

- **Obtener el estado que falta.** Pedir al servicio responsable el estado actual mediante una API y reemplazar la vista. Funciona cuando el productor expone una API así, y convierte el buffer en una vía rápida con una red de seguridad.
- **Alertar y esperar.** Una persona mira. Es lo correcto para Ledger, donde un asiento erróneo cuesta más que uno lento.
- **Marcar la vista como sospechosa y dejar que la conciliación la repare.** Es lo correcto para los modelos de lectura que toleran estar obsoletos (Capítulo 13).

El bloqueo por clave del Capítulo 8 es la forma sin versiones de este mecanismo. Allí, una marca de dead letter bloquea la clave, los eventos posteriores esperan en un almacén de retenidos y un bypass especial deja pasar el evento con redrive para liberarlos. Aquí una dead letter de un evento crea un hueco, los eventos posteriores esperan en el buffer y el evento con redrive los libera por sí mismo, porque llega como `last_version + 1` y el bucle de drenaje hace el resto. Las versiones sustituyen tanto a la marca como al bypass, y por eso este es el diseño más fuerte donde el productor puede proporcionarlas.

Los costos son reales: una tabla de buffer que dimensionar y limpiar, una política de timeout que afinar y una nueva clase de agregado atascado por la que alguien necesita recibir un page. Si tus eventos llevan el estado completo y tus consumidores pueden aplicar con seguridad “gana el más nuevo”, el buffer es innecesario y yo lo dejaría fuera.

Retries, dead letters y orden

El Capítulo 6 y el Capítulo 8 terminaron cada uno con una pregunta sobre el orden que dejaron para aquí. La forma corta es que todo mecanismo de manejo de fallos mueve un mensaje en el tiempo, y eso es un reordenamiento.

- **Un retry en el sitio** en una partición de log bloquea a los sucesores, así que el orden se conserva y toda la partición paga en latencia.
- **Un retry topic o una cola con retardo** deja que los sucesores sigan adelante, así que el mensaje fallido vuelve más tarde que eventos a los que precedía.
- **Un visibility timeout** en un broker basado en colas devuelve un mensaje a la cola tras el timeout, donde compite con mensajes que se enviaron después de él.
- **Una dead letter** salta el mensaje, y un **redrive** lo trae de vuelta después de todo lo que vino a continuación.

La comprobación de versión es lo que hace sobrevivibles las opciones segunda a cuarta. Toma el ejemplo que dio el Capítulo 6: Inventory recibe de Orders `OrderPlaced` y luego `OrderCancelled` de un mismo pedido, y `OrderPlaced` pasa a un retry topic tras un fallo transitorio. Con la versión del agregado de Orders, `OrderPlaced` es la versión 1 y `OrderCancelled` es la versión 2. La cancelación llega primero, e Inventory aún no tiene vista de ese pedido, así que `last_version` toma por defecto el valor 0 y 2 es más que $0 + 1$. La cancelación se retiene, no se aplica contra una reserva que no existe. Cuando el `OrderPlaced` reintentado llega como versión 1, se aplica, y el bucle de drenaje libera la cancelación. La reserva huérfana que describió el Capítulo 6 no aparece.

Esto solo funciona para consumidores que mantienen versiones y un buffer. Uno que no lo haga debe elegir, como argumentó el Capítulo 6, entre bloquear la partición y tolerar el reordenamiento. Y el consejo del Capítu-

lo 8 para los libros mayores sigue en pie: para un consumidor en el que un asiento tardío o ausente significa un registro financiero erróneo, detente y espera. Un consumidor en pausa es ruidoso, y un Ledger que aplica asientos fuera de orden en silencio no lo es.

Advertencia: Un retry topic combinado con un consumidor sin comprobación de versión reordena los eventos en silencio. Nada falla, ninguna alerta salta y el estado es erróneo. Si desvías los fallos a un lado, haz que la comprobación de versión forme parte del mismo cambio.

Cambiar el número de particiones

La partición de una clave es una función de la clave y del número de particiones, normalmente un hash módulo la cantidad. Cambia la cantidad y cambia la función.

Supón, a modo de ilustración, que una clave tiene hash 17. Con cuatro particiones se asigna a la partición 1, y con seis particiones a la partición 5. Un hash de 20 pasa de la partición 0 a la partición 2, mientras que el 14 y el 15 se quedan donde estaban. La mayoría de las claves se mueven. Con hashing módulo, una clave conserva su partición al pasar de cuatro a seis solo cuando su hash módulo 12 es 0, 1, 2 o 3, de modo que cerca de dos tercios de las claves terminan en otro sitio. El fallo no es raro: afecta a toda clave que tenga mensajes en vuelo durante el cambio.

El peligro es el solapamiento. Los mensajes de una clave que se mueve escritos antes del cambio están en la partición antigua, todavía sin consumir. Los mensajes escritos después del cambio van a la partición nueva. Ahora dos consumidores procesan la misma clave al mismo tiempo, sin orden entre ellos. Si además un consumidor mantiene estado local por clave, como un cache o una ventana de agregación, ese estado está en la instancia equivocada.

Formas de abordarlo, en orden aproximado de preferencia:

1. **Aprovisionar suficientes particiones desde el principio.** Aprovisiona de más respecto al pico esperado de paralelismo, porque el número es barato de mantener y caro de cambiar. Muchos brokers basados en

log permiten añadir particiones pero no quitarlas, así que un error en una dirección es permanente.

2. **Drenar antes del cambio.** Detén los productores o pausa el stream, deja que los consumidores lleguen al final de las particiones antiguas y luego cambia la cantidad. El orden se conserva al costo de una ventana de indisponibilidad.
3. **Migrar a un topic nuevo.** Crea un topic con la cantidad correcta, ejecuta un cutover en el que los consumidores terminen el topic antiguo antes de empezar el nuevo y retira el antiguo. Es lo más seguro y lo que más trabajo da.
4. **Apoyarse en las comprobaciones de versión.** Si todos los consumidores detectan y ponen en buffer los eventos desordenados, la transición es sobrevivible sin pausa, y la propia transición es además una buena prueba de si la comprobación funciona.

Rebalanceos y revocación

Los rebalanceos de los grupos de consumidores tienen una forma parecida a menor escala, y ocurren en cada deploy. Cuando una partición pasa a otro consumidor, el trabajo en curso en el antiguo puede solaparse con los primeros mensajes en el nuevo. Lo que haga el consumidor en la revocación decide qué tan grande es ese solapamiento.

En la revocación, el consumidor deja de despachar mensajes nuevos de las particiones revocadas, deja que los lanes terminen lo que tienen o lo abandona, y hace commit del offset contiguo antes de ceder la partición. Que un rebalanceo revoque todas las particiones o solo las que se mueven depende de la estrategia de asignación de tu cliente, y una estrategia de todo a la vez detiene a todo el grupo mientras dura; comprueba cuál usas. Un consumidor que tarda en darse cuenta de que perdió una partición sigue procesando como un zombie mientras el nuevo responsable hace el mismo trabajo, así que espera algo de solapamiento hagas lo que hagas.

El inbox y la comprobación de versión mantienen inofensivo ese solapamiento solo para los efectos dentro de la transacción de la propia base de datos del consumidor. No hacen nada por un correo o por una llamada a la API de un transportista hecha por los dos consumidores. Esos efectos secundarios externos necesitan su propia clave de idempotencia, como describe el Capítulo 7, porque el inbox protege el estado local y nada más.

Qué necesita cada consumidor de Parcelly

No todos los consumidores necesitan el mismo orden, y tratarlos igual es fuente tanto de bugs como de desperdicio. Una tabla, dibujada para Parcelly, muestra cómo lo decidiría yo.

Consumidor	Necesidad de orden	Mecanismo	Si falta un mensaje
Notifications	Ninguna; cada mensaje es un hecho independiente	Cualquier orden, sin necesidad de clave	Dead letter y continuar
Vista de estado de Orders	Por pedido, con eventos que llevan estado	Clave <code>order_id</code> , ignorar versiones obsoletas	Obtener el estado actual
Reservas de Inventory	Por pedido, transiciones	Clave <code>order_id</code> , versiones más hold-back buffer	Retener la clave, alertar
Shipping	Por pedido, <code>OrderConfirmed</code> antes de <code>OrderCancelled</code>	Clave <code>order_id</code> , versiones más hold-back buffer	Retener la clave, alertar
Ledger	Por pedido, sin huecos, sin descartes	Clave <code>order_id</code> , detenerse ante un hueco en la secuencia por pedido que pertenece a Payments	Detenerse y esperar a una persona

La fila de Ledger nombra su contador a propósito. Ledger consume eventos de Payments, así que compara una secuencia que Payments incrementa por pedido, no `order_version`, que pertenece a Orders. Un contador solo es comparable entre eventos del único servicio que es su responsable.

La primera fila es la más barata y merece más crédito del que recibe. La mejor manera de manejar el orden es diseñar el consumidor de modo que no importe. Una operación conmutativa, como añadir un elemento a un conjunto o tomar el máximo de dos valores, da el mismo resultado en cualquier orden. Un consumidor construido con esas operaciones no necesita clave, ni buffer, ni versión. Cuando puedes remodelar un handler de esa manera, yo lo haría antes de recurrir a la maquinaria.

Trade-offs y costos

Estás cambiando paralelismo por orden, y el tipo de cambio es el agregado. Una clave más gruesa da un orden más fuerte y menos reparto. Una clave más fina da reparto y un orden más débil. No hay ninguna clave que dé ambas cosas, y la decisión corresponde a quien conoce las reglas de negocio.

Las claves calientes son un costo de diseño, no un accidente operativo. Una clave de partición elegida a partir de un identificador de negocio hereda el sesgo del negocio. Una venta relámpago lo revela. Puedes mitigarlo, y no puedes eliminarlo sin renunciar al orden de esa clave.

Las versiones exigen disciplina a los productores. El agregado debe incrementar la versión en la misma transacción que el cambio de estado, en todos los cambios, desde todos los caminos de código. Un solo camino que lo olvide deja un hueco que los consumidores esperarán para siempre. También vale lo inverso: cada incremento debe publicar exactamente un evento, o la secuencia tiene un hueco que ningún consumidor puede cerrar. Los backfills, las herramientas de administración y las migraciones de datos son donde esto falla, y un consumidor que empieza a mitad del stream necesita una vista sembrada antes de poder comparar nada.

Los buffers añaden estado y un fallo nuevo. Una tabla de retención necesita dimensionarse, limpiarse, tener un timeout, un responsable y una alerta. Es además un lugar más donde puede esconderse un agregado atascado. Para los consumidores que pueden aplicar “gana el más nuevo”, sáltatelo.

Las comprobaciones de versión no ordenan entre agregados ni entre topics. Protegen la secuencia de un agregado. Si una regla necesita saber que un evento del topic A vino antes que un evento del topic B, un número de versión no lo proporciona. Eso es un problema de flujo de trabajo, y el Capítulo 10 lo trata como tal.

Cuándo no molestarse. Si el consumidor es conmutativo, si los eventos llevan el estado completo y “gana el más nuevo” es correcto, o si un evento tardío es inofensivo y una conciliación periódica lo reparará, sáltate la maquinaria. Hacer menos es una decisión con un costo, y debería ser una decisión registrada.

Errores comunes

- **Pedir orden global.** Parece el valor por defecto seguro y limita el throughput a un solo lector. Enuncia el requisito real, el orden por agregado, y diseña para eso.
- **Indexar por algo que no es el agregado.** Indexar por tipo de evento, por servicio o por un valor aleatorio ocurre porque el valor por defecto lo hace por ti. La solución es indexar por el identificador de la entidad cuyo estado cambia el consumidor.
- **Usar timestamps para ordenar eventos.** Parecen números de versión naturales. El desfase de relojes vuelve el orden insignificante, así que lleva en su lugar una versión del agregado.
- **Paralelizar dentro de un consumidor sin lanes por clave.** Un thread pool sube el throughput y rompe el orden de la partición. Aplica un hash a la clave hacia un lane y da a cada lane un worker.
- **Descartar eventos delta obsoletos.** Una comprobación de versión conviene a los eventos que llevan estado. Aplicada a incrementos y asientos del libro mayor, descarta cambios reales. Clasifica cada evento como de estado o de delta antes de elegir una política.
- **Olvidar los tombstones.** Borrar una fila al cancelar deja que un evento tardío recree el pedido. Conserva la clave y la última versión después del borrado mientras puedan llegar eventos obsoletos.
- **Retener eventos en memoria.** Un buffer que muere con el proceso pierde eventos que el broker ya considera entregados. Persístelo, en la misma transacción que la vista.
- **Cambiar el número de particiones en producción sin un plan.** La correspondencia cambia para cerca de dos tercios de las claves al pasar de cuatro a seis particiones, y el orden cambia en silencio con ella. Drena, migra o protégete con versiones.
- **Mezclar partitioners entre productores.** Los servicios en distintos lenguajes pueden aplicar un hash distinto a la misma clave. Fija un único partitioner en todas partes, o calcula la partición en el relay.
- **Lanes sin límite.** Un lane lento llena entonces la memoria mientras el punto de commit se queda quieto. Pon un límite a cada lane, pausa las particiones y haz commit por partición desde el poll loop.

- **Confiar en el inbox para los efectos externos durante un rebalanceo.** El inbox cubre solo la base de datos del propio consumidor. Un correo o una llamada a un transportista necesita su propia clave de idempotencia.
- **Suponer que las claves compartidas ordenan eventos entre topics.** El mismo `order_id` en dos topics no da ninguna secuencia entre ellos. Haz que el consumidor espere a ambos, o modela la dependencia en un flujo de trabajo.

Ideas clave

- El orden global te cuesta todo el paralelismo y rara vez hace falta; el orden por agregado es el orden por el que vale la pena diseñar.
- Una clave de partición que identifica al agregado da eventos ordenados por entidad en un broker basado en log, pero solo con el mismo partitioner y el mismo número de particiones en todas partes; los brokers basados en colas necesitan una función de agrupación o un consumidor que tolere el desorden.
- El paralelismo dentro de un consumidor debe ser por clave, con un worker por lane, con límite y con commit por partición, o deshace en silencio el orden que entregó la partición.
- Los números de versión que el agregado incrementa dentro de su transacción permiten a un consumidor detectar eventos obsoletos, duplicados y faltantes, mientras que los timestamps no pueden.
- Ignorar un evento obsoleto es correcto para los eventos que llevan estado e incorrecto para los deltas y los asientos del libro mayor, que deben ponerse en un buffer o aplicarse en cualquier orden.
- Los retry topics, el redrive y los cambios de particiones reordenan todos los mensajes, así que cualquier consumidor que dependa del orden necesita una comprobación de versión antes de usarlos.
- Todo hold-back buffer necesita un timeout, un responsable y una alerta, porque un hueco que nunca se cierra es un agregado atascado; el bloqueo por clave del Capítulo 8 es la forma sin versiones de la misma idea.

El orden da a cada evento un lugar dentro de un agregado, pero un proceso de negocio abarca varios agregados y servicios, y algunos de sus pasos tienen que deshacerse cuando falla uno posterior. El Capítulo 10 se ocupa de las sagas, donde el orden de los pasos es el diseño y la compensación es la manera en que un flujo de trabajo se recupera.

PARTE IV

Flujos de trabajo y contratos

Capítulo 10 — Sagas y flujos de trabajo de larga duración

Imagina un martes por la tarde en Parcelly. Una persona compradora hace un pedido de una lámpara de cerámica. Orders emite `OrderPlaced`, Payments autoriza la tarjeta e Inventory reserva la última lámpara del stock del vendedor. Entonces la tarjeta es rechazada y Payments emite `PaymentFailed`. En ese mismo instante, una segunda persona compradora mira una página de producto que dice “agotado”, porque la reserva del primer pedido sigue reteniendo la lámpara.

Nadie hizo nada mal. Cada servicio cumplió con lo suyo, cada transacción local se confirmó y todos los mensajes se entregaron. El problema es que la operación de negocio, “venderle esta lámpara a esta persona compradora”, está repartida entre cuatro servicios y varios minutos, y ninguna transacción la cubre. La mitad ocurrió. La otra mitad no ocurrirá, y alguien tiene que devolver la lámpara.

El momento importa. Todavía no se ha capturado nada, así que deshacer es barato: se libera la reserva y el pedido termina. Si la lámpara ya se hubiera pagado y el fallo llegara después, devolverla sería un acto distinto y más caro, y el capítulo vuelve sobre esa línea.

Ahora supón que el mismo fallo llega mientras el vendedor cancela la publicación, mientras la persona compradora edita la dirección de entrega o mientras la respuesta de la captura se pierde en el camino de vuelta. Este capítulo trata de la maquinaria que permite que un flujo como este termine limpiamente o se deshaga limpiamente. ¿Cómo logras que un flujo de trabajo de varios servicios termine en un estado que el negocio acepte, cuando no puedes mantener un bloqueo a lo largo de todo él y cualquier paso puede fallar?

El problema y por qué es difícil

Dentro de una base de datos, una transacción te da atomicidad: o se producen todos los cambios o no se produce ninguno. Entre servicios no la tienes. Cada servicio es propietario de su base de datos, y el message broker se interpone entre ellos. El Capítulo 4 mostró cómo hacer atómicos el cambio de estado de un servicio y el evento que publica. No hizo atómica la reacción del siguiente servicio con el cambio del primero, y no puede hacerlo.

La respuesta clásica eran las transacciones distribuidas, normalmente two-phase commit. Mantienen bloqueos en cada participante hasta que un coordinador decide, y si el coordinador falla después de que los participantes hayan votado, estos esperan con esos bloqueos retenidos hasta que vuelva. Exigen participantes que admitan el protocolo y hacen que el participante más lento o menos disponible marque el límite de todos. Un proveedor de pagos externo o la API de un transportista no participa en tu transacción en absoluto. Para un flujo que incluye una llamada a un tercero, esa opción queda cerrada antes de empezar a sopesarla.

Una **saga** es la alternativa. Divide una operación de negocio en una secuencia de transacciones locales, una por servicio. Cuando un paso falla, la saga no hace rollback, porque las transacciones anteriores ya se confirmaron y otras personas pueden haber leído sus resultados. Ejecuta **compensaciones**: transacciones locales nuevas que deshacen semánticamente los pasos anteriores. Una compensación es una acción hacia adelante que devuelve el negocio a un estado aceptable, no una máquina del tiempo.

Tres cosas lo hacen difícil en la práctica.

Primero, los estados intermedios son visibles. Entre “stock reservado” y “pago capturado”, el pedido no está ni confirmado ni cancelado, y todo lo que lo lee ve ese estado intermedio. La atomicidad desaparece; el aislamiento también.

Segundo, la compensación es una decisión de diseño, no un mecanismo. Algunos pasos tienen un inverso limpio, como liberar una reserva. Otros solo tienen uno aproximado, como reembolsar un pago capturado, que cuesta comisiones y tarda días. Algunos no tienen ninguno: un correo que salió ya está enviado.

Tercero, la saga misma puede fallar. El proceso que conduce el flujo de trabajo puede caerse entre pasos, un mensaje puede llegar dos veces, una respuesta puede no llegar nunca, y una compensación puede fallar igual que una acción. Todo lo que los capítulos 4 a 9 construyeron para mensajes individuales ahora tiene que sostenerse para un flujo completo.

Nota: Una prueba útil antes de adoptar una saga: enumera cada paso anterior al pivot (el paso después del cual la saga solo avanza, definido más adelante en este capítulo) con su compensación y di cuánto cuesta. Los pasos posteriores al pivot están pensados para no tener ninguna. Si varios pasos anteriores dicen “ninguna” o “un ticket de soporte”, quizá la frontera entre esos servicios esté en el lugar equivocado, y mover los pasos a la transacción local de un solo servicio puede salir más barato que coordinarlos.

El pedido de Parcelly como saga

Toma el flujo del happy path y ponles nombre a sus pasos como saga. Cada paso es una transacción local en un servicio, seguida de un evento que se escribe en el transactional outbox de ese servicio dentro de la misma transacción.

Paso	Servi- cio	Transacción local	Compensación
1	Orders	Crear el pedido en estado pendiente, emitir <code>OrderPlaced</code>	Marcarlo como cancelado, emitir <code>OrderCancelled</code>
2	Pay- ments	Autorizar la tarjeta, emitir <code>PaymentAuthorized</code>	Anular la autorización, emitir <code>PaymentVoided</code>
3	Inven- tory	Reservar stock, emitir <code>StockReserved</code>	Liberar la reserva, emitir <code>StockReleased</code>
4	Pay- ments	Capturar el pago, emitir <code>PaymentCaptured</code>	Ninguna dentro de la saga: este es el pivot (un reembolso pertenece a un flujo de cancelación aparte)
5	Orders	Confirmar el pedido, emitir <code>OrderConfirmed</code>	Ninguna: se reintenta hasta que tenga éxito

Los dos pasos de Payments no son transacciones de base de datos en sentido estricto. Cada uno es una llamada a un proveedor externo con una clave de idempotencia estable, seguida de un registro local del resultado, que es el patrón del Capítulo 7. Las garantías de la saga descansan en ese emparejamiento, y un timeout en la llamada al proveedor arrastra la incertidumbre que describió el Capítulo 3.

Los pasos 2 y 3 se ejecutan en paralelo, y la captura se solicita solo después de que hayan llegado tanto `PaymentAuthorized` como `StockReserved`. Shipping reacciona a `OrderConfirmed` y Notifications reacciona a lo que emite Shipping. Por ahora trataremos esos dos últimos como ajenos a la saga y volveremos sobre el porqué.

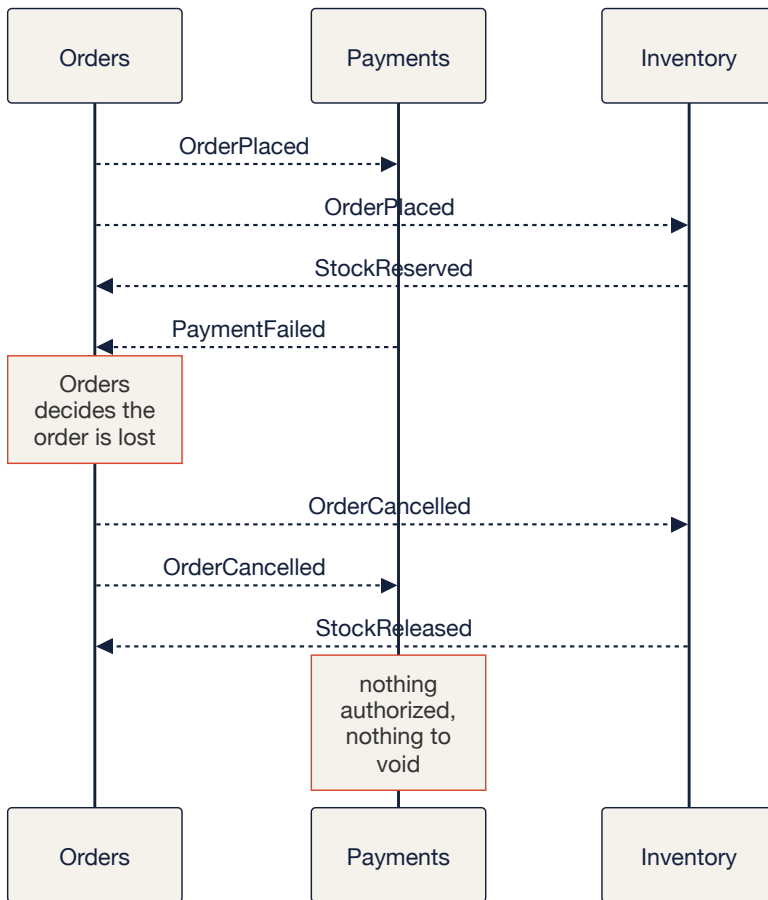
Las ramas de fallo siguen la tabla a partir del flujo de eventos. Si llega `PaymentFailed`, no existe ninguna autorización, así que la única compensación es que Inventory libere una reserva que pudo haberse hecho o no, emitiendo `StockReleased`. Si llega `StockReservationFailed`, hay una autorización retenida y nada capturado, así que Payments la anula y emite `PaymentVoided`. Un reembolso aparece solo después de la captura. Antes de la captura, el deshacer es una liberación o una anulación, y llamarlo reembolso sería incorrecto en el libro mayor e incorrecto en el extracto bancario de quien compró.

En ambas ramas de fallo Orders reacciona y emite `OrderCancelled`, y los demás servicios compensan en respuesta a él. Esa frase es todo el diseño de la coreografía. Si es todo el diseño que quieres es la siguiente pregunta.

Coreografía

En una saga **coreografiada** no hay un responsable central. Cada servicio escucha eventos, hace su transacción local y publica el siguiente evento. El flujo de trabajo existe solo como la suma de esas suscripciones.

El diagrama siguiente muestra la rama de fallo del pago como una coreografía. Fíjate en quién decide qué: nadie da una instrucción, y cada reacción la dispara un evento al que el servicio ya está suscrito.



Las ventajas son reales. Los servicios siguen débilmente acoplados: Inventory sabe qué significa `OrderCancelled` y nada de Payments. No hay un único componente cuyo fallo detenga todos los flujos. Añadir un consumidor, como un servicio de analítica que reacciona a `OrderConfirmed`, no exige cambiar a los demás.

Las debilidades aparecen a medida que el flujo crece. La definición del flujo de trabajo queda desperdigada entre suscripciones en varias bases de código, y la pregunta “¿qué ocurre después de `StockReserved`?” no se puede responder leyendo un solo archivo. Se cuelan dependencias cíclicas: Orders escucha a Payments, que escucha a Orders. Cada rama nueva añade eventos y hay que actualizar todos los servicios en un orden cuidadoso. Sobre todo, nadie es responsable de la pregunta “¿dónde está ahora el pe-

dido 81724?”. La respuesta es un join entre los logs de varios servicios, que es el problema que nombró el Capítulo 1 y al que el Capítulo 13 vuelve con tracing y conciliación.

Creo que la coreografía funciona bien cuando el flujo es corto, tiene pocas ramas y ningún paso depende de una conjunción de pasos anteriores. Dos o tres servicios que reaccionan a un mismo evento encajan de forma natural. El flujo del pedido de Parcelly ya pasó esa línea, porque la captura necesita que hayan llegado dos resultados previos.

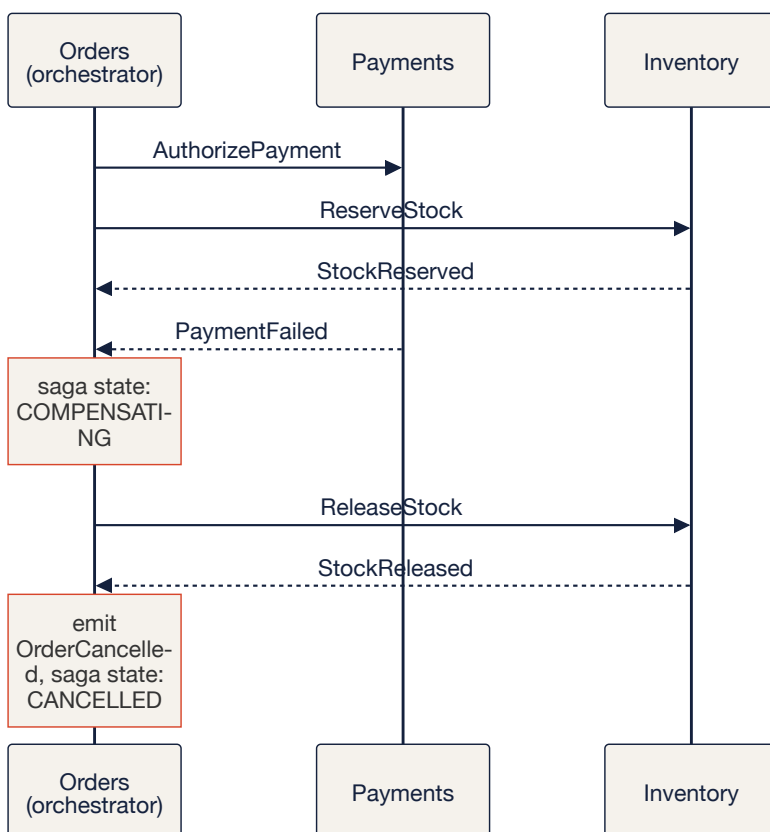
Orquestación

En una saga **orquestada**, un componente es responsable del flujo de trabajo. Guarda el estado, decide el siguiente paso, envía comandos a los participantes y reacciona a sus respuestas. Los participantes saben hacer su trabajo local e informar del resultado; no saben dónde se ubican dentro del flujo más amplio.

Esta sección introduce dos comandos además de los canónicos:

`AuthorizePayment` (de Orders a Payments) y `ReserveStock` (de Orders a Inventory). La confirmación no necesita comando, porque es la transacción local del propio orquestador. Como todos los comandos de este libro, llevan un `command_id` en lugar de `event_id`, y las respuestas son los eventos ordinarios. En la versión coreografiada, Payments e Inventory arrancaban con `OrderPlaced`; aquí el orquestador les dice que lo hagan.

El diagrama muestra el mismo fallo de antes, pero ahora cada flecha que sale de Orders es un comando. Fíjate en cómo el estado de la saga vive en un solo lugar y en cómo los comandos de compensación nombran exactamente el trabajo que hay que deshacer.



El orquestador es el lugar donde se responde la pregunta “¿qué pasa ahora?”. Es una máquina de estados, y una máquina de estados se puede leer, probar y dibujar. Una rama nueva es un cambio en un solo lugar. El orquestador sabe qué pasos se completaron, así que una compensación apunta a lo que realmente ocurrió en vez de a todo lo que pudo haber ocurrido.

Los costos son la imagen espejular. El orquestador conoce a todos los participantes y se convierte en un punto de acoplamiento. Si se descuida, la lógica de dominio se fuga de los servicios hacia él, y los participantes se vuelven llamadas a procedimientos remotos con pasos extra. Además es un componente que debe ser correcto, estar siempre activo y ser él mismo recuperable, pues un orquestador caído que perdió su estado es una saga que nadie terminará jamás.

En mi opinión, un orquestador que vive dentro del servicio responsable del proceso de negocio es la forma más defendible. En Parcelly ese servicio es Orders, que ya es responsable del ciclo de vida del pedido. Un servicio de orquestación independiente que es responsable de los flujos de trabajo de todos tiende a convertirse en lo que cada equipo tiene que cambiar para poder entregar cualquier cosa.

Cómo elegir entre ambas

Ningún estilo es mejor en general. La elección depende de la forma del flujo y de quién tiene que responder preguntas sobre él.

Pregunta	Coreografía	Orquestación
¿Dónde se define el flujo?	Repartido entre suscripciones	En una sola máquina de estados
Acoplamiento entre servicios	Solo a través de los contratos de eventos	El orquestador conoce a cada participante
Añadir un paso o una rama	Tocar varios servicios	Tocar el orquestador y el participante
Pasos paralelos y joins	Incómodo; cada servicio registra lo que ha visto	Natural; el orquestador guarda el join
“¿Dónde está el pedido 81724?”	Join entre logs	Leer una fila
Punto único de fallo	Ninguno, pero tampoco hay responsable	El orquestador, que debe ser recuperable
Riesgo al crecer	Dependencias cíclicas y ocultas	Un servicio todopoderoso lleno de lógica de dominio
Buen encaje	Flujos cortos, pocas ramas	Flujos largos, condiciones, timeouts, joins

Combinar ambos es normal y sensato. Una saga de pedido orquestada puede seguir emitiendo `OrderConfirmed` como un evento simple al que Shipping y Notifications reaccionan de forma coreografiada. La saga debería cubrir la parte del flujo que necesita deshacerse de forma coordinada; las notificaciones y la analítica no lo necesitan, y meterlas solo añade estados.

Compensaciones y pasos que no se pueden deshacer

Diseña la compensación de un paso cuando diseñas el paso, no cuando el primer incidente la pide. Para cada paso, anota qué quiere el negocio cuando la saga falla después de él.

Unas cuantas reglas mantienen honestas las compensaciones.

Una compensación es una acción de negocio nueva, no un deshacer de la base de datos. `StockReleased` no borra el historial de la fila de la reserva. Registra que la reserva terminó. `PaymentRefunded` no borra la captura; el libro mayor conserva ambos asientos. Por eso Ledger es append-only y por eso las compensaciones aparecen ahí como eventos reales.

Ordena los pasos según lo difícil que sea deshacerlos. Pon primero los pasos fáciles de revertir y al final los difíciles o imposibles de revertir. Autorizar y reservar son baratos de deshacer, por eso van antes de capturar. La captura es el punto en que un reembolso empieza a costar dinero y tiempo. De aquí sale la idea de **paso pivot**: el punto a partir del cual la saga ya no compensa y solo avanza. En Parcelly, la captura es el pivot. Antes de ella, un fallo significa liberar y anular. Después, un fallo en la confirmación no es retirada sino retry, porque el negocio tiene el dinero de la persona compradora y le debe un pedido.

Algunas acciones no tienen inverso. Un correo que ya se envió, un transportista que ya recibió el envío, un webhook sobre el que el sistema de un vendedor ya actuó: puedes enviar un seguimiento, pero no puedes desenviar. Estos pasos van después del pivot, y su fallo debe tratarse con retry y con corrección, no con compensación. Por eso mismo Notifications es puramente reactivo y nunca forma parte de la saga: sus pasos no se pueden compensar y no deberían poder bloquear nada.

Las compensaciones deben reintentarse hasta que tengan éxito. El fallo de una acción lleva a una compensación. El fallo de una compensación no lleva a nada, salvo a una saga atascada y un sistema inconsistente. Trata la compensación como algo que debe completarse al final, con la disciplina de retries del Capítulo 6 y el manejo de poison messages del Capítu-

lo 8. Una compensación que falla de forma permanente debería llegar a una persona mediante un parking lot, con el estado de la saga registrado, y no descartarse.

Las compensaciones deben ser idempotentes y tolerar llegar antes de tiempo. Lo primero es territorio del Capítulo 7: un `ReleaseStock` duplicado no debe liberar dos veces. Lo segundo es más sutil. Supón que `Orders` envía `ReserveStock`, sufre un timeout y empieza a compensar enviando `ReleaseStock`. La reserva fue lenta, no se perdió, y la liberación puede llegar primero a `Inventory`. Si `Inventory` trata una liberación de una reserva desconocida como un no-op, el `ReserveStock` tardío tiene éxito después y la lámpara queda retenida para siempre. La forma segura es que `Inventory` registre la liberación contra el pedido, de modo que la reserva posterior la vea y se niegue. Algunos textos lo llaman tombstone. Sea cual sea el nombre, la compensación debe dejar una marca que la acción pueda leer.

Estado de la saga y timeouts

Una saga orquestada necesita estado durable, y este pertenece a la misma base de datos que el servicio que la ejecuta, de modo que los cambios de estado se sumen a la misma transacción local que las escrituras del outbox y del inbox.

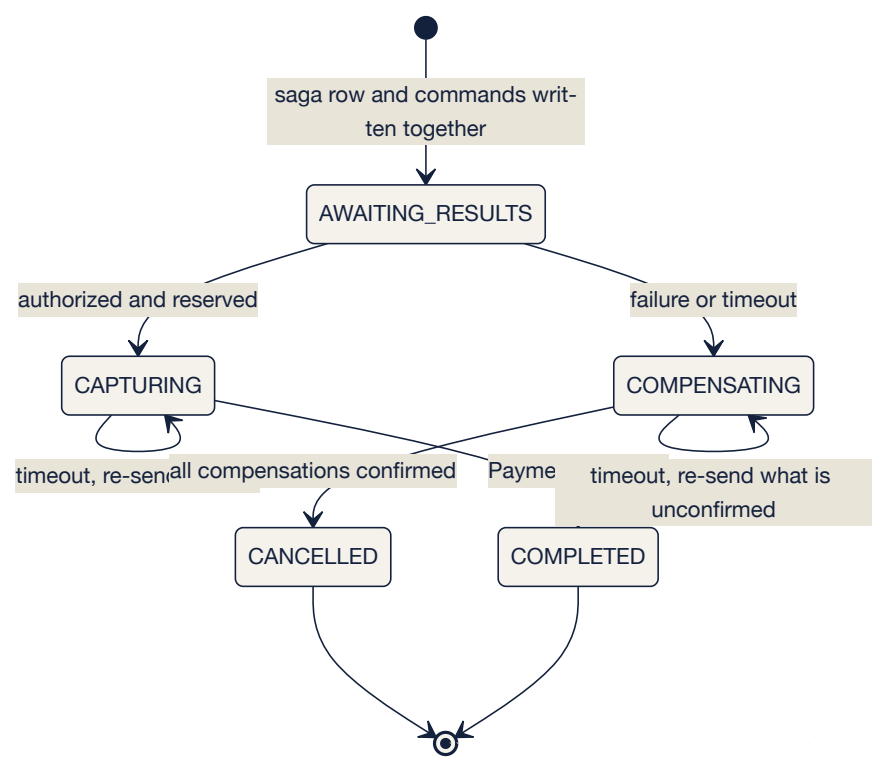
```
table saga_instance:
  saga_id      text primary key
  order_id     text unique not null
  state        text not null      // see below
  authorized   boolean not null
  reserved     boolean not null
  stock_released boolean not null // compensation confirmed
  auth_voided  boolean not null  // compensation confirmed
  rounds       int not null      // deadlines spent in this state
  deadline_at  timestamp not null
  version      int not null
  updated_at   timestamp not null
```

La columna `state` es un conjunto pequeño de posiciones con nombre: `AWAITING_RESULTS`, `CAPTURING`, `COMPLETED`, `COMPENSATING`, `CANCELLED`. La fila de la saga y los primeros comandos se escriben en una sola transacción, así que no existe una posición “iniciada” aparte que alguien pudiera observar. Los dos booleanos `authorized` y `reserved` registran qué resul-

tados han llegado, que es como funciona el join: la captura se solicita cuando ambos son verdaderos y el estado es `AWAITING_RESULTS`. Las dos banderas `stock_released` y `auth_voided` registran qué compensaciones se han confirmado, de modo que “todas las compensaciones confirmadas” es algo que la fila puede responder. Una compensación que no hace falta empieza ya confirmada: tras `PaymentFailed` no hay ninguna autorización que anular, así que `auth_voided` es verdadero al entrar en `COMPENSATING`.

Una restricción `unique` sobre `order_id` impide que corran dos sagas para un mismo pedido. La columna `version` protege contra que dos handlers actualicen la misma fila a la vez, y los handlers que siguen dependen de ella.

Una máquina de estados dibujada una vez vale más que cualquier descripción en prosa. El siguiente diagrama muestra las transiciones legales; fíjate en las salidas que no llevan a `COMPLETED` y en los dos estados que se repiten a sí mismos con su propio timeout.



Observa que `CAPTURING` no tiene salida hacia `COMPENSATING`. Eso es el pivot expresado en la máquina de estados. Una captura cuya respuesta nunca llegó se reenvía, y si el proveedor informa de un rechazo permanente, se maneja como una vía de negocio explícita y aparte, con una persona en ella.

Los bucles importan tanto como las salidas. Una saga en `CAPTURING` o `COMPENSATING` está esperando una respuesta, y una respuesta puede perderse como cualquier otro mensaje. Si solo `AWAITING_RESULTS` tuviera un deadline, una saga atascada en cualquiera de los otros dos se quedaría ahí sin nada que la despierte. Por eso todo estado no terminal tiene un deadline, y el deadline se reinicia en cada transición.

Timeouts

Cada espera en una saga necesita un deadline, porque una respuesta que nunca llega también es un fallo, y el que más probablemente se olvida. El Capítulo 3 describió cómo un timeout esconde tres posibilidades: la acción falló, la acción tuvo éxito y la respuesta se perdió, o la acción aún no ha terminado. La saga no puede saber cuál es, así que necesita una política para cada espera, escrita.

Un timeout también es un mensaje. Este capítulo introduce un evento interno para él, `SagaTimedOut`, emitido por el temporizador del propio orquestador. La implementación habitual es un proceso temporizador que selecciona las sagas cuyo `deadline_at` ya pasó y entrega a cada una un evento `SagaTimedOut` por el mismo handler que cualquier otra entrada. Así el timeout sigue el mismo camino que todo lo demás, con el mismo inbox, las mismas comprobaciones de estado y el mismo outbox.

```

function handleSagaTimedOut(message):
  if inboxContains(message.event_id):
    // fast path only; the unique constraint is the safety net
    ack(message)
    return
  try:
    transaction:
      inboxInsert(message.event_id)
      saga = loadSaga(message.payload.saga_id)
      if saga.state == "AWAITING_RESULTS":
        saga.state = "COMPENSATING"
        saga.rounds = 0
        sendOutstandingCompensations(saga, message.event_id)
      else if saga.state == "COMPENSATING":
        saga.rounds = saga.rounds + 1
        sendOutstandingCompensations(saga, message.event_id)
      else if saga.state == "CAPTURING":
        // same command_id as the first attempt: a repeat, not
new work
        saga.rounds = saga.rounds + 1
        outboxInsert(CapturePayment, saga.order_id,
                     command_id: commandIdFor(saga.saga_id,
"capture"))
        // any other state (COMPLETED, CANCELLED): a stale
timeout, recorded and dropped

        // sets a clock for a non-terminal state; escalates past
MAX_ROUNDS
        armNextDeadline(saga)
        saveSaga(saga, expected_version: saga.version)
        ack(message)
    on UniqueViolation:
      // a concurrent duplicate committed first; the work is done
      ack(message)
    on VersionConflict:
      // another handler changed the row; nothing above committed
      nack(message)

```

La rama `AWAITING_RESULTS` envía comandos de compensación sin preguntar qué paso se completó. Es deliberado. En el momento del timeout el orquestador no sabe si un `StockReserved` tardío viene en camino, así que pide los deshacer que siguen pendientes y confía en que los participantes traten “nada que deshacer” como éxito. El Capítulo 7 es la razón de que sea seguro.

`armNextDeadline` hace dos cosas dentro de la misma transacción. Fija `deadline_at` en `now() + timeoutFor(saga.state)`, de modo que ningún estado no terminal se quede sin reloj. Y cuando `saga.rounds` supera `MAX_ROUNDS`, digamos cinco, la saga deja de ser un problema de tiempos y

pasa a ser un problema humano: lanza una alerta y registra el estado de la saga en el parking lot del Capítulo 8, y las rondas siguen corriendo. Las sagas terminales no reciben un deadline nuevo. Como el deadline, el cambio de estado y los comandos se confirman juntos, una caída no puede dejar una saga que avanzó pero ya no está vigilada.

```
function sendOutstandingCompensations(saga, causing_event_id):
    // asked for again each round until its confirmation has arrived
    if not saga.stock_released:
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "release",
        causing_event_id))
    if not saga.auth_voided:
        outboxInsert(VoidAuthorization, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "void",
        causing_event_id))
```

Una saga en **CAPTURING** nunca da marcha atrás. El pivot ya pasó, así que los únicos movimientos son reenviar la captura con la misma clave y preguntarle a una persona.

Elegir la duración de un timeout no es algo que yo pueda hacer por ti. Depende de cuánto tarda el camino sano más lento, cuánto esperará una persona compradora y cuánto le cuesta a un vendedor retener stock. Un deadline fijado en un valor muy corto convierte lo lento en fallido e inunda el sistema de compensaciones. Uno muy largo retiene reservas mientras las personas compradoras ven una página de agotado.

Las llegadas tardías también necesitan una regla. Si **PaymentAuthorized** llega cuando la saga ya está **CANCELLED**, el orquestador debe emitir **VoidAuthorization** otra vez, porque alguien tiene fondos retenidos en una tarjeta por un pedido que no existe. El evento tardío demuestra que existe una autorización. La anulación enviada en el timeout puede seguir en vuelo, puede haberse procesado justo después de que la autorización se completara, o la llamada al proveedor puede haberse ejecutado antes de que Payments comprobara la marca que deja la anulación. Enviar la anulación de nuevo resuelve los tres casos.

Ese reenvío solo funciona si no se confunde con una repetición. Aquí se encuentran dos reglas sobre identidad. Un comando cuya repetición no debe causar trabajo nuevo, como **CapturePayment**, conserva un solo **command_id** por saga y paso, de modo que un retry se reconoce y se res-

ponde con el primer resultado. Una compensación que legítimamente puede emitirse otra vez toma su `command_id` del mensaje que la disparó: saga, paso y el `event_id` causante. La anulación reemitida es entonces un comando nuevo. También tiene que ser idempotente sobre el estado objetivo, no solo sobre el comando, así que Payments interpreta `VoidAuthorization` como “no puede quedar ninguna autorización viva para el pedido 81724” y comprueba eso, en lugar de repetir la respuesta en cache “nada que anular” del primer intento. Una máquina de estados que ignora los eventos que llegan en el estado equivocado pierde estos casos. Manéjalos de forma explícita: decide, por estado y por evento, si aplicar, ignorar o compensar.

Bloqueos semánticos

Como las sagas renuncian al aislamiento, otros flujos pueden leer y modificar un pedido a mitad de la saga. Una persona compradora puede editar la dirección de entrega mientras la captura está pendiente, un vendedor puede cancelar, un agente de soporte puede reembolsar. Los números de versión del Capítulo 9 ayudan con los eventos obsoletos, pero no impiden que dos flujos hagan cambios de negocio incompatibles.

Un **bloqueo semántico** es una bandera en los datos que dice “una saga está trabajando en esto”. No es un bloqueo de base de datos. Es un valor de estado que el resto del código está escrito para respetar. En Parcelly, el estado de pedido `PENDING` es uno: mientras se mantiene, el pedido no se puede editar, no se puede enviar y no se puede reembolsar, y la página del pedido muestra “procesando”. Del mismo modo, la reserva en Inventory es un bloqueo sobre la lámpara, por eso la segunda persona compradora vio “agotado”.

El bloqueo semántico tiene que liberarse en todos los caminos: al completarse, al compensar y al vencer el timeout. Una saga que se estanca sin liberar su bloqueo deja el pedido congelado, un fallo silencioso que aflora como tickets de soporte. El timeout de arriba es la protección, y la conciliación del Capítulo 13 es la segunda línea.

Vale la pena conocer dos contramedidas más. Una es hacer las actualizaciones conmutativas cuando sea posible, de modo que el orden no importe. La otra es diseñar una vista conservadora: por ejemplo, mostrar al ven-

dedor un conteo de stock que ya reste las reservas pendientes, para que nadie prometa lo que está a punto de quedar retenido. Ninguna elimina la necesidad de decidir, para cada estado, qué pueden hacer otros flujos mientras una saga está en curso. Considero esa tabla, estado por operación permitida, parte del diseño de cualquier saga.

Outbox e idempotencia dentro de una saga

El handler de la saga es donde se encuentran los capítulos anteriores. Consume un mensaje, cambia el estado y emite el siguiente comando o evento. Hecho con descuido, es un dual write por partida doble.

La forma que funciona combina los capítulos 4 y 7. Cada handler ejecuta una transacción local que hace tres cosas: registra el mensaje en el inbox, actualiza la fila de la saga y escribe los comandos o eventos salientes en el transactional outbox. El relay publica desde ahí.

```

function handleStockReserved(message):
  if inboxContains(message.event_id):
    ack(message)
    return
  try:
    transaction:
      // unique event_id: a concurrent duplicate fails here
      inboxInsert(message.event_id)
      saga = loadSagaByOrder(message.payload.order_id)
      if saga.state in ["CANCELLED", "COMPENSATING"]:
        // late success after we gave up: undo it
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id,
"release", message.event_id))
      else if saga.state == "AWAITING_RESULTS":
        saga.reserved = true
        if saga.authorized:
          saga.state = "CAPTURING"
          outboxInsert(CapturePayment, saga.order_id,
                       command_id:
commandIdFor(saga.saga_id, "capture"))
        // a conflict rolls back the whole transaction, inbox
row included
        saveSaga(saga, expected_version: saga.version)
        // any other state: irrelevant, recorded and dropped
        // commit point: inbox, saga state and outbox rows are one
unit
        ack(message)
      on UniqueViolation:
        ack(message)
      on VersionConflict:
        // the redelivery will see the other handler's flag
        nack(message)

```

Esta es la regla que introdujo el Capítulo 7 para los consumidores que publican: la fila del inbox, el cambio de estado y la inserción en el outbox se confirman en una sola transacción. Si el proceso se cae antes del commit, no pasó nada y el broker reentrega. Si se cae después del commit y antes del ack, la reentrega encuentra la fila del inbox, se confirma con ack y el trabajo se omite; un duplicado concurrente pierde en la restricción única y se confirma igual. Un mensaje obsoleto o irrelevante también se registra y se confirma, de modo que nunca vuelve. La saga avanza una vez por mensaje, sin importar cuántas veces se entregue el mensaje.

La comprobación de `expected_version` es lo que hace correcto el join. `PaymentAuthorized` y `StockReserved` pueden ser procesados al mismo tiempo por dos workers, y cada uno carga una fila con ambas banderas en falso. Sin la comprobación, la segunda escritura pisa a la primera, y la

saga espera en `AWAITING_RESULTS` hasta que su timeout compensa un pedido sano. Con ella, una transacción pierde, hace rollback junto con su fila del inbox, y su reentrega carga una fila que ya contiene la otra bandera. Cargar la fila con un bloqueo de fila logra lo mismo.

Los comandos que envía la saga son a su vez mensajes que se entregarán `at-least-once`, así que el participante necesita la misma protección, y el `command_id` es de donde viene. Para una acción que no debe repetirse, devuélvalo de la saga y del nombre del paso. Así Payments reconoce un `CapturePayment` reintentado y responde con el primer resultado. Si el orquestador regenera un `command_id` aleatorio en cada intento, cada retry parece trabajo nuevo y a la persona compradora se le puede cobrar dos veces. Las compensaciones que pueden reemitirse siguen la otra regla de la sección sobre llegadas tardías: el id viene del mensaje que las dispara, y el participante es idempotente sobre el estado objetivo. El `causation_id` y el `correlation_id` que viajan en el envelope también hacen trazable cada saga: el `correlation_id` es el flujo, y cada `causation_id` es un paso dentro de él.

Advertencia: Una compensación que se envía sin clave de idempotencia es tan peligrosa como un pago que se envía sin ella. Un `RefundPayment` reintentado sin una clave estable puede reembolsar dos veces a quien compró, y el segundo reembolso es dinero que no vuelve.

Dónde encajan los motores de flujos de trabajo

Todo lo anterior se puede construir con una tabla, un temporizador y unos cuantos handlers, y para un flujo del tamaño del pedido de Parcelly yo empezaría por ahí. En algún momento la maquinaria pasa a ser la mayor parte del código: temporizadores durables, retries por paso, versionado de los flujos de trabajo en ejecución, un historial visual de cada instancia. Ese es el problema que abordan los motores de flujos de trabajo.

La clase de herramienta es amplia. Algunas son bibliotecas o servicios que persisten el progreso del flujo de trabajo y lo reanudan tras una caída, con código que parece un programa secuencial. Otras interpretan una definición declarativa de estados y transiciones. Los proveedores de nube ofre-

cen versiones gestionadas, y existen varios proyectos de código abierto. No voy a comparar productos, porque el panorama cambia más rápido que un libro y la elección depende de tu lenguaje, de tus habilidades operativas y de tu tolerancia a operar otro componente crítico.

Lo que le pediría a cualquier motor es una lista corta. ¿Dónde vive el estado del flujo de trabajo y puedes inspeccionarlo? ¿Cómo hace atómicos el efecto secundario de un paso y su propio progreso, o te deja el dual write? ¿Qué pasa con un flujo de trabajo que ya está en ejecución cuando haces un deploy de una definición modificada? ¿Cómo se expresan las compensaciones y los timeouts? Un motor que guarda el progreso pero ejecuta tu paso en una transacción aparte sigue necesitando pasos idempotentes; el motor desplaza el problema, no lo elimina.

El resumen honesto es que un motor te ahorra construir y operar la máquina de estados, y te cuesta una dependencia, un modelo que aprender y una pieza nueva que puede caerse. Para una o dos sagas pequeñas, una tabla es más barata. Para muchos flujos de trabajo de larga duración con pasos humanos y esperas largas, un motor se paga solo.

Trade-offs y costos

Una saga cambia atomicidad por disponibilidad e independencia, y el precio se paga en varias monedas.

Complejidad. Cada saga multiplica los estados. Un flujo de tres pasos, cada uno con una acción y una compensación, más timeouts y llegadas tardías, tiene muchos más estados alcanzables de lo que sugiere el happy path, y cada uno necesita una decisión y una prueba. El Capítulo 15 vuelve sobre cómo probarlos.

Estados intermedios visibles. No hay aislamiento. Quienes compran, quienes venden, las herramientas de soporte y otros servicios ven un pedido a medio hacer. Lo aboradas en el diseño con bloqueos semánticos y estados claros, y aceptas que algunos usuarios lo verán de todos modos.

Consistencia eventual, a propósito. Entre un fallo y la última compensación, el sistema es inconsistente, durante segundos o, con un mal timeout, horas. Eso solo es aceptable si el negocio lo ha acordado.

Responsabilidad operativa. Alguien debe vigilar las sagas atascadas en un estado más allá de su deadline, las compensaciones que no pueden completarse y la antigüedad de la saga en ejecución más vieja. Sin un responsable y una alarma, las sagas fallan en silencio.

La compensación no es gratis. Un reembolso tiene comisiones y demora, un envío cancelado puede acarrear una tarifa, y un mensaje a quien compró que diga “tu pedido fue cancelado” es en sí mismo un evento visible. Diseña el flujo para que el fallo común sea barato de deshacer.

Cuándo no usar una saga: cuando los pasos pueden compartir una transacción local; cuando el fallo es raro y una corrección manual sale más barata que construir compensaciones, lo cual es una elección legítima para un proceso interno de bajo volumen; cuando el “flujo de trabajo” son en realidad reacciones independientes que no necesitan un deshacer coordinado, como enviar un correo o actualizar un índice de búsqueda. Y cuando no puedes definir la compensación. Si un paso no se puede deshacer y no se puede posponer, la pregunta honesta de diseño es si la frontera entre los servicios está en el lugar equivocado.

Errores comunes

- **Construir una saga donde bastaría una transacción.** Tienta porque los servicios ya existen y el diagrama se ve moderno. Si dos pasos comparten responsable y base de datos, fúndelos en una transacción local.
- **Diseñar el happy path y dejar las compensaciones para después.** Ese después llega en forma de incidente. Escribe la compensación junto al paso y nombra el evento que emite.
- **Llamar rollback o reembolso a cualquier deshacer.** Antes de la captura la compensación es una liberación o una anulación; un reembolso existe solo después de la captura. Usar la palabra equivocada oculta el estado real en el libro mayor y en el extracto de quien compró.
- **Dejar que una saga espere para siempre.** Un deadline ausente es invisible hasta que llama un cliente. Dale a cada espera en cada estado no terminal un timeout y un resultado con nombre, incluidas las esperas de la respuesta de la captura y de las confirmaciones de compensación.

- **Ignorar los eventos que llegan en el estado equivocado.** Un `PaymentAuthorized` tardío para un pedido cancelado es una tarjeta retenida. Decide por estado y evento si aplicar, ignorar o compensar.
- **Compensaciones no idempotentes.** Una liberación o un reembolso reintentado que actúa dos veces provoca un segundo incidente mientras arregla el primero. Dale un `command_id` estable a las acciones que no deben repetirse, deriva el id de una compensación reemitible a partir de su causa, y haz que los participantes sean idempotentes sobre el estado objetivo. Un id fijo para una anulación reenviada repite “nada que anular” y deja la autorización retenida.
- **Meter todo el negocio en el orquestador.** El orquestador debería decidir el orden y el resultado; los participantes deberían ser responsables de las reglas. Si el orquestador valida los niveles de stock, está haciendo el trabajo de Inventory.
- **Poner pasos no compensables al principio.** Enviar el traspaso al transportista antes de la captura significa que un fallo te deja con un envío y sin dinero. Ordena los pasos de modo que los difíciles de deshacer vayan al final.
- **Omitir la liberación del bloqueo.** Un bloqueo semántico fijado en `PENDING` y nunca limpiado en el timeout o en la compensación congela el pedido. Libéralo en cada camino de salida.

Ideas clave

- Una saga reemplaza una transacción distribuida por una secuencia de transacciones locales y compensaciones, y renuncia al aislamiento para lograrlo.
- Una compensación es una acción de negocio nueva, como `StockReleased` o `PaymentVoided`, no un rollback, y antes de la captura nunca es un reembolso.
- La coreografía encaja con flujos cortos de pocas ramas, la orquestación encaja con flujos con joins, timeouts y muchas ramas, y a menudo se combinan, con el orquestador dentro del servicio responsable del proceso.
- Ordena los pasos según lo difícil que sea deshacerlos, y trata el primer paso que no se puede compensar, como la captura, como un pivot a

partir del cual la saga solo avanza.

- Cada espera en cada estado no terminal necesita un deadline, cada estado necesita una regla para eventos tardíos y duplicados, y cada compensación debe ser idempotente y tolerar llegar antes que la acción que deshace.
- Dentro de un handler, una sola transacción local debería combinar el inbox, el estado de la saga y el outbox, protegida por una comprobación de versión para que handlers concurrentes no se pisen, y con un `command_id` fijo para una acción que no debe repetirse, como la captura, pero derivado del mensaje que la dispara para una compensación que puede reemitirse.
- Un motor de flujos de trabajo puede ser responsable de la máquina de estados, los temporizadores y el historial, pero no elimina la necesidad de pasos idempotentes ni de una responsabilidad clara sobre los flujos de trabajo atascados.

Una saga supone que los eventos describen lo que ocurrió de una forma que todos los participantes pueden leer y en la que pueden confiar. Que lo hagan depende de lo que lleva cada evento, de cómo se llama y de quién es su responsable, que es el tema del Capítulo 11.

Capítulo 11 — Diseño de eventos

Imagina que el equipo de Shipping de Parcelly quiere programar los envíos más rápido. Hoy recibe `OrderConfirmed`, que lleva un `order_id` y poco más, y luego llama a Orders por HTTP para obtener la dirección de entrega y las líneas del pedido. Orders anda lento los lunes por la mañana, y Shipping se ralentiza con él. Alguien del liderazgo técnico propone la solución obvia: poner el pedido entero en el evento, para que Shipping nunca necesite preguntar.

En una semana el evento ha crecido. Orders añade el correo electrónico de la persona compradora, la dirección completa, la cuenta de pago del vendedor y un campo llamado `internal_status` que usa su propia pantalla de administración. Notifications empieza a leer el correo del evento en lugar de su propia tabla. El equipo de Ledger nota que la cuenta de pago ahora viaja por el broker y reposa en un topic con un período de retención largo. Entonces Orders renombra `internal_status` durante un refactor, y tres consumidores fallan la misma tarde.

Nadie en esta historia cometió un error de confiabilidad en el sentido de los capítulos anteriores. El outbox funcionó, los consumidores eran idempotentes, los retries eran sensatos. El daño vino del diseño del propio mensaje: qué dice, qué lleva y quién puede cambiarlo. Este capítulo pregunta cómo diseñar un evento para que siga siendo útil, seguro y modificable durante años después de que lo lea el primer consumidor.

El problema y por qué es difícil

El Capítulo 1 señaló que los servicios de un sistema orientado a eventos están acoplados a través del contrato del evento: su nombre, sus campos y su significado. Ese acoplamiento es invisible en el momento en que escribes el evento, porque en ese momento hay un productor y un consumidor, y ambos los escriben personas que se sientan cerca. El contrato parece un detalle interno.

Deja de serlo en el momento en que se suscribe un segundo consumidor, y el productor suele enterarse tarde o nunca. En un broker basado en log, un nuevo grupo de consumidores puede empezar a leer el topic sin avisar a nadie. En un broker basado en colas, alguien enlaza una cola nueva al mismo exchange o a la misma suscripción al topic. Desde ese punto, cada campo del payload es una promesa hecha a personas a las que quizá nunca conozcas.

Tres cosas lo hacen difícil en la práctica.

- **Es fácil añadir a los eventos y difícil quitarles.** Añadir un campo cuesta una línea de código. Quitarlo exige encontrar a todos los consumidores, incluidos los de repositorios que no lees y los que hacen re-play del historial antiguo.
- **La presión del diseño apunta en la dirección equivocada.** El equipo productor quiere entregar, así que emite lo que tiene a mano. Los equipos consumidores quieren evitar llamadas extra, así que piden más campos. A nadie le pagan por decir que no.
- **Los errores son duraderos.** Un evento escrito hoy puede quedarse meses en un log de eventos y reproducirse en un consumidor que aún no existe. Un evento mal nombrado o un campo personal filtrado permanece almacenado mucho después de que desaparezca el código que lo produjo.

Así que el diseño de eventos no es decoración. Es la parte del sistema donde unas pocas horas de reflexión deciden cuán caro será cada cambio posterior.

Eventos, comandos y documentos

La primera decisión de diseño es qué clase de mensaje envías. Hay tres clases comunes, y tienen reglas distintas.

Un **evento** enuncia un hecho que ya ocurrió: `PaymentCaptured`. Al productor no le importa quién escuche, y no espera nada a cambio. Los eventos se nombran en pasado, porque informan de algo que ya sucedió; un evento posterior puede contrarrestarlo, pero no puede deshacerlo. Un consumidor puede ignorar un evento, y esa es una respuesta legítima.

Un **comando** le pide a un servicio concreto que haga algo: `CapturePayment`. Tiene un único destinatario previsto y la expectativa de que el destinatario actúe. Un comando puede ser rechazado, y a quien lo envía normalmente le importa el resultado. Los comandos son imperativos, llevan un `command_id` en lugar de `event_id` y nunca se llaman eventos. Una saga orquestada, que cubre el Capítulo 10, conduce a sus participantes con comandos de esta clase.

Un **mensaje de documento** lleva un bloque de datos sin decir que algo ocurrió ni que alguien deba actuar, como la instantánea nocturna del catálogo de un vendedor. El receptor decide qué significa. Los mensajes de documento son legítimos, pero son la clase que más a menudo se disfraza de evento, y el disfraz causa confusión.

Clase	Dice	Destinatario	Nombre	¿Puede el receptor rechazarlo?
Evento	Ocurrió un hecho	Cualquiera suscrito	Pasado: <code>OrderPlaced</code>	Sí, ignorándolo
Comando	Haz esto	Un servicio con nombre	Imperativo: <code>CapturePayment</code>	Sí, respondiendo con un fallo
Documento	Aquí hay unos datos	Quien lo lea	Sustantivo: <code>CatalogSnapshot</code>	No aplica

El fallo común es el comando con ropa de evento. Supón que Orders publica `SendConfirmationEmail`, un nombre imperativo, y lo llama evento porque viaja por el topic. Orders ahora depende de que Notifications se comporte de cierta manera, pero no tiene forma de oír un rechazo. Si Notifications está caído, Orders no lo sabe. Si se suscribe un segundo servicio, se encuentra con una instrucción que no va dirigida a él.

Aplicaría una prueba rápida. Pregunta si el productor seguiría siendo correcto aunque nadie consumiera el mensaje. Para `OrderPlaced` la respuesta es sí: el pedido se hizo de todos modos. Para `SendConfirmationEmail` la respuesta es no, porque el mensaje solo tiene sentido si alguien actúa. Esa segunda clase pertenece a un canal donde el destinatario tiene nombre y el fallo vuelve a quien envía.

Parte de esto es también una cuestión de dirección. Los eventos apuntan desde el responsable de un hecho hacia cualquiera que esté interesado. Los comandos apuntan desde un coordinador hacia un ejecutor. Cuando el

nombre de un evento empieza a sonar como una instrucción a un servicio particular, la dirección se ha invertido.

Cómo es un comando

Las dos clases también fallan de forma distinta, y el envelope tiene que reflejarlo. Un evento perdido es silencioso por diseño: el productor nunca esperó que nadie en particular actuara, así que nadie nota la ausencia. Un comando perdido o tardío es diferente. Quien lo envía está esperando un resultado, y lo único que revela la pérdida es su propio deadline, por eso la saga del Capítulo 10 pone un deadline en cada estado no terminal. Un comando también puede llegar después de que quien lo envió se rindió y compensó, así que el receptor necesita una manera de rechazarlo.

Por tanto, un comando lleva un envelope más pequeño y distinto del de un evento:

```
{
  command_id:      "cmd-capture-saga-7731",          // stable across
retries of the same intent
  command_type:    "CapturePayment",
  target:          "payments",                      // one named
recipient
  reply_to:        "orders.saga-replies",           // where the
outcome goes
  expects:         ["PaymentCaptured", "PaymentCaptureFailed"],
  correlation_id:  "order-81724",
  causation_id:    "e3",
  expires_at:      "2026-10-06T14:05:00Z",          // optional; judged
on the receiver's clock
  payload:         { order_id: "81724", amount_minor: 4990, currency:
"EUR" }
}
```

`command_id` reemplaza a `event_id`, y el Capítulo 10 explica cómo se deriva: de la saga y el paso para una acción que no debe repetirse, y con el `event_id` causante para una compensación que puede emitirse otra vez. `reply_to` y `expects` le dan al resultado un camino de vuelta, de modo que un rechazo es algo que quien envía puede oír. `expires_at` permite que un receptor descarte un comando que llega después de que quien lo envió dejó de esperar. Es opcional porque tiene un costo: se compara con el reloj del receptor, y el Capítulo 3 dio las razones por las que no se puede confiar en ese reloj al segundo. Fíjalo con margen y trátalo como protección contra comandos de hace horas, no contra los de hace segundos.

Eventos de dominio y eventos de integración

Dentro de un servicio, el código a menudo lanza eventos sobre sus propios agregados: `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. Estos **eventos de dominio** describen el modelo interno y cambian siempre que cambia el modelo. Son útiles dentro del servicio, por ejemplo para actualizar un modelo de lectura o disparar una regla en la misma transacción.

Un **evento de integración** es la versión que cruza la frontera del servicio. Se elige deliberadamente, se nombra por lo que otros servicios necesitan saber y se mantiene estable. `OrderPlaced` es un evento de integración. No es un volcado de lo que fuera que el agregado del pedido tuviera en ese momento.

La tentación es publicar los eventos de dominio directamente en el broker, porque la maquinaria ya existe. En mi opinión, esta es la manera más común de filtrar estructura interna. Piensa en lo que ocurre cuando el equipo de Orders divide un agregado en dos, o fusiona dos tipos de línea. Si los eventos de dominio fueran el contrato público, todos los consumidores se rompen, aunque nada del negocio haya cambiado.

La solución es un paso de traducción en el servicio productor. Los eventos de dominio siguen siendo libres de cambiar. Una pequeña función de mapeo decide cuáles de ellos se convierten en eventos de integración, con qué nombres y con qué campos. Esa función es donde decides cuál es el contrato público, y el transactional outbox del Capítulo 4 es un lugar natural para llamarla: el evento de integración es lo que se escribe en la fila del outbox.

```
function toIntegrationEvent(domainEvent, context):
    // the mapping is the public contract; internals stay behind it
    if domainEvent.kind == "OrderSubmitted":
        return {
            event_id:      newEventId(),
            event_type:     "OrderPlaced",
            schema_version: 1,
            occurred_at:    domainEvent.occurred_at,
            producer:       "orders",
            correlation_id:  context.correlation_id,
            causation_id:    context.causation_id,
            partition_key:   domainEvent.order_id,
            payload: {
                order_id:    domainEvent.order_id,
                buyer_id:    domainEvent.buyer_id,
                line_count:   domainEvent.lines.length,
                total_minor:  domainEvent.total_minor,
                currency:     domainEvent.currency
            }
        }
    // most domain events never leave the service
    return nothing
```

Observa que el nombre interno `OrderSubmitted` y el nombre público `OrderPlaced` difieren. Eso está permitido, y es sano. El nombre público es el que aprenden otros equipos. El nombre interno puede seguir al código.

Thin events y fat events

¿Cuánto debería llevar el payload? Vale la pena nombrar los dos extremos.

Un **thin event**, a veces llamado notificación, lleva un identificador y poco más: “se hizo el pedido 81724”. Un consumidor que necesita detalles llama de vuelta al productor. Un **fat event** lleva suficiente estado para que los consumidores actúen sin llamar de vuelta. Este estilo suele llamarse *event-carried state transfer*, porque el evento traslada un fragmento de estado de un servicio a otro.

Cada lado tiene un modo de fallo que los capítulos anteriores ayudan a explicar.

Los thin events traen de vuelta el acoplamiento síncrono que los eventos debían eliminar. Shipping solo puede avanzar cuando Orders está accesible en ese momento, que es exactamente el problema de los lunes por la mañana del inicio. También crean una condición de carrera: para cuando

el consumidor llama de vuelta, los datos pueden haber cambiado, y el consumidor lee un estado más nuevo que el que disparó el evento. Si la llamada de vuelta falla, el consumidor cae en el mundo de retries del Capítulo 6, contra una dependencia que puede estar lenta por mucho tiempo.

Los fat events cuestan en otros lugares. El payload crece, y también el número de campos que has prometido mantener. Los consumidores empiezan a tratar el evento como una réplica de los datos del productor, y la réplica envejece. Un consumidor que guardó en cache la dirección de `OrderConfirmed` el lunes puede usarla el viernes, después de que la persona compradora la corrigió, a menos que lea también el evento posterior. Y todo lo que va en el payload se copia a cada consumidor y a cada segmento de log retenido, lo que sube la apuesta para los campos sensibles.

	Thin	Fat
Payload	Identificador y tipo de evento	Identificador más el estado que necesitan los consumidores
El consumidor necesita que el productor esté activo	Sí, para obtener los detalles	No
Riesgo de obsolescencia	Bajo al leer, pero compite con cambios posteriores	El consumidor guarda una instantánea que envejece
Tamaño del contrato	Pequeño	Grande
Exposición de privacidad	Baja	Mayor
Replay	Necesita los datos actuales del productor, que pueden diferir de los de entonces	Autocontenido, reproduce lo que se sabía

Tiendo a preferir un camino intermedio: **lleva lo que el consumidor necesita para decidir y actuar, y nada de lo que solo quizá quiera**. Para `OrderConfirmed`, Shipping necesita el id del pedido, las líneas que debe empacar y una dirección de entrega, así que eso pertenece al evento. No necesita el método de pago de la persona compradora, los datos de pago del vendedor ni las reglas de descuento que produjeron el total. Una buena manera de hallar el límite es preguntarle a cada equipo consumidor qué haría con cada campo. Un campo sin consumidor nombrado y sin uso declarado es candidato a eliminarse.

Un segundo detalle importa aquí. Si el evento lleva estado, indica en el payload a qué momento pertenece ese estado. Con la versión del agregado (Capítulo 4), que el Capítulo 9 llama `order_version` para los pedidos de Parcelly, un consumidor puede saber si la instantánea que tiene es más nueva o más antigua que la de un segundo evento; `occurred_at` es solo una pista aproximada, por las razones que se dan más abajo. El Capítulo 9 cubre cómo detectar la llegada desordenada con tales números, así que este capítulo solo anota que un fat event sin versión invita a la trampa de last-writer-wins descrita allí. No confundas este número con `schema_version` del envelope: la versión del agregado cuenta los cambios de un pedido y avanza con cada cambio, mientras que `schema_version` nombra la versión incompatible de la forma del payload y avanza rara vez (Capítulo 12).

La elección no tiene por qué ser global. Un equipo puede empezar con thin para los eventos poco usados y añadir campos cuando un consumidor real los pida. Empezar thin y crecer es más barato que empezar fat y encoger, porque encoger implica un cambio incompatible, que el Capítulo 12 trata con detalle.

Nombrar eventos

Los nombres de los eventos de Parcelly que se usan a lo largo del libro siguen un pequeño conjunto de reglas, y esta sección las enuncia con sus razones.

- **Pasado**, porque el evento es un hecho: `PaymentCaptured`, no `CapturePayment`, que es el comando.
- **Sujeto primero, PascalCase**: primero el agregado, luego lo que ocurrió. `ShipmentDispatched`, no `DispatchedShipment`. Ordenados alfabéticamente, los eventos sobre un mismo agregado quedan agrupados en catálogos y dashboards.
- **Lenguaje de negocio, no lenguaje técnico**. `OrderConfirmed` dice algo que entiende un product owner. `OrderRowUpdated` dice algo sobre una tabla.
- **Sin tecnología y sin consumidor en el nombre**. `OrderPlacedKafkaEvent` filtra el broker. `OrderPlacedForShipping`

filtra un consumidor, y el siguiente consumidor tendrá que preguntarse si el evento es para él.

- **Sin prefijo de servicio, salvo que dos servicios emitan hechos sobre el mismo sustantivo.** Supón que Orders y Shipping pueden informar ambos de un `DeliveryDelayed`: uno significa que el vendedor no ha entregado el paquete, el otro que el transportista va tarde. Los consumidores no pueden distinguirlos por el nombre, así que calificarlos se vuelve necesario (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). El ejemplo es ilustrativo; los servicios de Parcelly de este libro no emiten ambos. De lo contrario, el productor ya aparece en el campo `producer`.

La regla de nombres más importante es la del significado. `OrderUpdated` es un nombre que no pasa la prueba del hecho: dice que algo cambió y nada más. Un consumidor debe comparar el estado nuevo con el anterior para saber qué ocurrió, y no puede distinguir una dirección cambiada de un total cambiado. Prefiere nombres que enuncien el hecho de negocio: `OrderCancelled`, `ShipmentAddressCorrected`. Cuando no encuentras un nombre de negocio, a menudo es señal de que el evento describe un cambio de base de datos y no uno de negocio.

Otra regla es la estabilidad. Una vez que los consumidores dependen del nombre de un evento, renombrarlo es un cambio incompatible con el mismo costo que quitar un campo. Dedicar esos 10 minutos extra al nombre antes del primer lanzamiento.

El envelope

Todo evento de este libro lleva el mismo envelope: `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` y `payload`. El envelope es la parte del mensaje en la que las herramientas pueden confiar sin entender el negocio. Un relay, un handler de dead letters, un sistema de tracing y una herramienta de replay pueden trabajar con él sin analizar el `payload`.

Cada campo responde a una pregunta que hace algún componente:

Campo	La pregunta que responde	Quién la hace
<code>event_id</code>	¿He visto antes este evento exacto?	Inbox, deduplicación (Capítulo 7)
<code>event_type</code>	¿Cómo interpreto el payload?	Enrutamiento, deserialización
<code>schema_version</code>	¿Qué versión incompatible de la forma tiene el payload?	Consumidores, schema registry (Capítulo 12)
<code>occurred_at</code>	¿Cuándo ocurrió el hecho, según lo vio el productor?	Conciliación, analítica, depuración
<code>producer</code>	¿Qué servicio es responsable de este hecho?	Responsabilidad, soporte, enrutamiento de preguntas
<code>correlation_id</code>	¿A qué flujo de negocio pertenece?	Soporte, conciliación, búsqueda en logs
<code>causation_id</code>	¿Qué lo causó directamente?	Reconstrucción de la cadena
<code>partition_key</code>	¿De qué agregado trata este evento?	Broker, orden (Capítulo 9)
<code>payload</code>	¿Cuáles son los detalles?	Lógica de negocio

Hay dos puntos que requieren cuidado. `occurred_at` es el reloj del productor, y el Capítulo 3 explicó por qué no debes ordenar eventos por él. Te dice aproximadamente cuándo ocurrió el hecho, y no sirve como secuencia. La `partition_key` es el identificador del agregado, y su elección fue el tema del Capítulo 9.

`correlation_id` no es un trace id. El contexto de trace (W3C Trace Context) viaja en headers junto al envelope, y el Capítulo 13 muestra cómo se captura en el outbox. Los traces se muestrean y expiran; `correlation_id` es la clave de negocio que sobrevive a ambas cosas, y es la que usa una persona de soporte para buscar.

El envelope entre lenguajes

Los servicios de Parcelly no están escritos todos en un mismo lenguaje. Orders y Payments corren en la JVM, Notifications es Node y los jobs de conciliación del equipo de Ledger son Python. La lista de campos de arriba dice qué contiene el envelope. No dice cómo se escribe cada campo en el cable, y ahí es donde un segundo lenguaje rompe un contrato que parecía correcto en el primero.

Campo	Tipo en el cable	Regla
<code>event_id</code> , <code>correlation_id</code> , <code>causation_id</code> , <code>partition_key</code>	string	Los identificadores son strings, aunque parezcan numéricos. Un id de 64 bits como número JSON lo redondea en silencio un consumidor JavaScript
<code>event_type</code>	string	Un nombre estable y neutral respecto al lenguaje, como <code>PaymentCaptured</code> , nunca el nombre de una clase o de un paquete, y nunca un header de tipo que un framework rellena por defecto
<code>schema_version</code>	entero	Pequeño, y un entero en todos los lenguajes
<code>occurred_at</code>	string	RFC 3339 en UTC con sufijo <code>Z</code> , con precisión fija de milisegundos, sin desplazamientos. Los parsers discrepan en todo lo demás
Dinero en <code>payload</code>	entero o string	Unidades menores enteras (<code>total_minor</code>) dentro de un rango documentado, o un string decimal. Nunca un float. Los números de JavaScript son exactos solo hasta $2^{53} - 1$, así que un campo que puede superarlo viaja como string
Campos de tipo enum	string	Trata el conjunto como abierto: un consumidor debe manejar un valor que no ha visto (Capítulo 12)

Dos reglas de ubicación completan el contrato. El envelope vive en el cuerpo del mensaje, y los headers del broker solo reflejan lo que el cuerpo ya dice, porque los headers son la parte que se pierde cuando un mensaje se copia, se envía a dead letters o se puentea a otro broker. Y el tipo de contenido se declara una vez, en el esquema, no se adivina mensaje por mensaje. Qué ocurre con los campos que un consumidor no conoce, y cómo difieren los formatos al decodificarlos, es una cuestión de compatibilidad y pertenece al Capítulo 12.

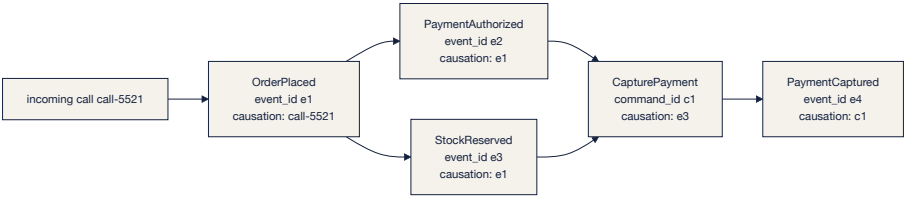
Correlación y causalidad

Los dos identificadores que más suelen confundir a quien lee son `correlation_id` y `causation_id`. Responden a preguntas diferentes.

El `correlation_id` lo comparten todos los eventos de un mismo flujo de negocio. Cuando una persona compradora hace el pedido 81724, todo lo que sigue lleva `correlation_id: "order-81724"`: la autorización, la re-

serva, la captura, el envío. Buscas por él cuando una persona compradora escribe y pregunta qué pasó.

El `causation_id` apunta al único mensaje que causó directamente este. Es el `event_id` de un evento, o el `command_id` de un comando. No se repite a lo largo del flujo. Cada salto lo fija con el identificador de su disparador inmediato, y seguir la cadena hacia atrás reconstruye el camino.



Mira los dos últimos pasos. `CapturePayment` lo disparan dos eventos, y `Orders` debe elegir uno para `causation_id`, aquí el que completó la condición. La ambigüedad es real, y yo dejaría escrita la regla para que todos los productores elijan de la misma manera. El evento `PaymentCaptured` nombra al comando, no al evento anterior, por eso el envelope admite ambas clases de identificadores en ese campo.

Cada evento del flujo comparte `correlation_id: "order-81724"`. El primer evento de un flujo fija el identificador de correlación, ya sea derivándolo del agregado o tomando uno de la solicitud entrante. Los eventos posteriores lo copian sin cambios.

Este libro mantiene el `order-81724` con alcance de agregado en todo momento, y esa es una elección con consecuencias. Un id con alcance de agregado es simple, pero todos los flujos de ese pedido se funden bajo él: un reembolso tres semanas después comparte el id con el checkout original, y no se pueden distinguir. La alternativa es un id con **alcance de flujo**, generado de nuevo para cada flujo (`checkout-81724-1`, `refund-81724-1`), con `order_id` buscable en el payload. Cuesta una cosa más que generar y propagar, y vale ese costo cuando un agregado puede iniciar un segundo flujo que debes distinguir del primero. El resto de este capítulo vale en ambos casos. Copiarlo es todo el trabajo, y un consumidor que olvida copiarlo rompe el rastro de todo lo que viene después.


```
function buildFollowUpEvent(trigger, eventType, payload):
    // trigger is the message being handled: an event or a command
    return {
        event_id:      newEventId(),
        event_type:    eventType,
        schema_version: currentVersionOf(eventType),
        occurred_at:   now(),
        producer:      thisServiceName,
        correlation_id: trigger.correlation_id, // copied, never
        regenerated
        causation_id:  identifierOf(trigger), // event_id or
        command_id
        partition_key: aggregateIdOf(payload), // this event's
        aggregate, not the trigger's
        payload:       payload
    }
```

La `partition_key` del disparador solo puede copiarse cuando ambos mensajes conciernen al mismo agregado. Un seguimiento sobre un pago o un envío que heredara la clave del pedido quedaría ordenado contra el stream equivocado (Capítulo 9). Parcellly usa el pedido como clave de casi todo, y por eso el error es difícil de detectar.

El costo de equivocarse se paga en el peor momento. Durante un incidente, la pregunta es “¿qué le pasó a este pedido a través de cinco servicios?”. Sin un `correlation_id` consistente, una persona ingeniera busca a mano en cinco logs. El Capítulo 13 muestra cómo el tracing y la conciliación usan estos campos, así que el trabajo de este capítulo es asegurar que estén presentes y sean correctos desde el primer evento.

Qué no pertenece a un evento

Un evento sale de la base de datos de su productor y entra en un sistema con reglas de acceso distintas, retención distinta y muchos lectores. Por eso la pregunta de diseño es también una pregunta sobre qué estás dispuesto a copiar en ese sistema.

Datos personales. Un evento con una dirección de correo, un número de teléfono o una dirección de entrega completa se almacena donde el broker almacene, posiblemente durante toda la ventana de retención, y lo lee cada consumidor. En un log de eventos, borrar más tarde los datos de una persona es incómodo, porque el log es append-only y los consumidores ya copiaron los campos. Algunos equipos responden con esquemas de cifrado

o topics aparte. Yo empezaría antes: pon un identificador en el evento, como `buyer_id`, y deja que un consumidor con una necesidad legítima obtenga los campos personales del servicio responsable en ese momento, sujeto a las reglas de acceso de ese servicio. Esto se inclina hacia thin events para los campos sensibles, y creo que el trato vale la pena. Si un fat event de verdad necesita una dirección, como ocurre con Shipping, entonces el evento debería llevarla deliberadamente, para esa audiencia, con el período de retención elegido sabiendo que está ahí.

Qué campos cuentan como datos personales y qué obligaciones se derivan depende de tu jurisdicción y de tus asesores legales. No conozco tus restricciones. El hábito de diseño es el mismo en todas partes: cada campo sensible de un evento debería tener una razón con nombre.

Secrets y credenciales. Los tokens, las claves y los números de cuentas de pago no pertenecen a un payload. Un mensaje se registra en logs, se copia a destinos de dead letters, se muestra en herramientas de administración y se adjunta a tickets. Supón que cada payload lo leerá una persona.

Detalle interno. Nombres de tablas, códigos de estado internos, feature flags, ids de base de datos que solo tienen sentido dentro de un servicio y valores cuyo significado depende del código del productor. `internal_status` del escenario inicial es el ejemplo. Un campo así es una promesa pública desde el momento en que alguien lo lee.

Instrucciones a consumidores específicos. Una bandera como `skip_email_if_gift` le dice a Notifications cómo comportarse. El evento debería enunciar el hecho, y el consumidor decide qué hacer con él.

Advertencia: Todo lo que entra en un log de eventos debe tratarse como permanente y legible por muchos. Un secret o un campo personal escrito una vez se copia a destinos de dead letters, backups y herramientas de replay, y eliminarlo en todas partes es mucho más difícil que dejarlo fuera.

Una lista de verificación corta ayuda al revisar un evento nuevo: ¿tiene cada campo un consumidor que lo necesita?, ¿es cada campo algo que el productor está dispuesto a mantener estable?, ¿podría el campo perjudicar a alguien si lo leyera una persona que no debería verlo?

Granularidad

¿Qué tan grande es un evento? Hay dos errores opuestos.

Demasiado fino. Publicar un evento `FieldChanged` por cada actualización de columna produce un stream de ruido. Los consumidores tienen que reconstruir el significado a partir de muchos mensajes, y el orden de esos mensajes empieza a importar de maneras sobre las que advirtió el Capítulo 9. Un cambio que era un solo hecho de negocio, la persona compradora corrigiendo una dirección, ahora llega como cuatro eventos que pueden verse en un estado a medio aplicar.

Demasiado grueso. Un evento `OrderChanged` que se dispara en cada modificación y lleva el pedido entero obliga a cada consumidor a calcular la diferencia entre el estado nuevo y el anterior, y a adivinar cuál era el significado de negocio. También significa que todos los consumidores se despiertan con cada cambio, incluso con los que no les importan.

Usaría el hecho de negocio como unidad: un evento por decisión o suceso que alguien del negocio nombraría. Que una persona compradora haga un pedido es un hecho. Que se capture un pago es otro. Cuando una transacción produce varios hechos, publica varios eventos mediante la misma escritura al transactional outbox, de modo que se confirmen o se reviertan juntos.

Una prueba práctica es preguntar si un consumidor podría reaccionar de forma significativa a este evento por sí solo. Si la respuesta es “solo después de que lleguen los tres eventos siguientes”, la granularidad es demasiado fina. Si la mayoría de los consumidores ignoran la mayor parte del payload, o el nombre del evento necesita un “y” para describirse, puede ser demasiado gruesa y esconder dos hechos.

Responsabilidad y contrato público

Cada evento tiene exactamente un responsable: el servicio responsable del agregado que el evento describe. Payments es responsable de `PaymentCaptured`. Orders es responsable de `OrderPlaced`. Los consumidores pueden leer el evento, pero solo el responsable lo define. Un consumidor que quiere otra forma se la pide al responsable, y no publica una variante propia bajo el mismo nombre.

Tratar un evento como un **contrato público** tiene consecuencias prácticas, y yo las escribiría en los acuerdos de trabajo del equipo.

- **El responsable documenta el evento:** su nombre, su significado, cuándo se emite, cuándo no, cada campo con su tipo y significado, y las garantías sobre orden y clave de partición.
- **El evento tiene una versión y un proceso de cambio.** El Capítulo 12 describe cómo evolucionar esquemas sin romper a los consumidores. La regla básica pertenece aquí: el responsable no cambia un evento publicado sin conocer las consecuencias para quienes lo leen.
- **Los consumidores se declaran.** Un registro breve de quién consume qué convierte una suscripción silenciosa en una dependencia conocida. No necesita ser sofisticado. Una lista en el repositorio que un pull request deba actualizar basta para empezar.
- **Los consumidores leen solo lo que necesitan.** La regla del tolerant reader del Capítulo 12 empieza aquí: ignora los campos que no usas, para que los cambios aditivos no te rompan.
- **La emisión es parte del comportamiento.** Un servicio que deja de emitir `OrderCancelled` en algún camino de código ha cambiado su interfaz pública tanto como uno que elimina un endpoint HTTP.

Las palabras “contrato” y “responsable” tienen peso ambas. Un contrato se puede probar. A un responsable se le puede enviar un page. Si nadie puede decir quién es responsable de un evento, nadie puede decir quién tiene permitido cambiarlo, y el escenario inicial se repite.

Trade-offs y costos

La disciplina cuesta velocidad. Mapear eventos de dominio a eventos de integración, nombrar con cuidado, escribir documentación y mantener una lista de consumidores ralentizan el primer lanzamiento. La recompensa llega después, en forma de cambios más baratos. Para un prototipo con un productor y un consumidor escritos por la misma persona, la ceremonia puede no compensar, y yo la mantendría ligera hasta que aparezca un segundo equipo.

Los fat events cambian el acoplamiento al tiempo por acoplamiento a la forma. Los consumidores ya no necesitan que el productor esté accesible, pero dependen de los campos que copia. Un payload más grande es un contrato más grande, con más que mantener estable y más que proteger.

Los thin events son baratos de publicar y caros de consumir. Cada consumidor paga una llamada de vuelta, y el productor debe atender esa carga de lectura, que normalmente alcanza su pico al mismo tiempo que el evento. Si la API del productor es lenta, el thin event traslada el problema del productor a cada consumidor.

Los envelopes cuidadosos cuestan bytes y atención. Nueve campos en cada mensaje se acumulan con volumen alto, y cada productor debe rellenarlos correctamente. La alternativa es reconstruir la información que falta durante un incidente, lo cual cuesta más. Cuánto importa esto depende de tu volumen de mensajes. El enfoque habitual es mantener el envelope pequeño y el payload ajustado.

Las reglas a nivel de cable cuestan flexibilidad. Strings para ids y dinero, un formato de marca de tiempo fijo y un envelope en el cuerpo hacen que un sistema políglota sea aburrido en el buen sentido. También implican convertir en los bordes, y un equipo que quiere tipos nativos en su propio lenguaje paga esa conversión en cada mensaje.

La expiración de un comando cuesta una dependencia del reloj.

`expires_at` protege contra comandos obsoletos, pero se juzga con el reloj del receptor. El margen que añades es tiempo durante el cual un comando realmente tardío todavía se acepta.

Los eventos anonimizados o solo con identificador limitan lo que los consumidores pueden hacer por sí solos. Un consumidor que debe contactar a quien compró necesita una llamada al responsable o una copia local de los datos que se le ha dado una razón para conservar. Eso es fricción, y es una fricción elegida.

Cuándo no poner tanta estructura alrededor de un evento. Un evento puramente interno dentro de un servicio, consumido por la misma base de código y nunca retenido, no necesita un envelope, una versión ni una lista de responsables. Reserva el tratamiento completo para los eventos que cruzan una frontera y, en particular, para los que sobreviven a un deploy.

Errores comunes

- **Publicar eventos de dominio directamente.** Tienta porque la maquinaria existe, y el resultado es que los refactors internos se convierten en cambios incompatibles. Añade un mapeo explícito a eventos de integración y trata ese mapeo como el contrato.
- **Llamar evento a un comando.** El pasado se omite o se añade de forma cosmética, y el productor termina dependiendo de un destinatario concreto sin una forma de oír un rechazo. Envía los comandos a un destinatario con nombre y espera un resultado de vuelta.
- **Eventos `Updated`.** `OrderUpdated` no dice nada sobre qué cambió, así que cada consumidor calcula diferencias de estado y adivina. Nombra el hecho de negocio, y si no existe un nombre, cuestiona si el evento hace falta.
- **Volcar todo el agregado en el payload.** Ahorra una conversación y crea un contrato público del tamaño de la tabla de la base de datos. Pregunta a cada consumidor qué hace con cada campo, y recorta lo que nadie usa.
- **Incluir datos personales y campos internos porque estaban a mano.** Corrígelo en la revisión con una lista de verificación, porque después de la publicación los datos ya están en un almacenamiento que no controlas.
- **Codificar según el valor por defecto del lenguaje.** Un nombre de clase como `event_type`, ids como números, dinero como floats, marcas de tiempo con el desplazamiento que eligiera el runtime. Funciona hasta que lo lee el segundo lenguaje. Escribe las reglas del cable una sola vez, como hace la tabla de arriba.
- **Copiar la `partition_key` del disparador.** Es correcto cuando ambos mensajes conciernen a un mismo agregado e incorrecto para un seguimiento sobre otro. Deriva la clave del agregado propio del evento.
- **Regenerar `correlation_id` en cada salto.** Alguien escribe un helper que crea un identificador nuevo por defecto. Cópialo del disparador y prueba que todos los consumidores lo hagan.
- **Usar `occurred_at` como orden.** Una marca de tiempo es una lectura de reloj, y los relojes discrepan. Usa la clave de partición y un número de versión como describe el Capítulo 9.

- **Sin responsable.** Un evento emitido por “la plataforma” o por dos servicios bajo un mismo nombre no tiene a nadie que pueda aprobar un cambio. Asigna un servicio responsable por cada tipo de evento.

Ideas clave

- Un evento enuncia un hecho, un comando solicita una acción a un destinatario con nombre y un documento lleva datos; mezclar estas tres clases lleva a acoplamiento oculto y rechazos perdidos, y un comando necesita un camino de respuesta y una expiración opcional porque solo el deadline de quien lo envía nota que se perdió.
- Los eventos de dominio describen el modelo interno de un servicio, mientras que los eventos de integración son el contrato público elegido deliberadamente, y un mapeo explícito entre ellos evita que un factor se convierta en un cambio incompatible.
- Los thin events mantienen pequeños los contratos pero reintroducen acoplamiento en tiempo de ejecución y carreras de llamada de vuelta, mientras que los fat events eliminan la llamada de vuelta pero agrandan el contrato, así que lleva lo que los consumidores necesitan para actuar y nada de lo que solo quizá quieran.
- Nombra los eventos en pasado con el sujeto primero, usa lenguaje de negocio y evita nombres de estilo `Updated` que esconden el hecho.
- El envelope existe para que las herramientas puedan deduplicar, enrutar, trazar y ordenar sin leer el payload; su codificación en el cable (ids como strings, marcas de tiempo RFC 3339 en UTC, dinero seguro, nombres de tipo neutrales respecto al lenguaje) es lo que lo mantiene legible en todos los lenguajes, y `correlation_id` se copia a lo largo del flujo mientras `causation_id` apunta al único disparador directo.
- Mantén fuera de los payloads los datos personales, los secrets, el estado interno y las instrucciones específicas de un consumidor, porque todo lo que está en un log de eventos se copia ampliamente y se elimina con dificultad.
- Cada evento tiene un servicio responsable, un significado documentado y una lista de consumidores conocidos, que es lo que convierte un mensaje en un contrato que se puede cambiar a propósito.

Un evento bien diseñado aun así tiene que cambiar, porque el negocio cambiará, y los consumidores se actualizarán a velocidades distintas mientras los eventos antiguos siguen en el log. El Capítulo 12 aborda la evolución de esquemas: cómo cambiar un contrato del que otros equipos ya dependen sin romperlos.

Capítulo 12 — Evolución de esquemas

Imagina que el equipo de Payments de Parcelly quiere admitir capturas parciales. Un vendedor envía la mitad de un pedido de dos artículos, y a la persona compradora hay que cobrarle esa mitad ahora y el resto más tarde. El evento `PaymentCaptured` lleva un único campo, `amount_minor`, que hasta ahora siempre ha significado “el pedido completo”. El equipo lo renombra a `captured_minor` y añade `remaining_minor`, lo cual es más limpio y dice lo que significa. Las pruebas del repositorio de Payments pasan y el deploy sale tarde un jueves.

Ledger no está en ese repositorio. Lee `amount_minor` para registrar un asiento de partida doble, no encuentra nada y rechaza como mal formado cada evento nuevo. Notifications lee el mismo campo para un recibo e imprime un monto vacío. Un job de informes que hace replay de los eventos del mes pasado se topa ahora con dos poblaciones en el mismo topic, nombres viejos y nombres nuevos, y su parser maneja solo una.

Nada de esto es un bug de Payments. El código hizo lo que el equipo le pidió. El fallo está en que un evento lo leen programas que su autor nunca conoce, y algunos leen eventos escritos hace meses. ¿Cómo cambias un contrato cuyos lectores no controlas, mientras las escrituras antiguas siguen en el log?

El problema y por qué es difícil

Un esquema de eventos es un contrato entre un productor y todos los consumidores, presentes y futuros. La firma de una función en tu propio código también es un contrato, pero un compilador verifica ambos lados y puedes cambiar los dos en un mismo commit. Un esquema de eventos no tiene compilador entre ambos lados. Productor y consumidores se despliegan en días distintos, pertenecen a equipos distintos y el broker acepta los bytes que lleguen.

Tres propiedades lo hacen más difícil que el versionado de una API común.

Primero, **los eventos sobreviven al código que los escribió**. En un broker basado en log, los eventos permanecen durante el período de retención configurado, y algunos sistemas los conservan indefinidamente. En un broker basado en colas la ventana es más corta, pero un backlog o una dead-letter queue (DLQ) pueden retener mensajes durante días. A un consumidor desplegado hoy se le puede pedir que lea algo escrito con un esquema de hace un año.

Segundo, **el replay convierte el pasado en una entrada viva**. El replay, definido en el Capítulo 2, es como reconstruyes una proyección, haces un backfill de un servicio nuevo o te recuperas de un bug. Entrega al código de hoy todos los esquemas históricos a la vez. Un consumidor que maneja solo la forma más reciente funciona hasta el día en que alguien hace un replay, y entonces falla primero con los eventos más antiguos. La forma es solo la mitad: el replay también aplica las reglas de negocio y los efectos secundarios de hoy a hechos antiguos, como enviar otra vez un recibo o llamar otra vez al proveedor de pagos. Un consumidor reconstruido por replay tiene que saber que está haciendo un replay, mediante un modo explícito o un rango de offsets, suprimir sus efectos externos y aplicar reglas válidas para hechos históricos. El Capítulo 7 cubre el lado de la deduplicación.

Tercero, **no puedes actualizar a todos a la vez**. Con los rolling deploys, las versiones vieja y nueva del mismo servicio corren lado a lado durante minutos u horas. Un cambio que es correcto una vez que todo se ha actualizado sigue siendo un incidente durante la actualización.

El enfoque obvio es tratar el esquema como un detalle interno y arreglar los consumidores después. Eso funciona mientras hay un consumidor y un equipo. Deja de funcionar cuando el productor no sabe cuántos consumidores existen, que es justo la situación que el diseño orientado a eventos busca crear. El Capítulo 11 sostuvo que un evento publicado es un contrato público. La evolución de esquemas es la forma de mantener esa promesa a lo largo del tiempo.

Compatibilidad, definida

Las mismas palabras se usan con soltura en muchos lugares, así que aquí van las definiciones que usa el libro.

- **Compatibilidad hacia atrás:** un consumidor que usa el esquema nuevo puede leer eventos escritos con el esquema antiguo. Lectores nuevos, datos viejos.
- **Compatibilidad hacia adelante:** un consumidor que usa el esquema antiguo puede leer eventos escritos con el esquema nuevo. Lectores viejos, datos nuevos.
- **Compatibilidad completa:** ambas a la vez.
- Las variantes **transitivas** de cada una verifican el esquema nuevo contra todas las versiones anteriores, no solo contra la última.

La dirección es fácil de invertir en la cabeza, así que prefiero hacer una pregunta concreta: “si despliego este cambio, ¿quién se rompe?”.

Compati- bilidad	Quién lee qué	Orden seguro de deploy	Protege contra
Hacia atrás	Lector nuevo, datos viejos	Primero los consumido- res, después los productores	El replay del historial, los backlogs rezagados
Hacia adelante	Lector viejo, datos nuevos	Primero los productores, después los consumidores	Los consumidores que no se pueden actualizar a demanda
Completa	Ambas direcciones	Cualquier orden	Flotas mixtas, consumidores desconocidos
Ninguna	No se garanti- za ninguna	Coordinado, con un plan de migración	Nada; consulta la sección de migración

Una advertencia sobre los nombres. Los registries y las bibliotecas de serialización usan estas palabras con pequeñas diferencias. Lee las definiciones de la herramienta que uses antes de confiar en un ajuste, y comprueba qué variante aplica tu registry por defecto: una regla no transitiva no basta para un log con replay. Prueba con eventos de ejemplo reales de cada versión.

Para un log con replay, la propiedad que necesitas es la compatibilidad hacia atrás contra todo el historial, porque un consumidor que hace replay puede encontrar cualquier versión publicada alguna vez. Para una cola

con retención corta, puede bastar con verificar contra la versión anterior. Eso es tanto una decisión de retención como una decisión de esquema.

Cambios seguros y cambios incompatibles

Un cambio es **aditivo** cuando los consumidores antiguos pueden ignorarlo y los consumidores nuevos pueden arreglárselas con su ausencia. Un cambio es **incompatible** (breaking) cuando algún lector, viejo o nuevo, ya no puede interpretar algún evento. Los veredictos siguientes suponen que los lectores se comportan como se describe en la sección del tolerant reader.

Cambio en <code>PaymentCaptured</code>	Veredicto habitual	Por qué
Añadir un campo opcional (<code>payment_method</code>)	Seguro	Los lectores viejos lo ignoran; los nuevos manejan su ausencia
Añadir un campo obligatorio sin valor por defecto	Incompatible	Los lectores nuevos no pueden interpretar los eventos viejos
Eliminar un campo opcional	Seguro solo si nadie lo lee	Los lectores que dependen de él no ven nada
Eliminar un campo obligatorio	Incompatible	Los lectores viejos lo esperan
Renombrar un campo	Incompatible	Se lee como una eliminación más una adición; unos pocos formatos pueden resolver un renombrado con alias
Cambiar un tipo (<code>int</code> a <code>string</code>)	Incompatible	Los parsers fallan o convierten en silencio
Ampliar un tipo numérico	Depende del formato	Algunos formatos lo promueven, otros no
Cambiar el significado de un campo	Incompatible, y del peor tipo	Nada falla; los resultados son incorrectos
Añadir un valor a una enumeración	Riesgoso	Los lectores viejos pueden rechazarlo o manejarlo mal

Dos filas merecen una mirada más detenida.

Cambiar el significado es el cambio peligroso

Renombrar un campo rompe con estruendo. Ledger falla en el primer evento, salta una alerta y alguien lee el stack trace. Cambiar lo que significa un campo, dejando intactos el nombre y el tipo, rompe en silencio. Imagina que Payments decide que `amount_minor` pasará a significar “monto capturado en este evento” en lugar de “monto total del pedido”. Todos los parsers siguen funcionando. Ledger registra montos parciales como si fueran totales, y el error aflora semanas después en una conciliación (Capítulo 13) como una diferencia que nadie sabe explicar.

Ninguna herramienta de esquemas lo detecta, porque el esquema no ha cambiado. Solo lo detecta una persona que revise el cambio. Mi regla: si el significado cambia, es un campo nuevo o un tipo de evento nuevo, nunca una edición del antiguo.

Las enumeraciones son un contrato oculto

Añadir un valor a una enumeración es aditivo en el esquema y a menudo no lo es en los consumidores. Un consumidor que mapea `payment_method` a una rama de manejo, con una rama `else` que supone “card”, manejará mal un nuevo “`bank_transfer`” sin ningún error. Los productores pueden documentar que las enumeraciones son abiertas. Los consumidores pueden tratar los valores desconocidos como desconocidos y enviarlos a un error visible o a un valor por defecto seguro, en lugar de absorberlos en un caso conocido.

Schema registries y formatos

Un **schema registry** es un servicio que almacena los esquemas que usa un sistema, les asigna identificadores y puede verificar un esquema propuesto contra los anteriores. Cumple dos funciones que conviene separar.

1. **Consulta.** Un mensaje lleva, o apunta a, el identificador del esquema con el que se escribió. Un consumidor obtiene ese esquema y decodifica los bytes. Esto es lo que hace viables los formatos binarios compactos: los nombres de los campos no viajan con cada mensaje.
2. **Control de acceso.** Antes de que un productor registre un esquema nuevo, el registry lo compara con las versiones anteriores según una regla de compatibilidad elegida y lo rechaza si rompe la regla.

La segunda función es la que defendería con más firmeza. Las reglas que viven en una lista de verificación de revisión se omiten el día en que el equipo está ocupado. Una verificación en el pipeline de deploy no se ocupa.

Subjects, topics y tipos de evento

Un registry verifica la compatibilidad por **subject**, y que un subject sea un topic o un tipo de evento decide si varios tipos de evento pueden compartir un topic. Compartir es lo que necesita el orden por agregado entre tipos de evento (Capítulo 9): el `PaymentAuthorized` y el `PaymentCaptured` de un mismo pago solo se mantienen en orden si viajan por la misma partición del mismo topic. Los ejemplos de Parcelly suponen un topic por tipo de agregado, como `payments.payment-events`, que transporta varios tipos de evento con clave igual al id del agregado, con la compatibilidad configurada por tipo de evento y no por topic. El costo es que los consumidores filtran por `event_type`, y que un topic versionado (Estrategia 3) mueve de golpe todos los tipos de evento de ese agregado. Un topic por tipo de evento es más simple de configurar y renuncia al orden entre tipos.

Tres familias de formato

El libro es agnóstico respecto del formato, así que esto es orientación, no una comparación de productos. Avro, Protobuf y JSON Schema difieren en el aspecto que tienen los datos en la red, lo cual condiciona qué es seguro cambiar.

Fami- lia	En la red	Campos identifi- cados por	Conse- cuencia práctica	Trampa
Estilo Avro	Binario compac- to, sin nom- bres de campo	Nombre, resuelto entre el esquema del escri- tor y el del lector	El lector ne- cesita el es- quema del escritor; los valores por defecto sos- tienen la compatibili- dad	Los alias permiten que un lector resuel- va un campo renombrado, así que un re- nombrado no siempre es incompatible. Un campo ausente se rellena con el valor por defecto del lector, de modo que la ausencia es invisible

Famili-a	En la red	Campos identi-ficados por	Conse-cuencia práctica	Trampa
Estilo Proto-buf	Binario compac-to	Número de campo	Renombrar es barato; reutilizar un número es peligroso	Barato solo en el wire binario; ProtoJ-SON y el código generado usan nombres, así que un renombrado rompe el código fuente y el JSON. Un escalar de proto3 sin presencia explícita decodifica un campo ausente como 0
JSON Sche-ma	JSON de texto	Nombre; esquema opcional en runtime	Fácil de depurar; la compatibili-dad es una convención si no se hace cumplir	Con un modelo cerrado (<code>additionalProperties: false</code>), añadir un campo rompe a los lectores viejos, y los registries juzgan de forma distinta los modelos abiertos y los cerrados

Tres ideas se aplican a todos ellos.

- **Los valores por defecto hacen seguras las adiciones, si el valor por defecto es honesto.** Un campo nuevo con un valor por defecto definido permite que un lector nuevo procese un evento viejo. Sin él, una adición se convierte en una ruptura. Para un campo que el lector debe poder detectar como ausente, como un monto, el valor por defecto tiene que ser “sin valor”: una unión de Avro de `null` y el tipo con `null` por defecto, o `optional` de proto3 o un tipo wrapper. Un valor por defecto de 0 decodifica un evento viejo como una captura de cero, y el lector no puede notar la diferencia. Entonces el lector comprueba la presencia, no el valor.
- **La identidad no debería reutilizarse.** Ya sea que un campo se identifique por nombre o por número, darle a una identidad antigua un significado nuevo es una ruptura. Retira los identificadores; no los recicles.
- **Los campos desconocidos deberían tolerarse.** Que un decodificador conserve, ignore o rechace los campos que no conoce depende del formato y de su configuración. Averígualo antes de diseñar en torno a ello.

Si usas JSON sin un lenguaje de esquemas, igual obtienes la mayor parte del beneficio guardando los archivos de esquema en control de versiones, validando en las pruebas del productor y conservando eventos de ejemplo

reales por versión. El registry añade consulta central y control de cumplimiento. No añade el razonamiento.

Dónde vive la versión

El envelope estándar lleva `schema_version` como un entero junto a `event_type`, y un consumidor lee el par para saber qué forma esperar. Mantenerlo en el envelope significa que un router, un contador de métricas o una herramienta de DLQ pueden verlo sin decodificar el payload. “¿Cuántos eventos con `schema_version: 1` siguen llegando?” pasa a ser una consulta barata en lugar de un proyecto.

Aquí es fácil confundir dos números, así que este capítulo les da nombres separados. La **versión incompatible** (breaking version) es el `schema_version` del envelope. Solo cambia con un cambio que un lector viejo no puede manejar: un renombrado, una eliminación, un cambio de tipo o un cambio de significado. La **revisión del registry** es el identificador que el registry asigna a cada esquema registrado, y cambia con cada modificación registrada, incluidas las aditivas. Un consumidor rechaza un evento cuya versión incompatible no conoce. Lee un evento con una revisión del registry más reciente pero con una versión incompatible conocida como la forma más reciente que conoce, e ignora lo que no reconoce. Ninguno de los dos es la versión del agregado del Capítulo 4, que cuenta los cambios de un pedido y no tiene nada que ver con la forma del payload.

Si cada cambio aditivo incrementara `schema_version`, todo consumidor aún sin actualizar rechazaría cada evento nuevo, y la compatibilidad hacia adelante se perdería. El precio de la separación es tener que distinguir dos números. Un ejemplo de esta misma línea es BabelQueue (babelqueue.-com), un estándar abierto de envelope independiente del lenguaje que congela su envelope en `schema_version 1`, permite campos nuevos solo como opcionales e ignorables, exige un incremento de versión para un renombrado, una eliminación o un cambio de tipo, hace que los consumidores rechacen una versión que no conocen, y evita que los campos desconocidos rompan el procesamiento y los preserve cuando el mensaje se publica de nuevo.

Un evento tal como lo publica Payments durante el paso de dual-publish de la migración que sigue:


```
{
  event_id:      "9c04b6e2-...",
  event_type:    "PaymentCaptured",
  schema_version: 1,
  occurred_at:   "2026-10-05T11:20:44Z",
  producer:     "payments",
  correlation_id: "order-90412",
  causation_id:  "5ad1e07c-...",
  partition_key: "order-90412",
  payload: {
    order_id:      "order-90412",
    amount_minor:  4990,
    captured_minor: 4990,
    currency:      "EUR"
  }
}
```

Compatibilidad entre lenguajes

Un esquema puede ser perfectamente compatible según las reglas del registry y aun así romperse entre dos lenguajes, porque las reglas describen la estructura y la ruptura está en cómo la lee un runtime. El Capítulo 11 fijó la codificación del envelope. Estas son las trampas del payload y de lo que hace un consumidor con los campos que no conoce.

Los campos desconocidos se pierden al re-emitar. Supón que un consumidor en Node de Notifications deserializa cada `PaymentCaptured` en una clase con `order_id`, `captured_minor` y `currency`, y copia el evento a un topic orientado a socios. Cuando Payments añada después `payment_method`, la copia lo descarta en silencio. Nada falla. El campo nunca está. Los decodificadores binarios de Protobuf suelen conservar los campos desconocidos; muchos mapeadores de JSON tipados no. Regla: un salto que reenvía un mensaje preserva los campos desconocidos, ya sea llevando el payload en bruto o un mapa genérico, o publica un evento nuevo propio bajo su propio contrato.

Números. `captured_minor` cabe sin problema en el entero de cualquier lenguaje. Un identificador o contador int64 no cabe en un número JSON de JavaScript más allá de $2^{53} - 1$, y el consumidor lo redondea sin error. Regla: documenta el rango de cada campo numérico, y envía como string todo lo que pueda excederlo, y todo decimal.

Timestamps. `timestamp-millis` y `timestamp-micros` de Avro guardan el mismo tipo de valor a escalas distintas. Un escritor en microsegundos leído como milisegundos produce una fecha a decenas de miles de años de distancia, y aun así se parsea. Regla: fija la unidad en el esquema y no la cambies nunca; un cambio de unidad es una versión incompatible.

Enumeraciones. El código generado maneja de forma distinta un valor desconocido: algunos lo conservan, otros lo mapean a un valor por defecto, otros lanzan una excepción. Regla: trata toda enumeración como abierta y decide en el consumidor qué significa lo desconocido.

Null y ausente. Muchos decodificadores no los distinguen, y algunos rellenan el hueco con un valor por defecto. Regla: para los campos cuya ausencia importa, usa las formas con detección de presencia del punto sobre valores por defecto de más arriba.

Nada de esto requiere una versión concreta de una biblioteca concreta. Todo requiere que una persona pruebe con un consumidor en cada lenguaje que Parcelly ejecuta, un costo que pueden asumir las pruebas de contrato del Capítulo 15.

Estrategias de versionado

Hay tres maneras de introducir un cambio, de la más barata a la más cara. Las probaría en orden y subiría solo cuando el peldaño inferior no pueda sostener el cambio.

Estrategia 1: añadir un campo opcional

Si el cambio es algo nuevo que los lectores viejos pueden ignorar, añade un campo opcional. Añadir `payment_method` a `PaymentCaptured` es el caso modelo. Registra una nueva revisión del registry, y la regla de compatibilidad decide si el registry la acepta. No cambia `schema_version`, porque ningún lector viejo pierde nada. Incrementar la versión incompatible por un cambio aditivo haría que todo consumidor aún sin actualizar rechazara los eventos nuevos, lo que anula el sentido de añadir el campo de forma compatible.

Esta estrategia no puede expresar un renombrado, un cambio de tipo, la eliminación de algo obligatorio ni un cambio de significado. Para esos casos, sube un peldaño.

Estrategia 2: introducir un nuevo tipo de evento

Cuando el significado cambia, dilo con un nombre nuevo. Supón que Parcelly trata la captura parcial como un hecho de negocio distinto de una captura completa. El modelo honesto es un evento nuevo, nombrado según las reglas del Capítulo 11. Este capítulo introduce `PaymentPartiallyCaptured` como un evento adicional.

El evento antiguo conserva su significado y sus consumidores siguen funcionando. El costo es que ahora existen ambos eventos, y un consumidor que quiera “todas las capturas” debe suscribirse a los dos. Es el trade-off correcto cuando el hecho nuevo es conceptualmente distinto, y el incorrecto cuando es el mismo hecho con una forma mejor. Ese segundo caso es un problema de representación, y un upcaster le encaja mejor.

Estrategia 3: un topic versionado

Cuando un cambio incompatible afecta a todo el stream, pon la forma nueva en un topic nuevo, por ejemplo `payments.payment-events.v2`. Los consumidores migran a su propio ritmo, y el topic antiguo se retira cuando se va su último lector. Con la disposición descrita arriba, el topic nuevo transporta todos los tipos de evento del agregado.

Es la opción más clara y la más cara. Las claves de partición, el orden y los offsets de los dos topics no guardan relación, así que un consumidor que pasa de uno a otro debe decidir desde dónde empezar a leer y cómo evitar procesar un hecho dos veces. Las reglas de orden del Capítulo 9 se aplican a cada topic por separado, y nada ordena los eventos entre ellos. En un broker basado en colas, el equivalente es una segunda cola con sus propios bindings, su manejo de dead letters y sus alertas. Recurriría a un topic versionado solo ante una ruptura real que un upcaster no pueda salvar, y solo cuando pueda nombrar a quien va a borrar el antiguo.

Upcasting y el tolerant reader

Dos técnicas permiten a los consumidores arreglárselas con varias versiones sin obligar al productor a congelarse.

Upcasting

Un **upcaster** convierte un evento escrito con un esquema antiguo al esquema actual antes de que la lógica de negocio lo vea. El código de negocio se escribe contra una sola forma, y la conversión de cada versión histórica vive en un único lugar.

```
function upcastPaymentCaptured(event):
    p = event.payload
    if event.schema_version == 1:
        // during the migration a v1 event carries either field, or
        both

        // check presence, not value
        if hasField(p, "captured_minor"):
            captured = p.captured_minor
        else:
            // v1 only ever described full captures, so the
            // order amount is the captured amount; may be absent
            captured = p.amount_minor
        return { order_id: p.order_id, captured_minor: captured,
            currency: p.currency }
    else if event.schema_version == 2:
        // v2 carries captured_minor only
        return { order_id: p.order_id, captured_minor:
            p.captured_minor, currency: p.currency }
    else:
        // an unknown newer breaking version must not be guessed at
        raise UnsupportedSchemaVersion
```

El comentario de la rama de la versión 1 lleva el trabajo real. Un upcaster codifica una afirmación sobre el pasado, y esa afirmación tiene que ser verdadera. Si algunos eventos antiguos fueron parciales en la práctica, el upcaster informa en silencio de un monto parcial como si fuera la captura completa. Prueba los upcasters con eventos archivados reales, no inventados.

`UnsupportedSchemaVersion` necesita una clasificación propia. Para el código actual es permanente, ya que reintentar no cambia nada, pero un deploy del consumidor lo cura. Por eso se retiene para un redrive una vez ac-

tualizado el consumidor (Capítulo 8), y no se envía al parking lot, donde esperaría a una persona que no tiene nada que decidir.

El lado del consumidor, dentro del handler antes de la lógica de negocio, es el lugar más simple para empezar. Reescribir el historial en un topic nuevo es útil cuando muchos consumidores necesitarían, de otro modo, el mismo upcaster. Reescribir en el sitio los eventos almacenados es una decisión distinta: los sistemas con event sourcing suelen negarse a hacerlo, porque el log es evidencia. Yo respetaría eso salvo que haya una razón legal o de seguridad, y entonces lo ejecutaría como una migración de datos con un plan.

El tolerant reader

Un **tolerant reader** toma de un evento solo lo que necesita e ignora el resto. Lee `order_id` y `captured_minor` y no le importa que haya además un `payment_method`. Es liberal en lo que acepta y estricto en aquello de lo que depende.

```
function readPaymentForLedger(event):
    payload = upcastPaymentCaptured(event)

    if not hasField(payload, "order_id") or
        not hasField(payload, "captured_minor") or
        not hasField(payload, "currency"):
        // a missing dependency will fail on every attempt,
        // so it is a poison message, not a retry case
        raise InvalidEvent

    return {
        order_id: payload.order_id,
        amount:   payload.captured_minor,
        currency: payload.currency
    }
```

La tolerancia tiene dos caras. Ignorar los campos desconocidos es seguro cuando los productores añaden cosas. Sustituir en silencio por cero un monto ausente es peligroso. El lector de arriba se niega a continuar cuando falta un campo del que depende, y pregunta por la presencia, no por el valor. Eso solo es honesto si el decodificador puede distinguir un campo ausente de un cero, y por eso importan los valores por defecto de más arri-

ba. Esa negativa es una clasificación: el evento nunca podrá tener éxito, así que toma la ruta de fallo permanente del Capítulo 6 y termina como un dead letter, como describe el Capítulo 8.

Nota: La tolerancia es un acuerdo entre ambas partes. El productor promete añadir y no cambiar el significado. El consumidor promete ignorar lo que no entiende y rechazar lo que no puede hacer con seguridad. Si cualquiera de las dos partes rompe su mitad, la otra no puede salvar el sistema.

El registry verifica al productor antes de que publique. El tolerant reader protege al consumidor después de que algo se cuele. No confiaría en uno en lugar del otro.

Migrar un cambio incompatible

A veces una ruptura no se puede evitar: la forma antigua está mal, una regulación obliga a sacar un campo o dos servicios deben remodelarse juntos. La respuesta es convertir un deploy incompatible en varios deploys compatibles. Es el patrón **expand and contract** aplicado a los eventos: expandir el contrato para admitir ambas formas, migrar a todos los lectores y luego contraer eliminando la forma antigua.

Un plan de migración para `PaymentCaptured`

Digamos que Parcelly realmente debe renombrar `amount_minor` a `captured_minor`, y que nada más del evento cambia. El valor conserva su significado: el monto capturado por el pedido completo. Es un renombre puro, y el plan de abajo solo es seguro por eso. Las capturas parciales son un cambio de significado y se tratan por separado, después de los pasos.

1. **Registrar la adición.** La nueva revisión del registry conserva `amount_minor` y añade `captured_minor` como campo opcional cuyo valor por defecto es “sin valor”, no 0. Bajo una regla de compatibilidad hacia atrás, el registry la acepta. `schema_version` sigue en 1, porque ningún lector viejo pierde nada.
2. **Actualizar primero los consumidores.** Cada consumidor aprende a leer `captured_minor` y a recurrir a `amount_minor` cuando falta. Tam-

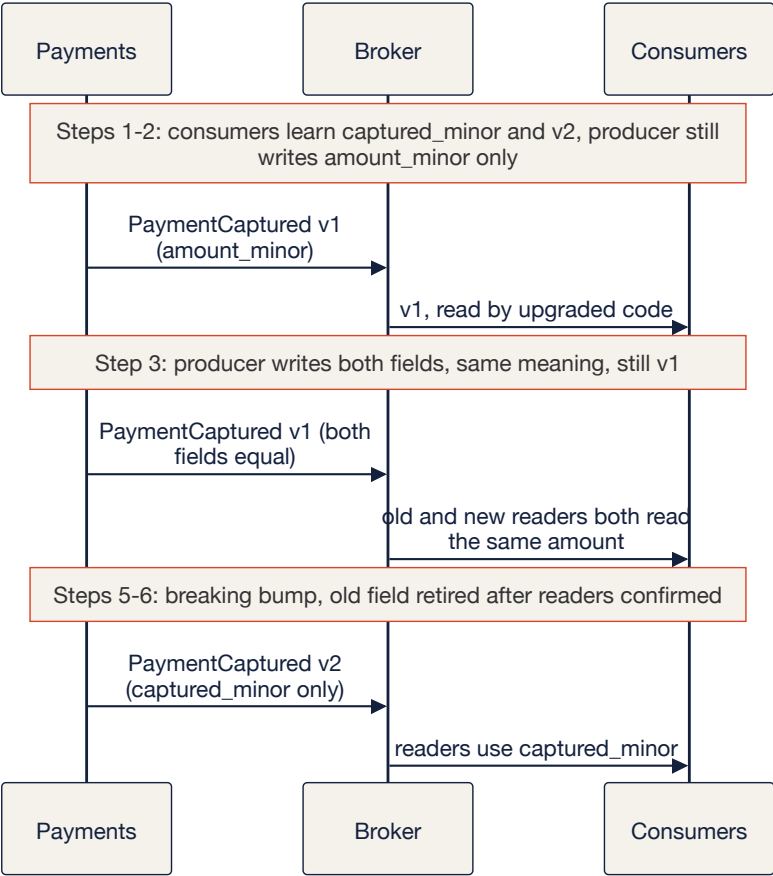
bién aprende a leer la versión incompatible 2, que aún no existe, para que el incremento del paso 5 no encuentre a nadie que lo rechace. Los lectores nuevos existen antes que los datos nuevos.

3. **Dual-publish.** Payments escribe ambos campos en cada evento, todavía con `schema_version` 1, o, si el cambio requiere un nuevo tipo de evento o un nuevo topic, publica la forma antigua y la nueva del mismo hecho. Esto no es el dual write del Capítulo 4: ambos mensajes salen de la misma transacción del transactional outbox, así que se mantiene la garantía habitual.
4. **Observar.** Mide quién sigue leyendo el campo antiguo o el topic antiguo. Las métricas del productor sobre `schema_version` muestran qué se sigue escribiendo; la actividad de los grupos de consumidores en el topic antiguo, la lista de consumidores del Capítulo 11 y las pruebas de contrato del Capítulo 15 muestran quién sigue leyéndolo. Un consumidor que nunca se mueve es una conversación pendiente, no un número que ignorar.
5. **Dejar de escribir la forma antigua.** Payments publica solo `captured_minor` e incrementa `schema_version` a 2, porque este es el único paso que un lector viejo no puede sobrevivir. Un consumidor que se perdió el paso 2 rechaza el evento, y este va a la DLQ para un redrive una vez actualizado ese consumidor (Capítulo 8). El paso 4 existe para mantener vacía esa lista. `amount_minor` permanece en el esquema como obsoleto hasta que el evento más antiguo que lo lleva haya salido de la retención.
6. **Contraer.** Elimina el campo antiguo y retira el topic antiguo si había uno. Quita el fallback de los consumidores solo cuando el evento más antiguo sin `captured_minor` haya salido de la retención, o muévelo a un upcaster permanente.

Las capturas parciales quedan fuera de este plan. Una vez que una captura puede ser parcial, `amount_minor` no tiene un valor honesto. Si lleva el monto capturado, su significado ha cambiado en el sitio, y un lector que se saltó el paso 2, ya sea desconocido o revertido, registra el monto parcial como si fuera el total del pedido, sin ningún error. Si lleva el total del pedido, ese mismo lector registra demasiado. El registry aprueba ambas versiones, porque la estructura no cambia. Esa es la lección: una verificación estructural de compatibilidad no puede ver un cambio de significado. Pon

el significado nuevo en un campo nuevo, en un nuevo tipo de evento (Estrategia 2, `PaymentPartiallyCaptured`) o en una nueva versión mayor en un topic nuevo. Y entonces no publiques capturas parciales hasta que todos los lectores hayan pasado a la forma nueva, y publícalas después únicamente en la forma nueva.

El orden es el corazón del plan. Para un cambio compatible hacia atrás, los consumidores van primero; para uno compatible hacia adelante, los productores pueden ir primero. Cuando no puedas saber cuál se aplica, ejecuta el plan tal como está escrito y conserva ambos campos durante un tiempo.



Dual-publish entre topics

Cuando el plan usa un topic versionado, el paso 3 publica el hecho dos veces, una en cada topic. Un consumidor que lee ambos no debe procesar dos veces el mismo hecho de negocio. Yo daría a las copias valores de

`event_id` distintos, porque son mensajes distintos, y llevaría el id del original en un campo documentado. Elijas lo que elijas, la clave de idempotencia del consumidor para ese hecho debe definirse a propósito. El Capítulo 7 cubre las claves; lo que importa aquí es que una migración es justo el momento en que se ponen a prueba los supuestos de idempotencia.

Advertencia: Durante el dual-publish, no dejes nunca que un consumidor lea el stream antiguo y el nuevo y aplique ambos. Un consumidor con dos fuentes para un mismo hecho necesita una regla de deduplicación, escrita, o registrará dos veces un asiento en el libro mayor.

Rollbacks e historial

El rollback forma parte del plan. Si ya se publicaron eventos de la versión 2, revertir el productor no los quita del log. Un consumidor revertido a un código que no puede leer la versión 2 falla ahora con eventos que ya existen. Por eso los consumidores deben poder leer la versión 2 antes de que cualquier productor la escriba, y el destino de un rollback de consumidor debe poder leer todas las versiones en retención, la versión 2 incluida. Un código que lee solo la versión 1 es un destino de rollback que hay que evitar, aunque haya sido el último release antes del cambio.

La compatibilidad hacia atrás maneja los eventos antiguos sin tocarlos. Si quieres borrar el código de fallback, tienes tres opciones: conservar el up-caster hasta que caduquen los eventos antiguos, reescribir los eventos almacenados en un topic nuevo, o guardar un snapshot del estado en otro lugar y dejar que el historial se extinga. Cada una cuesta algo: código permanente, una migración en la que debes confiar o la capacidad de hacer replay. No sé cuál debería elegir tu sistema sin saber con qué frecuencia haces replay y durante cuánto tiempo esperan tus reguladores que conserves los registros.

Deprecar eventos

Es más fácil añadir eventos que quitarlos, porque un consumidor que no conoces puede depender de cualquiera. Un ciclo de vida razonable tiene cuatro etapas.

1. **Anunciar.** Marca el evento como obsoleto en el registry o en el catálogo de eventos, nombra el reemplazo, indica una fecha de retiro y un contacto.
2. **Medir.** Averigua quién sigue leyéndolo: la pertenencia a grupos de consumidores en un log, los bindings en una cola, las consultas al registry. Trata la lista como evidencia, no como prueba, porque un job por lotes mensual puede estar ausente hoy y presente la semana que viene.
3. **Dejar de producir.** Pasada la fecha, deja de publicar.
4. **Retirar.** Elimina el esquema y el topic o la cola una vez que la retención haya hecho caducar el último evento y los lectores se hayan ido.

Dos precauciones. Una deprecación sin plazo nunca termina, así que pon la fecha por escrito y cúmplela. Y un evento obsoleto del que se hace replay después de su retiro fallará, así que decide qué debería hacer el replay para el período afectado antes de retirarlo.

La responsabilidad importa. El equipo del productor es responsable del evento y de su deprecación, como describió el Capítulo 11. Los consumidores son responsables de sus migraciones. Una política que deje decidir al productor por sí solo, o que permita a los consumidores negarse indefinidamente, fracasará. La política útil es un aviso del productor y una fecha que cumplen los consumidores.

Trade-offs y costos

- **Un registry es otro servicio que operar.** Está en el camino del deploy de todo productor y, según la configuración, de la decodificación de los consumidores. Necesita disponibilidad, control de acceso y un responsable, y los clientes necesitan un plan para cuando no sea alcanzable.
- **Las reglas de compatibilidad frenan a los productores, deliberadamente.** Un equipo que podría haber renombrado un campo en una tarde ahora ejecuta una migración de seis pasos. Ese es el precio de los deploys independientes.
- **Los upcasters se acumulan.** Cada versión histórica añade una rama y una prueba, y el código no se simplifica hasta que los eventos antiguos

salen de la retención. Una retención indefinida significa upcasters indefinidos.

- **Los tolerant readers ocultan errores.** Un consumidor que ignora los campos desconocidos tampoco nota cuándo un productor empieza a enviar algo que debería importarle. La tolerancia sin monitoreo convierte fallos ruidosos en deriva silenciosa.
- **El dual-publish multiplica el tráfico y el trabajo de deduplicación** mientras dure la migración.
- **Dos números de versión en lugar de uno.** La versión incompatible y la revisión del registry evitan que los cambios aditivos hagan fallar a los consumidores, y le cuestan al equipo el esfuerzo de distinguirlas en los dashboards y en las conversaciones.
- **Los números de versión no son documentación.** `schema_version: 4` dice que algo cambió, no qué ni a quién afecta. Un registro de cambios guardado junto al esquema vale más que el número.
- **Las reglas entre lenguajes cuestan comodidad.** Strings para los números grandes, preservación del payload en bruto y unidades de timestamp fijas significan código de conversión adicional en cada lenguaje, y una prueba por lenguaje.

Cuándo no usar estos patrones

No construyas toda la maquinaria para un evento con un productor, un consumidor y un equipo que despliega ambos juntos. Un deploy coordinado es más barato, y el riesgo del que te asegurarías no existe. Vuelve a evaluarlo cuando aparezca un segundo consumidor, sobre todo uno que pertenezca a otro equipo.

No trates el modo estricto de un registry como un sustituto del diseño. Una secuencia de adiciones seguras puede dejar un evento con 40 campos opcionales y sin un propósito claro. Si un evento sigue creciendo, probablemente son dos eventos, y se aplican las reglas de granularidad del Capítulo 11.

No crees un topic versionado por cada cambio. Un sistema que lo hace termina con muchos streams a medio retirar y nadie que sepa cuál es el vigente.

Errores comunes

- **Tratar un renombrado como un cambio pequeño.** Es tentador porque el nombre nuevo es más limpio y las pruebas pasan. Para todo lector es una eliminación más una adición. Añade el nombre nuevo junto al antiguo, migra a los lectores y luego quita el antiguo.
- **Cambiar el significado de un campo en el sitio.** El nombre y el tipo se mantienen, así que ninguna herramienta objeta. Da al significado nuevo un campo o un tipo de evento nuevo.
- **Añadir un campo obligatorio sin valor por defecto.** Parece una adición, pero a los eventos viejos les falta. Hazlo opcional con un valor por defecto, o introdúcelo con una versión nueva y un upcaster.
- **Probar la compatibilidad solo contra la última versión.** El replay lee todo lo que hay en retención. Verifica contra cada versión que siga ahí y conserva eventos de ejemplo de cada una en las pruebas de los consumidores (Capítulo 15).
- **Actualizar los productores antes que los consumidores.** El equipo del productor es responsable del cambio y está listo primero. Para un cambio compatible hacia atrás, los consumidores van primero. Confirma el orden antes del deploy.
- **Fallar con estruendo ante campos desconocidos.** Un parser estricto convierte cada cambio aditivo en un incidente. Ignora lo que no uses; rechaza solo aquello de lo que dependes.
- **Aplicar en silencio un valor por defecto a un campo ausente.** Sustituir por cero un monto ausente oculta un defecto, y un decodificador que rellena con 0 lo oculta antes de que tu código se ejecute. Usa un valor por defecto “sin valor” para esos campos, comprueba la presencia y lanza un error para que la ruta de fallo retenga el evento para una persona.
- **Incrementar `schema_version` en cada cambio.** Cada cambio aditivo se convierte entonces en un fallo duro para todo consumidor aún sin actualizar. Incrementa la versión incompatible solo para los cambios que un lector viejo no puede manejar, y deja que la revisión del registry cuente el resto.
- **Reenviar a través de una clase tipada.** La copia descarta todo campo que la clase no declare, y nada se queja. Preserva los campos desconocidos al re-emitir, o publica un evento nuevo bajo tu propio contrato.

- **Reutilizar un número de campo o el nombre de un topic.** Un número reciclado lee los datos antiguos como el campo nuevo, y un topic reutilizado mezcla dos formas. Retira los identificadores.
- **Deprecar sin una fecha ni un responsable.** El evento persiste y el siguiente equipo teme quitarlo. Da a toda deprecación una fecha, un reemplazo y una persona que responda las preguntas.
- **Revertir un consumidor sin comprobar el esquema.** Verifica que el destino del rollback pueda leer todas las versiones en retención.

Ideas clave

- Un esquema de eventos es un contrato con lectores que no controlas y con datos escritos hace mucho; compatibilidad hacia atrás significa que los lectores nuevos pueden leer datos viejos, hacia adelante que los lectores viejos pueden leer datos nuevos, y un log con replay necesita compatibilidad hacia atrás contra toda versión que siga en retención.
- Añadir un campo opcional es seguro, mientras que los renombrados, los cambios de tipo, los campos obligatorios nuevos y los cambios de significado son incompatibles, y un cambio de significado es el peor porque nada falla con estruendo; solo un cambio incompatible mueve `schema_version`, y la revisión del registry cuenta el resto.
- Un schema registry se gana su lugar haciendo cumplir la compatibilidad en el pipeline de deploy, pero verifica la estructura y no el significado, así que una persona sigue revisando qué se supone que dice un campo.
- Prefiere la estrategia de versionado más barata que sostenga el cambio: primero un campo opcional, un nuevo tipo de evento cuando el significado es un hecho distinto, y un topic versionado solo cuando todo el stream debe romperse.
- Los upcasters y los tolerant readers permiten a los consumidores arreglárselas con varias versiones, siempre que el upcaster codifique una afirmación verdadera sobre el pasado y el lector compruebe la presencia, no el valor, y rechace lo que no puede interpretar con seguridad.
- Entre lenguajes, las rupturas silenciosas son los campos desconocidos perdidos, los enteros grandes, las unidades de timestamp, las enume-

raiones cerradas y null frente a ausente; así que preserva lo que no conoces y deja por escrito la representación.

- Migra un cambio incompatible como una secuencia de cambios compatibles, con el incremento en el único paso que un lector viejo no puede sobrevivir, y planifica los rollbacks en el mismo documento.

Un contrato que evoluciona con seguridad solo tranquiliza si puedes ver qué fluye por el sistema, y ahí es donde empieza el Capítulo 13.

PARTE V

Operar y demostrar

Capítulo 13 — Observabilidad y conciliación

Imagina una mañana de jueves en Parcelly. Soporte reenvía un ticket: una persona compradora pagó un pedido hace dos días, el extracto de la tarjeta muestra el cargo y la página del pedido sigue diciendo “Awaiting confirmation”. Soporte abre la pantalla de administración de Orders y ve `PAYMENT_PENDING`. Preguntan al equipo de Payments, que encuentra una fila `PaymentCaptured` en su base de datos y un outbox limpio. Preguntan al equipo de Orders, cuyos logs de consumidor no muestran ningún error para ese pedido. El dashboard de todos está en verde.

Tres ingenieros buscan ahora en los logs de tres servicios por número de pedido. Una línea de log usa `order-81724`, otra usa `81724`, y el relay de Payments registra solo el id interno de la fila. Tras una hora encuentran el evento en el outbox de Payments, marcado como publicado. Tras otra hora descubren que el topic del broker que alimenta a Orders tenía un grupo de consumidores que se quedó atrás durante un deploy. La retención del broker borró un tramo de mensajes que el grupo no había leído, y la política de reinicio de offset del grupo estaba configurada para saltar a la posición más reciente, así que retomó después del hueco sin lanzar ningún error. El evento existió, se publicó y nunca se leyó.

Nada de esto fue un bug de código en el sentido que señalaría un revisor. El sistema hizo lo que se construyó para hacer, y nadie pudo ver que lo hacía. Peor aún, el ticket existe porque una persona compradora lo notó antes que cualquier alarma, y nadie sabe cuántos otros pedidos están igual. Este capítulo plantea dos preguntas. ¿Qué necesita ver un equipo para responder “¿dónde está este pedido?” en minutos, y cómo encuentra un equipo los pedidos sobre los que nadie se ha quejado todavía?

El problema y por qué es difícil

En un sistema síncrono, un fallo suele llegar con un stack trace asociado a una solicitud. En un sistema orientado a eventos, el trabajo de un flujo de negocio se reparte entre varios servicios, varias bases de datos, un broker y varios momentos en el tiempo. Ningún proceso aislado guarda la historia completa, y ningún log la contiene.

Tres propiedades lo hacen difícil. Primero, **los saltos están desacoplados en el tiempo**. El productor termina y se olvida, y el consumidor puede ejecutarse segundos u horas después, en otro proceso, sin nada de la pila de llamadas del productor. Las herramientas que siguen una solicitud a través de llamadas anidadas la pierden en el primer mensaje.

Segundo, **los peores fallos son silenciosos**. El Capítulo 3 describió los mensajes sin leer que un broker borra cuando un consumidor se retrasa más allá de la retención. El Capítulo 8 describió mensajes que se procesaron de forma incorrecta sin lanzar un error. En ambos casos nada falla, así que nada alerta. Un sistema solo puede notar un evento que nunca llegó si sabe que debería haber llegado.

Tercero, **componentes de aspecto sano pueden sumar un flujo roto**. El relay está en marcha, el broker está arriba, el consumidor no reporta errores, y un pedido sigue atascado. La salud de los componentes responde “¿está vivo este proceso?” y el negocio pregunta “¿se completó este pedido?”. Son preguntas distintas, y la brecha entre ellas es donde trabaja este capítulo.

El Capítulo 3 describió la semántica de un fallo como tres preguntas: qué puede fallar, cómo lo observarías y qué garantía sobrevive a él. Este capítulo trata de la segunda pregunta, y de qué hacer cuando la respuesta resulta ser “no lo observarías”.

El enfoque tiene dos mitades. La observabilidad te da los medios para seguir un flujo y ver cómo se acumula la presión. La **conciliación**, que compara las vistas de la verdad de dos sistemas y repara las diferencias, te da una manera de encontrar lo que ninguna señal reportó. Un equipo que solo tiene la primera mitad ve los fallos ruidosos. Un equipo que solo tiene la segunda encuentra los problemas con días de retraso. Yo construiría ambas.

Llevar el contexto a través de saltos asíncronos

Todo evento ya lleva dos identificadores del Capítulo 11:

`correlation_id`, compartido por todos los eventos de un flujo de negocio, y `causation_id`, que apunta al desencadenante directo. Son la columna vertebral de la historia. El distributed tracing añade un segundo hilo, que registra los tiempos y la relación padre-hijo entre unidades de trabajo.

Un **trace** es un árbol de spans con tiempos, cada uno una unidad de trabajo en un proceso. Dentro de un proceso, la biblioteca de tracing guarda el span actual en un estado ambiental. A través de un mensaje, ese estado desaparece a menos que lo pongas en el mensaje. El enfoque estándar es serializar el contexto de trace, un trace id, un span id y una bandera de muestreo, en los headers del mensaje. W3C Trace Context define un formato común para esto, con dos headers, `traceparent` y `tracestate`, y la mayoría de las bibliotecas de tracing lo siguen. La capacidad de los headers y su propagación difieren según el broker y el puente: algunos limitan el número o el tamaño de los headers, y algunos puentes los descartan, así que comprueba la ruta que de verdad siguen tus mensajes.

El lugar donde se captura el contexto importa más de lo que parece. El Capítulo 4 puso una columna `headers` en el outbox precisamente para esto. El relay se ejecuta más tarde, en otro proceso, sin ningún span ambiental, así que un relay que inyecta el contexto en el momento de publicar adjuntaría el trace propio del relay en lugar del que hizo el pedido. El contexto debe capturarse cuando se escribe la fila del outbox, dentro de la transacción del productor.

```
function writeOutboxEvent(event):
  // runs inside the caller's transaction
  headers = {}
  injectTraceContext(currentSpan(), headers)
  headers.correlation_id = event.correlation_id
  outboxInsert(event, headers)

function relayPublish(row):
  // the relay forwards stored headers verbatim and never
  // recomputes them: a republication must carry the original
  context
  publish(topicFor(row.event_type), row.event, row.headers)
```

Del lado del consumidor hay dos maneras de relacionar el span nuevo con el antiguo. Un span de consumidor puede ser hijo del span del productor, lo que dibuja un único árbol. O puede iniciar un trace nuevo y mantener un **link** al contexto del productor. La elección correcta depende del caso.

Caso	Relación	Por qué	Qué cuesta
Un solo mensaje, consumido de inmediato	Hijo del span del productor	Una sola cascada desde la solicitud hasta el efecto	Si la entrega llega tarde, el trace queda abierto durante todo el retraso y las duraciones engañan
Batch o fan-in	Trace nuevo, un link por productor	Un span tiene un solo padre, así que un consumidor de batch no puede ser hijo de doce productores	El backend debe admitir links, y alguien tiene que navegarlos
Fan-out	Un span de productor, un span de consumidor hijo por suscriptor	El árbol sigue siendo un árbol	El trace crece con cada suscriptor que se añade
Replay, re-drive o entrega tardía	Trace nuevo con un link	Un padre de hace horas estiraría el trace original	La decisión de muestreo se toma de nuevo, así que un trace de productor muestreado puede enlazar con un trace de consumidor no muestreado

Aun así, tomaría los links como opción por defecto, porque el retraso y la reentrega son el caso normal para el que este libro planifica. El precio es la cascada única de extremo a extremo que la mayoría de las interfaces de tracing dibujan para padre-hijo. Para un consumidor que siempre se ejecuta a los pocos instantes de su productor, padre-hijo es la mejor opción y la tomaría.

```
function startConsumerSpan(message):
    producerContext = extractTraceContext(message.headers)
    span = newSpan("consume " + message.event_type)
    if producerContext exists:
        span.addLink(producerContext)
    span.setAttribute("event_id", message.event_id)
    span.setAttribute("correlation_id", message.correlation_id)
    // from a broker attribute where one exists; otherwise from a
    header
    // that the retry path increments on every republish
    span.setAttribute("delivery_attempt", deliveryAttempt(message))
    return span
```

Fíjate en que el span registra `event_id` y el intento de entrega. Los brokers basados en log no llevan una cuenta de entregas, y solo algunos brokers basados en colas la exponen, así que decide de dónde sale el número antes de apoyarte en él. Cuando un mensaje se reentrega, obtienes un segundo span con el mismo `event_id`. El par te dice que el mensaje se entregó dos veces, y el campo `outcome` descrito más abajo te dice si se aplicó dos veces.

El tracing tiene límites, y tendría cuidado de no apoyarme en él para la garantía. Los backends muestrean, así que el trace del único pedido que te importa puede no existir. Un puente, una copia de dead letter o un productor escrito a mano pueden descartar los headers de contexto. Una manera de evitar depender de los headers es llevar el identificador del trace en el cuerpo: el envelope de BabelQueue (babelqueue.com), un estándar abierto y neutral respecto del lenguaje, lleva en el cuerpo del mensaje un `trace_id` que acuña el primer productor, que los productores posteriores copian sin cambios y que sobrevive al paso por dead letters, de modo que el rastro se mantiene aunque un salto descarte los headers. Un trace que termina en una fila del outbox nunca llega a un consumidor que no ha leído la fila. Por eso los identificadores del envelope importan más que la biblioteca de tracing: `correlation_id` debe aparecer en cada línea de log y en cada fila de base de datos que maneja el evento, y así funciona incluso cuando el tracing está apagado.



Lee el diagrama buscando lo que no cambia: el `correlation_id` es el mismo en cada salto, mientras que cada servicio guarda un contexto de trace nuevo en su propio outbox.

Reconstruir la historia de un pedido

Con los identificadores en su sitio, la pregunta “¿dónde está el pedido 81724?” se convierte en una consulta. La implementación más barata es una convención y no un producto: cada servicio escribe una línea de log estructurada por evento manejado y por evento publicado, con `correlation_id`, `event_id`, `event_type`, `causation_id`, resultado y timestamp como campos. Buscar en esos logs `correlation_id = "order-81724"` te da cada salto en cada servicio.

```
log {
  at: "2026-03-14T09:41:09Z", service: "orders",
  action: "consumed", outcome: "applied",
  event_type: "PaymentCaptured", event_id: "3f2a9c1e-...",
  occurred_at: "2026-03-14T09:41:07Z",
  correlation_id: "order-81724", causation_id: "b81d77f0-...",
  delivery_attempt: 1, lag_ms: 1840
}
```

El campo `outcome` merece atención. Usa un conjunto pequeño y cerrado: `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` y `dead_lettered`. El Capítulo 9 sostuvo que un evento obsoleto que se ignora merece dejar un rastro, y el Capítulo 7 cubre la comprobación del inbox que produce `duplicate_skipped`. Una vez que esos resultados son campos, “¿cuántos eventos descartó este consumidor como obsoletos en la última hora?” es un conteo y no una investigación.

Ordenar por `occurred_at` da una línea de tiempo, y seguir `causation_id` da un árbol: `OrderPlaced` causó `PaymentAuthorized`, que junto con `StockReserved` causó `CapturePayment`, y así sucesivamente. En el escenario de la apertura, el árbol habría mostrado `PaymentCaptured` publicado por `Payments` y ninguna línea `consumed` en `Orders`. El primer salto que falta es el lugar donde mirar. Eso es más útil que cualquier línea de log aislada: un hueco en la cadena es evidencia de que algo se perdió, mientras que una cadena larga solo muestra que el flujo avanzó.

Si operas una saga del Capítulo 10, su fila `saga_instance` es una segunda fuente. Responde “¿cuál cree el orquestador que es el estado?”, y los logs responden “¿qué ocurrió en realidad?”. Cuando discrepan, has encontrado algo que merece un ticket.

También conservaría un **event journal** para los flujos más importantes: un registro de solo anexo de envelopes de eventos, cada uno con un hash de su payload en lugar del payload mismo, escrito por cada servicio junto a su inbox. Cuesta almacenamiento y algo de carga de escritura, y a cambio puedes responder preguntas después de que el broker haya olvidado los eventos, incluida la de si un evento reconstruido sigue coincidiendo con el original (consulta la sección de reparación más abajo). Conserva la retención lo bastante larga para que se ejecute tu conciliación más lenta.

Métricas que importan

Las métricas te hablan de la presión antes de que un mensaje cualquiera salga mal. La lista de las útiles en un sistema orientado a eventos es corta, y la mayoría ya se ha insinuado en capítulos anteriores. Empezaría con estas.

Señal	Qué mide	Por qué importa
Retraso del consumidor (mensajes)	Cuánto se ha quedado atrás el consumidor respecto de la cabeza	Tamaño del backlog; útil para la capacidad
Antigüedad del mensaje, del lado del consumidor	Tiempo desde que se produjo el mensaje sin procesar más antiguo	Lo que experimenta el negocio
Antigüedad del outbox	Antigüedad de la fila sin publicar más antigua	Salud del relay (Capítulo 5)
Latencia de extremo a extremo	Tiempo desde el primer <code>occurred_at</code> del flujo hasta el efecto en el consumidor final	El deadline real del flujo
Tasa de retries y conteo de intentos	Proporción de mensajes que necesitan más de un intento	Degradación oculta por los retries (Capítulo 6)
Llegadas a la DLQ y su antigüedad	Dead letters nuevos, y el más antiguo sin revisar	Fallos que necesitan a una persona (Capítulo 8)
Conteos de ignorados y duplicados	Eventos obsoletos descartados, duplicados omitidos	Presión de orden y de reentrega

Señal	Qué mide	Por qué importa
Tasa de drenaje y tiempo para ponerse al día	Tasa de procesamiento frente a tasa de llegada; retraso dividido por la tasa neta de drenaje	Si un backlog se despeja antes de su deadline o del límite de retención (Capítulo 6)

El retraso, y por qué la antigüedad del mensaje es mejor

El **retraso del consumidor** en mensajes es el número que la mayoría de los dashboards muestran primero, y es el más fácil de malinterpretar. Un retraso de 10,000 mensajes significa poco por sí solo: si el consumidor drena 5,000 por segundo, va dos segundos atrasado, y si drena 10 por segundo, va un cuarto de hora atrasado. El retraso en mensajes responde una pregunta de capacidad. El retraso en tiempo responde la pregunta del negocio.

La **antigüedad del mensaje** es el tiempo desde que se produjo el mensaje más antiguo que sigue esperando. Algunos brokers basados en colas exponen directamente la antigüedad del mensaje más antiguo; en los demás, dérivala de un timestamp del mensaje en espera más antiguo. En un broker basado en log, dérivala del timestamp del registro en el offset confirmado del consumidor. En ambos casos el número está en segundos, así que se puede comparar con un deadline que eligió el negocio.

Importa el timestamp del que restas, porque los candidatos miden cosas distintas. El momento de anexión del broker mide cuánto esperó un mensaje en el broker, que es la salud del consumidor. `occurred_at` mide el tiempo desde que ocurrió el hecho, incluido el tiempo que una fila estuvo en el outbox esperando al relay, que es la vista del negocio sobre el retraso. El momento de creación del productor depende de la configuración: según cómo esté configurado, un broker de log guarda este o el momento de anexión, y un relay que vuelve a publicar una fila antigua obtiene un momento de creación nuevo a menos que fije uno de forma explícita. Usaría el momento de anexión para el gauge de antigüedad del consumidor y `occurred_at` para la latencia de extremo a extremo. Usar `occurred_at` para el gauge hace que un consumidor sano parezca muy tardío en cuanto el relay vuelve a publicar una fila antigua. La antigüedad de un grupo es el máximo entre sus particiones, y las alertas deberían usar ese máximo.

Vale la pena nombrar una trampa. Una partición sin tráfico nuevo tiene un retraso de cero y, con un cálculo ingenuo, una antigüedad de cero o una antigüedad que crece para siempre. Decide cómo reporta una partición inactiva, y pruébalo. La misma función debe sobrevivir al fallo de retención descrito más abajo: un offset confirmado que ha quedado por detrás del inicio del log apunta a datos borrados, así que esa comprobación va primero y se reporta por separado. Un consumidor atascado en un poison record, como en el Capítulo 8, muestra un throughput de cero y una antigüedad creciente, y esa combinación debería ser la que envíe un page a alguien.

```
function reportConsumerAge(partition):
    committed = committedOffset(partition)
    head = headOffset(partition)
    if committed >= head:
        // nothing waiting: report zero, and report liveness
        separately
        emitGauge("message_age_seconds", 0, partition)
        return
    logStart = logStartOffset(partition)
    if committed < logStart:
        // retention already deleted unread data: report how much, as
        a
        // gauge that follows the condition and not how often we
        checked
        emitGauge("unread_offsets_lost", logStart - committed,
partition)
        // the gauge clears at the next commit, so also leave a record
        logLostRange(partition, committed, logStart)
        committed = logStart
        record = readRecordAt(partition, committed)
        // append time: how long this record has waited at the broker
        emitGauge("message_age_seconds", now() - appendTime(record),
partition)
```

La retención es el deadline que no ves

La antigüedad del mensaje tiene una segunda función, menos obvia. En un broker basado en log, un consumidor que se retrasa más que el período de retención pierde mensajes en silencio: el broker borra datos que nunca se leyeron, y el consumidor continúa desde donde lo ponga su política de reinicio de offset. En un broker basado en colas ocurre lo mismo cuando los mensajes caducan. El Capítulo 3 nombró este fallo, y no viene acompañado de ningún error.

Esa política de reinicio es una decisión, y cada opción tiene un precio. Reiniciar al offset más temprano disponible hace replay de lo que quede, así que la pérdida es parcial y el consumidor debe absorber una avalancha de mensajes antiguos. Reiniciar al offset más reciente salta hacia adelante y hace silenciosa la pérdida, que es lo que ocurrió en el escenario de apertura. Fallar, de modo que el consumidor se detenga y espere a una persona, convierte una pérdida silenciosa en un incidente. Elegiría fallar para cualquier flujo en el que un hueco cueste dinero, y aceptaría que el consumidor esté caído hasta que alguien decida. El Capítulo 14 cubre cómo recuperarse del hueco una vez que lo has notado: reiniciar a un timestamp y hacer replay, con los efectos secundarios suprimidos.

La defensa es aritmética. Compara la antigüedad del mensaje con el período de retención o de caducidad, y alerta cuando la antigüedad alcance una fracción de este, digamos la mitad. En ese punto el deadline aún está por delante, y una persona tiene tiempo de añadir consumidores, pausar a un productor o ampliar la retención. Una vez pasado el deadline, la única evidencia es la ausencia de eventos, y eso es lo que la conciliación existe para detectar. Pondría la proporción en el mismo dashboard que la antigüedad misma, porque un retraso aceptable para el negocio y fatal para la retención es una situación real.

Para un conector de CDC, la señal equivalente es cuánto del log de la base de datos no ha consumido aún el conector, comparado con cuánto está dispuesta a conservar la base de datos para él. Un conector que está caído durante más tiempo del que la base de datos retiene su log puede necesitar un snapshot nuevo para recuperarse, y eso es una interrupción por sí misma.

Latencia, retries y el resto

La **latencia de extremo a extremo** necesita una definición antes de necesitar un gráfico. La mediría desde el `occurred_at` del primer evento de un flujo hasta el momento en que se confirma el efecto final, por ejemplo de `OrderPlaced` a `OrderConfirmed`. Reporta percentiles, no un promedio: la cola lenta son las personas compradoras que escriben a soporte. El consumidor final ve solo el envelope que consumió, y ese envelope lleva la hora de su propio hecho, no el inicio del flujo. O bien propagas la hora de inicio del flujo en un header, o la buscas por `correlation_id` en el event jour-

nal o en el servicio responsable. Después etiqueta la medición con el `event_type` final. Ambos timestamps vienen de los relojes de servicios distintos, así que espera algo de desfase en el resultado; es pequeño frente a un deadline de negocio, y es una razón para no alertar por diferencias de menos de un segundo.

La **tasa de drenaje** es la forma observable del backpressure. Divide el retraso por la diferencia entre la tasa a la que el consumidor procesa mensajes y la tasa a la que llegan, y obtienes el tiempo para ponerse al día. Si las llegadas superan al procesamiento, el retraso nunca se despeja, y ese es el número sobre el que alertar, mucho antes de que la antigüedad alcance el límite de retención. El Capítulo 3 definió el backpressure y el Capítulo 6 cubre cómo acotar el trabajo que asume un consumidor.

La **tasa de retries y el conteo de intentos** dejan al descubierto el problema que el Capítulo 6 llamó fallos ocultos: una dependencia que lleva semanas degradada mientras los retries la absorben en silencio. El histograma del conteo de intentos es más informativo que el conteo de errores, porque un mensaje que siempre tiene éxito en el tercer intento no produce ningún error.

Las **llegadas a la DLQ** y la antigüedad del dead letter más antiguo sin revisar vienen del Capítulo 8. Suma los **conteos de duplicados y obsoletos** de los capítulos 7 y 9. Un conteo de duplicados que sube justo después de un deploy es normal, ya que la propiedad del trabajo se mueve, y un conteo que sube a una hora estable apunta a handlers lentos. En un broker basado en colas, la causa habitual es un visibility timeout más corto que el tiempo de ejecución del handler. En un broker basado en log, es un rebalanceo, o un consumidor que excede su poll timeout y pierde sus particiones.

Alertas que significan algo

El fallo de la mayoría de las configuraciones de alertas no es que haya pocos datos. Es que los datos envían un page a las personas por causas en lugar de por efectos, y que, tras unas semanas de ruido, nadie confía en el buscapersonas. Exigiría a las alertas un único estándar: si esto salta a las tres de la mañana, una persona con nombre puede decir qué hacer al respecto.

Alerta sobre **síntomas atados a un deadline de negocio**. “La fila sin publicar más antigua del outbox tiene más de dos minutos” es un síntoma de que algo entre un cambio de estado y su anuncio está atascado. “La CPU del consumidor supera el 80 por ciento” es una causa, y puede importar o no. Usa las métricas de causa en dashboards donde alguien investiga, y reserva el page para los síntomas.

Da a cada alerta un **responsable y un umbral con una razón**. El Capítulo 8 hizo el mismo punto para las DLQ: un correo perdido y una captura perdida merecen umbrales distintos, y la diferencia debería quedar escrita junto a la alarma. La razón importa porque los umbrales se desvían, y un número sin una razón declarada se sube la primera vez que molesta.

Alerta sobre la **ausencia además de la presencia**. Un relay que se cae no emite ningún error; no emite nada. Un grupo de consumidores que ha perdido a todos sus miembros muestra retraso, pero un topic que ha dejado de recibir eventos por completo no muestra retraso ni errores. Para los flujos con un ritmo constante, como los pedidos en horario laboral, una alerta como “no se han observado eventos `OrderPlaced` durante 10 minutos” detecta un silencio del lado del productor que todas las demás señales pasan por alto. El umbral tiene que seguir el patrón de tráfico, y para los flujos sin ritmo es la herramienta equivocada.

Por último, dos niveles bastan para la mayoría de los equipos: un page para todo lo que amenace un deadline o pierda datos, y un ticket para la presión lenta, como una DLQ creciente de mensajes de poco valor. Un tercer nivel tiende a ignorarse.

Advertencia: Subir el umbral de una alerta para detener el ruido es como un equipo cambia una molestia por un punto ciego. Si una alerta salta con demasiada frecuencia, arregla su causa o su definición. No subas el número hasta que deje de saltar.

Conciliación

Las métricas y los trazes observan lo que hace el sistema. No pueden decirte nada sobre un evento que nunca entró en el sistema, un handler que confirmó el estado equivocado sin un error, o una reparación que alguien

aplicó a mano. Para eso, comparas el sistema con un registro independiente de lo que debería ser verdad. Eso es la conciliación.

Un job de conciliación toma dos vistas de los mismos hechos y reporta las diferencias. Los pares suelen ser obvios una vez que los buscas:

- **Orders frente a Payments:** todo pedido en estado `PAID` tiene un pago capturado, y todo pago capturado pertenece a un pedido que lo conoce.
- **Payments frente a Ledger:** todo `PaymentCaptured` tiene un `LedgerEntryPosted` correspondiente por el mismo monto en la misma moneda.
- **Tus registros frente a los de un proveedor:** el informe de liquidación del proveedor de pagos frente a tus propios registros de capturas. El Capítulo 7 apuntó aquí para los efectos que no se pueden hacer idempotentes, donde la conciliación es la última línea de defensa.
- **Un modelo de lectura frente a su fuente:** un índice de búsqueda o una vista que el Capítulo 9 dijo que se podía marcar como sospechosa y reparar más tarde.

Tres tipos de diferencia cubren la mayoría de los casos: **faltante** (la fuente lo tiene, el lado derivado no), **obsoleto** (ambos lo tienen pero el lado derivado discrepa, normalmente porque va atrasado) y **sobranante** (el lado derivado tiene algo que la fuente no conoce). Cada uno apunta a una causa distinta. Faltante sugiere eventos perdidos o nunca consumidos, obsoleto sugiere orden o retraso, o de vez en cuando un handler que escribió el valor equivocado, y sobranante sugiere un bug o un cambio manual.

Lo que merece cuidado es que los eventos en tránsito se ven exactamente como discrepancias. Un pago capturado hace cuatro segundos y que aún no está en Ledger es normal. Un job que lo reporte producirá ruido y enseñará a la gente a ignorarlo. Por eso el job compara solo registros más antiguos que un **período de gracia** elegido a partir de tu latencia de extremo a extremo normal, y confirma una discrepancia por segunda vez antes de reportarla.

```

function reconcileCaptures(window_start, window_end):
    upper = min(window_end, now() - gracePeriod)
    suspects = []
    for each batch in readCapturesInWindow(window_start, upper,
batchSize):
        // list per payment_id
        entries = fetchLedgerEntries(keysOf(batch))
        for each capture in batch:
            found = entries[capture.payment_id]
            if count(found) is 0:
                suspects.add({ key: capture.payment_id, kind:
"missing" })
            else if count(found) > 1:
                suspects.add({ key: capture.payment_id, kind: "extra"
})
            else if differs(found[0], capture):
                suspects.add({ key: capture.payment_id, kind: "stale"
})
        for each batch in readLedgerEntriesInWindow(window_start, upper,
batchSize):
            captures = fetchCaptures(keysOf(batch))
            for each entry in batch:
                if captures[entry.payment_id] is null:
                    suspects.add({ key: entry.payment_id, kind: "extra" })
            // a second look after a delay removes events that were merely in
flight
            sleep(recheckDelay)
            confirmed = filter(suspects, stillMismatched)
            for each item in confirmed:
                // upsert on (key, kind): first_seen_at survives, last_seen_at
moves
                upsertDiscrepancy(item.key, item.kind, now())
            // anything open in this window that no longer mismatches is
closed here
            resolveOpenDiscrepanciesInWindow(window_start, upper, except:
confirmed)
            return count(confirmed)

```

Varios detalles de este bloque sostienen la corrección. La ventana está acotada y se aplica el corte, así que el job lee datos que han tenido tiempo de asentarse. Cada batch es una búsqueda por clave en lugar de un join entre bases de datos, porque los servicios de Parcelly no leen las tablas de los demás, así que el job obtiene el otro lado a través de la propia interfaz de ese servicio o de un extracto publicado. El job solo registra discrepancias y no cambia nada. Y como hace un upsert en una tabla de discrepancias con clave por elemento y tipo, ejecutarlo dos veces, o tras una caída a mitad de una ventana, produce la misma tabla, sin una pila creciente de filas duplicadas. La misma función cierra las discrepancias cuando ambos lados coinciden, que es el paso en el que se apoya el código de reparación de

más abajo. Un pago registrado en Ledger dos veces lo detecta la pasada directa, que cuenta las coincidencias. La pasada inversa es la que encuentra una entrada “extra” sin ninguna captura detrás, como un asiento de un pago que nunca se capturó, y una comprobación en una sola dirección jamás lo haría.

Elige la cadencia según el daño que causa una discrepancia. Payments frente a Ledger es financiero y merece una ejecución frecuente, con una comparación completa diaria como respaldo. Un índice de búsqueda puede necesitar una ejecución nocturna. La salud del propio job también importa: alerta si no ha terminado dentro de su ventana esperada, porque una conciliación que se detuvo en silencio es peor que ninguna, ya que la gente cree que está vigilando.

Nota: Un job de conciliación es un consumidor de los datos que verifica, y puede equivocarse. Trata una discrepancia primero como una pista para investigar. Las primeras semanas de un job nuevo se dedican sobre todo a encontrar falsos positivos causados por el timing, el redondeo o una regla de comparación que no coincide con el negocio.

Reparación y backfill

Encontrar una discrepancia es la mitad del trabajo. La otra mitad es cambiar el estado para que coincida, sin crear una segunda discrepancia.

La regla a la que me atendería es que **las reparaciones pasan por la misma ruta que los cambios normales**. Si a Ledger le falta una entrada para un pago capturado, no insertes la fila con una consola de base de datos. El Capítulo 3 trató las acciones de los operadores como parte del modelo de fallos, y una fila editada a mano no emite ningún evento, así que todos los demás consumidores de ese hecho siguen equivocados, y la siguiente conciliación reporta una inconsistencia de tu propia cosecha. En su lugar, vuelve a publicar el hecho, o envía el comando que lo causa, a través del outbox del productor, para que los consumidores lo manejen como manejan todo lo demás.

Una reparación se puede reintentar, interrumpir y ejecutar dos veces por dos ingenieros, y debe seguir siendo segura en cada caso. La idempotencia del Capítulo 7 lo resuelve solo si el evento de reparación lleva el `event_id` **original**. Un evento de reparación con un id nuevo es un evento distinto para todos los consumidores. Si el primer evento solo se retrasó, llega después de la reparación, y Ledger registra el pago dos veces. Por eso la reparación vuelve a publicar el hecho que ya existe: el mismo envelope, con el mismo `event_id` y `causation_id`, y el inbox trata como duplicado la copia que llegue en segundo lugar.

Eso exige que el productor conserve el id original. La manera más simple es guardar `event_id` en el registro de negocio (aquí, la captura) cuando se escribe la fila del outbox, junto con un hash del payload. Si la fila del outbox sigue existiendo, se marca como sin publicar y el relay la envía de nuevo. Si se purgó, el envelope se reconstruye a partir del registro con el id almacenado.

Reconstruir tiene un límite. El registro es el estado actual, y el evento es un hecho del pasado. Si el registro cambió desde entonces, digamos que un reembolso parcial ajustó el monto, un payload reconstruido a partir de él difiere del original aunque lleve el mismo `event_id`. Por eso se guarda el hash: la reparación reconstruye el payload, compara su hash con el almacenado y se detiene a la espera de una persona si difieren. Un registro que no tiene id ni hash almacenados, normalmente uno escrito antes de que existiera el hábito, tiene que recurrir a un id derivado de su clave de negocio y a campos que no pueden cambiar, y entonces el consumidor también debe estar protegido por una restricción única sobre esa clave, como describe el Capítulo 7. Añadiría esa restricción a Ledger de todos modos, como segunda línea de defensa.

Una republicación también **se difunde**. El topic entrega el envelope a todos los suscriptores, no solo a Ledger. Un suscriptor cuyo inbox aún conserva el `event_id` lo omite. Uno cuya fila del inbox se purgó, porque el evento es más antiguo que su ventana de deduplicación (Capítulo 7), lo aplica por segunda vez, y Notifications podría enviar el mismo correo otra vez. Así que hay dos rutas seguras. Si el evento es más reciente que la ventana de deduplicación más corta entre todos los suscriptores, vuelve a publicar a través del outbox del productor como arriba. Si es más antiguo, entrega la reparación solo al consumidor que no la recibió, a través de la

cola o el topic de reparación de ese consumidor, o reiniciando los offsets de ese único grupo, y apóyate en una restricción sobre la clave de negocio como protección. El Capítulo 8 aplica el mismo principio al redrive.

```
function repairMissingLedgerEntry(discrepancy):
    capture = loadCapture(discrepancy.item_key)
    if ageOf(capture.event_id) > shortestDedupWindow():
        // a broadcast could be applied twice where the inbox row is
    gone
        return sendRepairToConsumer("ledger", capture, discrepancy)
    try:
        transaction:
            original = outboxFindByEventId(capture.event_id)
            if original exists:
                // same envelope and event_id; the relay publishes it
            again
                outboxMarkUnpublished(original.id)
            else:
                // purged: rebuild the identical envelope, never a new
            id
                envelope = rebuildOriginalEnvelope(capture)
                if hashOf(envelope.payload) != capture.payload_hash:
                    // the record changed since the event was
            published
                raise RepairNeedsReview(discrepancy.id)
                outboxInsert(envelope)
                recordRepairAttempt(discrepancy.id, actor, now())
                // commit point: the repair is durable and the relay delivers
            it
            on UniqueViolation as e:
                if e.constraint != "outbox_event_id_key":
                    // a different constraint: a real error
                    raise e
                // a concurrent repair inserted the same event_id first
                return
            // resolution is recorded by the next reconciliation, not here
```

Dos detalles sostienen la seguridad. Marcar de nuevo como sin publicar una fila sin publicar es inofensivo si se repite, y la violación de unicidad de una segunda inserción concurrente se trata como “ya manejado” en lugar de como un error. El envelope republicado también conserva su `causation_id` original, de modo que la cadena de causalidad sigue siendo veraz, y el vínculo con la discrepancia vive en el registro de la reparación. El último comentario es el importante. La función de reparación no marca la discrepancia como resuelta. Eso lo hace la siguiente ejecución de la conciliación, cuando ambos lados coinciden. Así se evita que una reparación que se publicó y luego se perdió se reporte como arreglada.

La reparación en bloque, que suele llamarse **backfill**, añade escala. Un backfill de 30,000 eventos hacia un consumidor en vivo compite con el tráfico normal y puede llevar la latencia del flujo más allá de su deadline. Ejecútalo en batches pequeños, con un rate limit y un interruptor de pausa, como recomendó el Capítulo 8 para el redrive, y vigila la antigüedad del mensaje y el retraso del consumidor mientras se ejecuta. En un broker basado en log puede que prefieras un replay deliberado hacia un grupo de consumidores aparte que reconstruya una vista derivada y luego haga el cambio. Es más limpio cuando el consumidor es una proyección, e incorrecto cuando el consumidor tiene efectos secundarios, como enviar correos.

La reparación a través del productor sirve para un puñado de registros. Cuando el daño es más amplio, como un grupo de consumidores que se saltó todo un hueco de retención, una base de datos restaurada o un clúster de brokers perdido, volver a publicar registro por registro deja de ser la herramienta adecuada. El Capítulo 14 cubre la recuperación por replay, los desastres y la migración de brokers.

Toda reparación necesita un registro: quién la ejecutó, qué discrepancias cubrió, qué publicó y cuándo. El rastro de auditoría es como el siguiente ingeniero se entera de que el pico de duplicados de las 14:10 fue un backfill, y es como un regulador o el equipo de finanzas se entera después de que una entrada de Ledger era una corrección.

Invariantes de negocio

La conciliación compara dos sistemas. Una **comprobación de invariantes** compara un sistema con una regla que el negocio da por verdadera. Los invariantes suelen ser más baratos de evaluar y detectan problemas que ningún segundo sistema podría, como un bug que hace que dos sistemas coincidan en la respuesta equivocada.

Para Parcelly, unos pocos ejemplos muestran la forma:

- Ningún pedido está **CONFIRMED** sin un pago capturado.
- Ninguna reserva permanece retenida más allá de su deadline.
- Ninguna instancia de saga permanece en **AWAITING_RESULTS** más allá de su `deadline_at`.

- Todo pago capturado tiene exactamente una entrada en Ledger, ni cero ni dos.
- La suma de las entradas de Ledger de un pedido se equilibra a cero.

Cada una es una consulta que no debería devolver filas. Ejecútala de forma programada y trata cada fila devuelta como una discrepancia con un responsable. El ejemplo de la saga es el que dejó abierto el Capítulo 10: el handler de timeout es la primera protección contra un pedido congelado, y una consulta de invariantes sobre instancias atascadas es la segunda, porque el handler que debería haberse disparado puede haber fallado él mismo.

```
table invariant_violation:
  invariant_name text not null
  item_key       text not null
  first_seen_at  timestamp not null
  last_seen_at   timestamp not null
  resolved_at    timestamp null
  primary key (invariant_name, item_key)
```

Usar como clave el invariante y el elemento significa que una violación que persiste se actualiza, no se duplica, y `first_seen_at` te dice cuánto tiempo lleva abierto un problema. Esa antigüedad es en sí misma una métrica sobre la que vale la pena alertar.

Escribiría los invariantes con las personas de producto y de finanzas y no para ellas. Los ingenieros saben lo que el sistema puede hacer, y no siempre saben lo que el negocio considera un defecto. Una regla como “un reembolso nunca excede la captura” es obvia para finanzas y puede no estar escrita en ninguna parte del código.

Trade-offs y costos

La observabilidad y la conciliación no son gratis, y no funcionan solas.

La telemetría tiene un precio. Los logs estructurados de cada evento manejado multiplican el volumen de logs, y las etiquetas de alta cardinalidad como `order_id` en las métricas pueden abrumar a un backend de métricas. Mantén los identificadores en los logs y los traces, y limita las etique-

tas de las métricas a conjuntos pequeños como servicio, topic y tipo de evento. Muestrea los traces, y acepta que el trace que necesitas puede no existir.

Un event journal duplica datos. Vale la pena en los flujos donde “¿qué ocurrió?” tiene peso legal o financiero. Para un flujo de poco valor, como una notificación cosmética, bastan los logs, y el journal es un costo sin lector.

La conciliación lee datos de producción. Un job de comparación puede cargar la base de datos fuente, y uno descuidado con los batches puede ralentizar el servicio que protege. Ejecútalo contra una réplica donde la tengas, y acota su tamaño de batch y su tasa. También añade una dependencia: si la interfaz del otro lado está caída, el job no puede ejecutarse.

Las alertas cuestan atención. Cada page gasta la credibilidad del siguiente. Un conjunto pequeño de alertas bien asignadas hace más que un conjunto grande que nadie lee.

La reparación tiene su propio riesgo. Un mecanismo de reparación es una vía para cambiar el estado de producción fuera del flujo normal de solicitudes. Sin auditoría, sin aprobaciones para los tipos peligrosos y sin un diseño idempotente, sustituye una clase de problema por otra peor.

Cuándo no construir todo esto: un sistema con un servicio y un consumidor, de bajo riesgo y con flujos cortos no necesita un event journal, tablas de invariantes ni una conciliación nocturna. Los logs estructurados con `correlation_id` y una alerta sobre la antigüedad del mensaje son el mínimo que mantendría a cualquier escala. El resto crece con el costo de un fallo silencioso. Si un evento perdido cuesta una discrepancia en el libro mayor, construye primero la conciliación. Si cuesta un resultado de búsqueda obsoleto, constrúyela al final.

Errores comunes

- **Alertar sobre la salud y no sobre el flujo.** Unos gráficos verdes de CPU y memoria dicen que los procesos están vivos y no dicen nada sobre si los pedidos se completan. Es tentador porque las métricas de infraestructura ya existen. Añade las alertas de síntoma: antigüedad del mensaje, antigüedad del outbox y violaciones de invariantes.

- **Vigilar el retraso solo en mensajes.** Un conteo de mensajes no tiene un deadline asociado. Conviértelo a antigüedad, y compara la antigüedad con el deadline del negocio y con el período de retención.
- **Injectar el contexto de trace en el relay.** El relay no tiene un span ambiental, así que adjunta el equivocado, o ninguno. Captura el contexto cuando se escribe la fila del outbox y reenvíalo sin cambios.
- **Identificadores inconsistentes.** Un servicio registra `order-81724`, otro `81724`. Buscar por correlación falla entonces justo cuando hace falta. Define el formato una sola vez en el diseño de eventos y haz que se cumpla con un helper de logging compartido. Mantén ese único formato también para el tráfico sintético: el Capítulo 15 marca los probes con un header aparte y nunca con un `correlation_id` especial.
- **Usar un solo timestamp para todas las antigüedades.** El momento de creación del productor, el momento de anexión del broker y `occurred_at` miden retrasos distintos. Usa el momento de anexión para el gauge de antigüedad del consumidor y `occurred_at` para la latencia de extremo a extremo.
- **Republicar una reparación a todos los suscriptores.** Un consumidor cuya fila del inbox se purgó aplica el evento otra vez. Comprueba el evento contra la ventana de deduplicación más corta, o entrega la reparación al único consumidor que no la recibió.
- **Conciliar sin un período de gracia.** El job reporta como discrepancia cada evento en tránsito, el equipo aprende a ignorarlo, y el único hallazgo real se pierde en el ruido. Compara solo registros asentados y vuelve a comprobar antes de reportar.
- **Reparar editando filas a mano.** Arregla una tabla, no emite ningún evento y deja equivocados a todos los consumidores posteriores. Repara a través del productor y del outbox para que el cambio viaje como cualquier otro.
- **Una reparación que no es idempotente.** Una segunda ejecución duplica la corrección. Vuelve a publicar con el `event_id` original, para que el inbox reconozca igual una segunda ejecución y un original tardío.
- **Dejar que la conciliación se degrade.** El job se rompe tras un cambio de esquema, nadie lo nota y el equipo cree que está cubierto. Alerta

cuando una ejecución llegue tarde o falle, y revisa de forma programada el backlog de discrepancias.

Ideas clave

- Un flujo repartido entre servicios necesita identificadores que sobrevivan a cada salto: `correlation_id` en cada línea de log y en cada fila, y el contexto de trace guardado en la fila del outbox para que el relay reenvíe el correcto.
- El tracing ayuda, pero se muestrea y puede descartarse, y links frente a padre-hijo es un trade-off que depende del caso, así que los identificadores del envelope deben bastar por sí solos para reconstruir la historia de un pedido.
- La antigüedad del mensaje, medida frente a un deadline de negocio y frente a la retención del broker, es más útil que el retraso en mensajes, y es la señal que te avisa antes de que un consumidor rezagado pierda eventos en silencio; la política de reinicio de offset decide si esa pérdida es silenciosa o ruidosa.
- Alerta sobre síntomas atados a deadlines, da a cada alerta un responsable y una razón declarada para su umbral, e incluye alertas por silencio, porque los fallos que pierden datos a menudo no producen ninguna señal.
- La conciliación encuentra lo que ninguna métrica reporta: compara registros asentados entre dos vistas de la verdad, clasifica las diferencias como faltantes, obsoletas o sobrantes, y las registra sin cambiar nada.
- Las reparaciones deben recorrer la ruta normal, volviendo a publicar el `event_id` original para que un original tardío y una reparación no puedan surtir efecto ambos, con un rastro de auditoría, y una discrepancia se cierra con la siguiente comparación exitosa, no con la reparación misma. Una republicación llega a todos los suscriptores, así que los eventos antiguos van solo al consumidor que no los recibió.
- Las comprobaciones de invariantes de negocio, como que no haya un pedido confirmado sin un pago capturado, detectan defectos que dos sistemas coincidentes ocultarían.

La observación y la conciliación te dicen que el sistema se está comportando mal, y la reparación arregla un puñado de registros. El Capítulo 14 cubre qué hacer cuando el daño es más amplio de lo que una reparación puede alcanzar: recuperarse por replay, sobrevivir a un desastre y pasar a un broker distinto.

Capítulo 14 — Recuperación, desastres y migración de brokers

Imagina un martes en Parcelly. El cluster de broker gestionado que transporta todos los eventos del marketplace deja de estar accesible. La página de estado del proveedor dice que la zona que lo aloja ha fallado y que habrá un cluster de reemplazo disponible en una hora. Cuando lo esté, los topics existirán y estarán vacíos, o contendrán datos hasta algún punto de los últimos minutos. Nadie puede decir exactamente cuál es ese punto.

Los servicios están bien. Orders, Payments, Inventory, Shipping y Ledger tienen cada uno su propia base de datos, y todas están intactas. Cada servicio tiene un relay que sigue reintentando su publicación, y los outboxes se van llenando. Los consumidores están inactivos, esperando a un broker que no está. Mientras tanto, el equipo tiene que decidir qué hacer cuando vuelva, y todas las opciones suenan razonables. Apuntar los consumidores al nuevo cluster y dejarlos correr. Volver a publicar todo desde los outboxes. Restaurar el último backup de algo. Cada opción es correcta para un store e incorrecta para otro.

Este capítulo trata del día después de que un store falla, desaparece o se reemplaza a propósito. Plantea una sola pregunta. Cuando el broker, la base de datos o la región ya no son lo que eran, ¿cuáles de las garantías que construiste siguen en pie y cómo las usas para volver a un estado en el que puedas confiar?

El problema y por qué es difícil

Los capítulos anteriores defendían contra fallos que dejan los stores intactos: un crash entre dos pasos, un consumidor lento, un duplicado, un poison message. Una recuperación es distinta porque lo que falló es la memoria misma. Un store retrocedió en el tiempo, o perdió parte de su historia,

o se está cambiando por otro. Cada defensa del libro depende de que algún store recuerde algo, así que la pregunta pasa a ser qué memoria sobrevivió y cuál no.

Eso es difícil por dos razones. Primero, los stores fallan de forma independiente y en direcciones distintas. Una base de datos restaurada a seis horas atrás y un broker que perdió sus dos últimos minutos discrepan entre sí de un modo que ninguno de los dos puede detectar. Ambos son consistentes consigo mismos. El Capítulo 3 llamó a esto perder un store entero, y señaló que al principio nada lanza un error.

Segundo, el trabajo de recuperación es poco frecuente, y el trabajo poco frecuente se hace mal bajo presión. Los comandos que mueven un grupo de consumidores hacia atrás, los ajustes que impiden que un replay envíe una segunda tanda de correos, el orden en que se reinician los servicios: todas son decisiones baratas de tomar por adelantado y caras de improvisar.

La tesis de este capítulo es la tesis del libro en su forma más desnuda. El broker no es lo que sobrevive. Lo que sobrevive a un cluster perdido, a una base de datos restaurada o a un cambio de producto es la semántica de fallos que diseñaste a su alrededor: el transactional outbox (Capítulo 4) que recuerda lo que nunca se confirmó, el inbox que reconoce lo que ya se hizo, el identificador estable, el contrato y el hábito de comparar dos vistas de la verdad. Si todo eso está en su sitio, el broker es reemplazable. Si no, el broker cargaba con la corrección del sistema y nadie lo sabía.

Recuperar mediante replay

El **replay**, definido en el Capítulo 2, es la relectura deliberada de la historia desde un log de eventos. Se aplica solo a la familia de log, porque una cola borra lo que entrega. El Capítulo 15 usa el replay como técnica de prueba. Aquí es una herramienta de recuperación, y tiene un procedimiento.

La situación a la que se ajusta es la que abrió el Capítulo 13: un grupo de consumidores se quedó atrás, la retención borró mensajes que no había leído y el grupo siguió adelante dejando un hueco. Si los datos aún se con-

servan, la primera recuperación que conviene considerar no es reproducir los eventos uno por uno desde el productor. Es mover la posición del grupo a antes del hueco y leer de nuevo.

Reiniciar un grupo a un punto en el tiempo

La posición de un grupo de consumidores se guarda como un offset por partición. Moverla hacia atrás es una acción administrativa sobre el broker, y la mayoría de los brokers basados en log pueden apuntar a un offset explícito o a una marca de tiempo, que el broker resuelve como el primer offset en o después de ese momento. Prefiero la marca de tiempo. Un offset es un número que significa algo para una partición de un topic en un cluster, y la marca de tiempo es lo que tú sabes: el deploy empezó a las 09:40, así que retrocede hasta las 09:30. Consulta la documentación de tu broker para saber cómo resuelve una marca de tiempo, ya que la hora del productor y la hora de append del broker no siempre son la que usa.

El procedimiento es corto y el orden importa.

1. Detén el grupo. Reiniciar los offsets de un grupo en ejecución o bien falla o bien lo sobrescribe el siguiente commit, según el broker.
2. Registra los offsets actuales antes de cambiar nada. Es la única forma de volver atrás si el reinicio fue un error.
3. Elige el destino con un margen. Retrocede más de lo que crees necesario, porque los duplicados son baratos y un hueco no.
4. Reinicia al destino, vuelve a arrancar el grupo y observa la antigüedad del mensaje sin procesar más antiguo (Capítulo 13) mientras se drena.
5. Sigue observando hasta que el grupo esté de nuevo al día y entonces concilia, porque el reinicio solo repara lo que el log todavía conservaba.

Elegir un margen es seguro solo gracias al inbox (Capítulo 7), que convierte la relectura de los mensajes que el grupo sí procesó en no-ops. Un grupo cuyos consumidores no son idempotentes no debe reiniciarse, diga lo que diga el procedimiento.

Limitar el replay a un grupo de consumidores

Un log permite que 10 grupos lean el mismo topic, y cada uno tiene sus propios offsets. Esa es la propiedad que hace seguro apuntar el reinicio. Reinicia el grupo que se perdió los eventos y ningún otro grupo lo nota. Reinicia el conjunto más pequeño de grupos que se perdieron algo, y nómbrales en el plan antes de ejecutarlo. El consumidor de `OrderPlaced` de `Payments` conserva su posición mientras el grupo de `Inventory` relea el último día.

Una segunda forma de la misma disciplina es un grupo separado para el replay. Si hay que reconstruir una proyección, por ejemplo la vista de búsqueda de `Orders`, arranca un grupo nuevo con otro nombre que lea desde el principio hacia una tabla nueva, y cambia el lector cuando se haya puesto al día. El grupo antiguo nunca se mueve, así que no hay nada que deshacer si la reconstrucción sale mal. Cuesta almacenamiento y un segundo consumidor durante un tiempo, un precio pequeño por un rollback gratuito.

Suprimir los efectos secundarios

Un replay aplica el código de hoy a los hechos de ayer. Si un handler envía correos, llama al proveedor de pagos o notifica a un transportista, el replay hace que lo vuelva a hacer. El Capítulo 12 planteó esto, y la regla que dio es el punto de partida: un consumidor reconstruido por replay tiene que saber que está haciendo un replay.

La primera idea que la gente tiene es una marca en el mensaje, un header que diga “esto es un replay”. No funciona, porque un mensaje reproducido es el mensaje original. El broker conserva los bytes que escribió el productor, y nada indica que se estén leyendo por segunda vez. El modo pertenece al consumidor, no al mensaje. El grupo que hace el replay se ejecuta como un deploy separado con un valor de configuración como `mode: replay`, o bien la capa de efectos comprueba el nombre del grupo.

Con el modo conocido, hay cuatro opciones para los efectos en sí, y cada una tiene un precio.

- **Poner una compuerta a los efectos en el borde.** Cada llamada externa pasa por una única función que comprueba el modo y, en replay, no hace nada. Es el diseño más claro y el más barato de auditar, pero solo

funciona si ningún handler la esquivo. Un handler nuevo que llame directamente a un proveedor la derrota, así que la regla necesita una prueba (Capítulo 15).

- **Poner claves de idempotencia en las llamadas externas.** Si la llamada de captura lleva una clave derivada de `event_id`, el proveedor reconoce el segundo intento. El Capítulo 7 lo trató. Protege solo mientras el proveedor recuerde la clave, así que es una segunda línea de defensa y no aquella en la que apoyarse para un replay del mes pasado.
- **Pausar el outbox de los eventos derivados.** Un handler en replay escribe filas nuevas en su propio outbox en la misma transacción, y esas filas salen hacia el siguiente servicio. Si los nuevos eventos llevan identificadores nuevos, los consumidores posteriores los tratan como hechos nuevos. O bien derivas el identificador de cada evento saliente de forma determinista a partir del `event_id` de entrada y el nombre del handler, de modo que la segunda ejecución produzca los mismos identificadores y los inboxes posteriores los absorban, o bien retienes el relay del servicio que hace el replay hasta haber comprobado qué produjo.
- **Hacer el replay en otro entorno.** Para una proyección sin efectos, una tabla nueva es la respuesta más limpia, como se vio arriba. Para un handler con efectos, un deploy distinto con su capa de efectos conectada a un stub es una forma de probar el replay antes de ejecutarlo de verdad.

Advertencia: Reiniciar un grupo que envía correos, cobra tarjetas o llama a transportistas, sin compuerta y sin claves, convierte cada evento histórico en una acción nueva. El inbox no ayuda si lo reinicias también, y para los efectos externos nunca podría, porque el proveedor no hace rollback junto con tu base de datos. Decide el método de supresión antes de mover los offsets, no después de la primera queja.

La ventana de deduplicación y el inbox

El inbox cambia lo que hace un replay, y esto juega a tu favor o en tu contra según lo que quieras.

Si la base de datos del consumidor está intacta y el rango que estás rele- yendo cae dentro de su ventana de deduplicación (Capítulo 7), el inbox reconoce cada `event_id` que ya ha procesado y lo omite. Eso es lo que quieres después de un hueco: el grupo procesa solo lo que falta, y el resto sale barato. Es también la razón por la que un replay puede reparar al consumidor que nunca vio un evento sin molestar a quienes sí lo vieron.

Si el rango es más antiguo que la ventana, las filas del inbox ya no están. El consumidor aplica por segunda vez los eventos antiguos. Para un consu- midor protegido por una restricción de clave de negocio, como una entra- da de libro mayor por captura, la segunda aplicación falla sin causar daño. Para uno que no la tiene, el efecto ocurre dos veces. Así que la ventana de deduplicación forma parte del plan de recuperación: antes de un replay, indica el evento más antiguo del rango, la ventana más corta entre los grupos que lo verán y qué protege al consumidor donde ambos no se sola- pan. El Capítulo 7 dio la regla de que la ventana debe superar cualquier replay que planees. Esto es el plan.

El inbox también puede ser el obstáculo. Un consumidor con un bug escri- bió un estado erróneo. Corriges el bug y haces el replay, y cada mensaje se omite, porque cada `event_id` está en el inbox. Ese caso aparece en seguida.

Replay, reparación y redrive

Tres herramientas vuelven a mover mensajes, y se distinguen por aquello sobre lo que actúan.

Herra- mienta	Sobre qué actúa	Ca- pítu- lo	Es adecuada cuando
Repara- ción	Un puñado de registros, republica- dos a través del outbox del produc- tor con el <code>event_id</code> original	13	Faltan o son erróneos unos po- cos elementos y el productor puede decir cuáles
Redrive	Dead letters, devueltas al consumi- dor que las rechazó	8	Los mensajes fallaron de forma visible y la causa está corregida
Replay	Un rango del log, releído por un grupo	14	Hay muchas claves afectadas, los datos siguen en retención y el consumidor es idempotente

La pregunta que distingue es quién sabe qué está mal. Con la reparación, el productor o un job de conciliación nombra los elementos concretos. Con redrive, el propio sistema apartó los mensajes. Con replay, nadie tiene una lista: el daño es un rango de tiempo, y la forma más barata de cubrir un rango es leerlo todo de nuevo y dejar que el inbox decida qué ya se hizo. El replay cambia precisión por cobertura.

La retención es un límite de recuperación

El replay solo llega hasta donde el log conserva datos. La retención, un ajuste elegido hace tiempo por el costo del disco, es por tanto también el límite de la recuperación más barata que tienes. Una caída que dura más que la retención, o un bug descubierto después de ella, te saca del camino barato y te lleva a republicar desde los productores y conciliar.

Fijaría la retención en función del tiempo que necesita el equipo para notar y corregir un problema, no solo en función del disco. Si un job de conciliación corre a diario y una corrección tarda un día en desplegarse, una retención de un día no deja nada. La retención también interactúa con el lado de los costos: una retención más larga aumenta el número de esquemas antiguos con los que puede toparse un replay (Capítulo 12) y prolonga el tiempo que los datos personales viven en el log. No sé cuál es el número correcto para tu sistema. Depende de lo rápido que lo notes, de cuánto tarde la reparación y de lo que el log pueda contener.

Un consumidor que se equivocó sin fallar

El Capítulo 3 tiene una fila para un bug de consumidor sin error: el handler tuvo éxito y escribió el resultado equivocado. Esta es la recuperación que más necesita el replay, y aquella en la que el inbox estorba.

Imagina que el consumidor de `PaymentCaptured` de Ledger convirtió importes con el exponente equivocado para una moneda, durante una semana. Cada mensaje recibió su ack. Nada falló, y el inbox contiene cada `event_id`. La conciliación (Capítulo 13) acaba informando de que los totales de Ledger para esa moneda no coinciden con los de Payments. La corrección se despliega, y ahora las filas erróneas existen y hay que corregirlas.

Reproducir el topic hacia el mismo consumidor no hace nada, porque el inbox dice que cada evento está hecho. Hay dos caminos honestos, y la elección depende de qué es Ledger.

Si el estado del consumidor es derivado y se puede tirar, como una proyección, reconstrúyelo. Arranca un grupo nuevo hacia un store nuevo con un inbox vacío, haz el replay desde el primer evento afectado, verifícalo contra la fuente mediante conciliación y cambia. El estado erróneo se descarta, no se corrige.

Si el estado del consumidor es un registro que no se debe reescribir, como en un libro mayor append-only, el replay sigue ayudando, pero no para sobrescribir. Lee el rango afectado con el código corregido en un modo de corrección. Para cada evento, calcula cuál debería haber sido la entrada, compárala con lo contabilizado neto de correcciones anteriores y registra una entrada correctora por la diferencia, con un identificador derivado del `event_id` original y del nombre propio de la corrección, de modo que ejecutar la corrección dos veces sea un no-op. La restricción única sobre el identificador de la propia entrada del libro mayor hace ese trabajo, así que la corrección no necesita una fila de inbox, y el bucle trata una clave duplicada como “ya corregido” y pasa al siguiente evento.

Las entradas erróneas permanecen, como historia. Las correcciones son hechos nuevos que explican por qué. Esto es más lento y más explícito que una reconstrucción, y es la única forma que acepta un auditor.

```

function correctLedgerRange(from_time, to_time, correction_name):
  for each event in readLog("payments.captured", from: from_time,
to: to_time):
    original = ledgerEntryFor(event.event_id)
    posted = original.amount_minor + sumCorrections(original.id)
    // the fixed logic
    should_be = buildEntry(event)
    if posted == should_be.amount_minor:
      continue
    correction_id = deriveId(event.event_id, correction_name)
    try:
      transaction:
        postLedgerEntry({
          id: correction_id,          // unique: re-running
this correction fails
          amount_minor: should_be.amount_minor - posted,
          corrects: original.id
        })
      // commit point: the correction is durable
    on DuplicateKey(constraint: "ledger_entry_id"):
      // an earlier run already made this correction; skip it
      continue

```

El hilo común es que el replay encuentra el rango afectado de forma barata, y la decisión sobre qué hacer con cada evento pertenece a la semántica propia del consumidor.

Escenarios de desastre

Un desastre es un fallo en el que lo que se pierde no es un mensaje sino un store. La forma útil de organizar las posibilidades es según qué store retrocedió en el tiempo. Escribe cada escenario con el patrón que usó el Capítulo 3: qué se rompió, cómo se manifiesta y qué garantía sigue en pie. La última de las tres es la útil, porque nombra aquello en lo que te apoyas, y un flujo que no tiene nada que poner ahí está expuesto en ese escenario. Antes de los escenarios, unas palabras sobre lo que persigues.

Objetivos para cada store

Dos números describen una recuperación. El **objetivo de punto de recuperación (RPO)** es cuántos datos puedes permitirte perder, medido en tiempo: un backup cada seis horas implica un RPO de hasta seis horas. El **objetivo de tiempo de recuperación (RTO)** es cuánto tiempo puedes estar sin el servicio. Ambos son decisiones, y el error habitual es tomar una sola para todo el sistema.

Difieren porque los stores difieren. Para los stores de este capítulo, un esbozo es el siguiente.

Store	Qué cuesta una pérdida	Camino habitual de vuelta	Qué lo determina
La base de datos de un servicio	Estado que no existe en ningún otro sitio, salvo que un evento lo lleve	Backup más log shipping	RPO por base de datos, fijado por el valor de negocio de los datos
El outbox	Eventos aún no publicados	La misma base de datos, así que el mismo RPO	Cada fila perdida es un evento que nunca existió
El inbox	La memoria de lo que se procesó	La misma base de datos que los efectos	Retrocede con ellos (Capítulo 3)
El broker	Eventos en vuelo e historia retenida	Replicación, o republicar desde los productores	El RPO lo fija la replicación, el RTO la rapidez con que los consumidores drenan
Una proyección derivada	Nada, si se puede reconstruir	Replay	El RTO es el tiempo del replay, el RPO es cero si el log lo contiene todo
Un proveedor externo	Efectos fuera de tu control	Sus registros y la conciliación	Fuera de ambos objetivos tuyos

Lo que muestra la tabla es que un RTO para la proyección reproducida es un cálculo de drenaje. Si el log guarda un mes y el grupo puede procesar un día de datos por hora, la reconstrucción tarda unas 30 horas, y ese es su RTO (supón esas cifras; las tuyas salen de la medición, como en el Capítulo 13). Un equipo que nunca ha medido su ritmo de drenaje no conoce su RTO.

También ataría los objetivos al modelo de fallos del Capítulo 3, flujo por flujo. El flujo de captura de pagos tiene una tolerancia distinta de la del flujo de notificaciones, y la tolerancia de Ledger a un hueco está más cerca de cero. Un objetivo uniforme acaba siendo demasiado estricto para las notificaciones y demasiado laxo para el libro mayor.

Restaurar una base de datos desde un backup

Supón que la base de datos de Orders se restaura a un punto seis horas atrás. ¿Qué le hace eso a la maquinaria de este libro?

El **outbox** retrocede con el estado. Las filas escritas en esas seis horas desaparecen, junto con los pedidos que anunciaban, aunque Payments e Inventory puedan haber actuado ya sobre los eventos. Las filas que estaban sin publicar en el punto de restauración pero se publicaron después en la línea temporal antigua vuelven a estar sin publicar, así que el relay las publica una segunda vez. Es un duplicado inofensivo, porque el `event_id` es el mismo.

Hay un peligro más agudo. Si los identificadores de pedido salen de una secuencia, la secuencia restaurada retrocede, y el siguiente pedido recibirá un identificador que ya se dio, a un pedido distinto, en la línea temporal perdida. Payments tiene ahora una captura para `order-81724` que pertenece a una persona compradora que Orders ha olvidado, y el nuevo `order-81724` pertenece a otra persona. Por esta razón preferiría identificadores aleatorios opacos para los agregados. Si debes conservar una secuencia, avanzala más allá del valor más alto jamás emitido antes de aceptar tráfico, un número que puedes leer de los servicios posteriores.

El **inbox** de un consumidor cuya base de datos se restaura retrocede junto con los efectos. El Capítulo 3 señaló que esto es correcto: los mensajes procesados antes del punto de restauración siguen en el inbox y sus efectos sobrevivieron, y los posteriores se olvidan en ambos sitios. Esa propiedad se cumple solo cuando el inbox y los efectos viven en la misma base de datos.

Lo que no retrocede es el **offset** del consumidor, que vive en el broker. El grupo está posicionado en el extremo del log, más allá de seis horas de eventos que la base de datos restaurada ha olvidado. El consumidor no los volverá a ver nunca a menos que alguien mueva el grupo hacia atrás, así que el plan de restauración incluye un reinicio de ese grupo.

El destino requiere cuidado, porque el punto de restauración es una hora del reloj de la base de datos, mientras que un reinicio por marca de tiempo trabaja con las horas de evento del log. En el momento de la restauración el grupo no estaba en el extremo. Estaba en alguna posición, y el evento en esa posición se produjo algún tiempo antes, hasta tanto como la antigüedad de mensaje del grupo (Capítulo 13). Así que el destino es la hora del evento en la posición del grupo en ese momento, menos un mar-

gen. Si el destino se fija solo a partir del punto de restauración, un grupo que iba una hora retrasado se salta la hora que aún no había procesado cuando la base de datos retrocedió.

Lee la posición del historial de antigüedad de mensajes o de los offsets registrados, y guarda ese historial fuera de la base de datos que vas a restaurar. Algunos equipos guardan el offset en la misma base de datos que los efectos, en la misma transacción, de modo que ambos retroceden juntos. Elimina el paso manual y asume la contabilidad de offsets que llevaba el broker, lo cual es un costo que hay que sopesar.

Los efectos externos se quedan donde estaban. La captura que Payments envió al proveedor en la línea temporal perdida existe en los registros del proveedor y no en la base de datos de Payments. Aquí es donde se necesita la conciliación (Capítulo 13). Compara tus registros con los del proveedor y repara las diferencias.

El orden en que se reinicia el sistema importa, y vale la pena escribirlo como procedimiento.

```
function resumeAfterRestore(service, restore_point):
    // 1. nothing moves while the stores disagree
    pauseConsumers(service)
    pauseRelay(service)

    // 2. compare before anything is published or applied
    findings = reconcileAgainstNeighbors(service, since:
restore_point)
    repairThroughNormalPath(findings)

    // 3. start from where each group was, not from when the incident
was
    for each group in consumerGroupsOf(service):
        position_time = eventTimeAtPosition(group, at: restore_point)
        resetToTimestamp(group, position_time - safetyMargin)

    // 4. re-reading derives events and commands again; their ids must
be
    //    deterministic, so the relay stays paused until they are
checked
    resumeConsumers(service)
    verifyDerivedOutboxRows(service)
    resumeRelay(service)

    // 5. a restore is not finished until reconciliation is quiet
    scheduleReconciliation(service, within: "one hour")
```

Los pasos son lo esencial. Pausar primero ambos lados impide que el servicio restaurado empeore su estado inconsistente, y conciliar antes de reanudar sigue el Capítulo 3. El reinicio convierte la restauración en un problema de replay, con una adición.

Un Orders restaurado que relee las horas perdidas produce filas nuevas de outbox, entre ellas un comando `CapturePayment`, y un `command_id` nuevo le parecería nuevo al inbox de Payments y podría capturar dos veces. Por eso los ids de los eventos derivados salen del `event_id` de entrada, como en la sección de efectos secundarios. Un comando disparado por dos eventos, como `CapturePayment`, toma su id de la clave de negocio, por ejemplo `order_id` más el tipo de comando (Capítulo 7), porque cuál de los dos se lee en último lugar puede cambiar en el replay. El relay se retiene hasta que alguien ha comprobado qué se produjo.

Perder datos del broker

Un broker puede perder datos sin perder el cluster. En una familia de log, una réplica que iba retrasada puede ascender a líder, y los mensajes que no tenía se pierden. El nivel de acknowledgment y el número mínimo de réplicas sincronizadas deciden si al productor se le dijo que un mensaje así estaba a salvo (Capítulo 2). Algunos brokers permiten o prohíben por configuración ascender a una réplica desactualizada, y yo averiguaría el valor por defecto del tuyo y lo dejaría por escrito. La retención puede hacer lo mismo en silencio, como mostró el Capítulo 13.

Lo que se ve: posiciones de consumidor que saltan o se reportan fuera de rango, un topic cuyo offset final es menor que antes y, después del hecho, hallazgos de conciliación de tipo “missing”. El acknowledgment del broker es una evidencia con un límite, y el límite es el número de copias que existían en ese momento.

Lo que sobrevive: el outbox, si conservaste las filas. Un relay que purga las filas justo después de publicar no tiene memoria de lo que envió. Uno que conserva las filas publicadas durante un período, digamos unos pocos días a modo de ilustración, puede marcar como no publicadas las filas de la ventana afectada y enviarlas de nuevo. La republicación conserva el `event_id` original, así que los consumidores que sí recibieron un evento tratan la segunda copia como un duplicado. Es el mismo paso que la reparación del Capítulo 13, aplicado a un rango, y lleva la misma precaución

de fan-out: los eventos más antiguos que la ventana de deduplicación de un suscriptor necesitan entregarse al único consumidor que se los perdió, no a todos.

Si las filas del outbox ya no están, los propios registros del productor son la siguiente fuente: reconstruye el envelope a partir del registro de negocio con su `event_id` y su hash de payload almacenados, y detente para pedir intervención humana donde el hash difiera. Y si no existe ninguno de los dos, el evento no se puede recuperar y la conciliación seguirá informando de él. En ese punto la pregunta correcta es de negocio: ¿se reconstruye el hecho que falta a partir de otro sistema, o se acepta como perdido y se explica? Diseñar de modo que esta pregunta rara vez se plantee es lo que se compra con conservar las filas del outbox durante un tiempo.

Perder una región

La pérdida de una región combina las dos anteriores y añade un problema propio. Supón que Parcelly ejecuta un cluster en espera en una segunda región, alimentado por una herramienta de replicación que copia los topics de forma continua, y que las bases de datos se replican de forma asíncrona.

La primera propiedad que hay que enunciar es que **la replicación asíncrona tiene un RPO**, y es el retraso de replicación en el momento del fallo. Todo lo que se había escrito en la región primaria y aún no se había copiado se ha perdido desde el punto de vista del standby. Esa es la medida de lo que pierdes, y es un número que varía minuto a minuto. También merece una alerta, contra un límite declarado, de la manera en que el Capítulo 13 trata la antigüedad de los mensajes: estás vigilando tu RPO.

La segunda propiedad es que **los offsets no viajan**. El standby tiene una copia de los mensajes, pero la copia tiene sus propios offsets, asignados cuando la réplica la escribió. La posición confirmada de un consumidor en la primaria no dice nada sobre el standby. Algunas herramientas de replicación ofrecen traducción de offsets; yo comprobaría qué garantiza la tuya, y no diseñaría una recuperación que dependa de que sea exacta.

El plan portable es arrancar cada grupo en el standby desde la hora del evento en la posición que había alcanzado cuando falló la primaria, menos un margen, y dejar que el inbox absorba el solapamiento. Es la regla de la

restauración de la base de datos. Un grupo que iba 40 minutos retrasado arranca 40 minutos atrás. Los eventos que nunca se replicaron no están en el standby en absoluto, y la ventana de outbox de más abajo los cubre.

Como la información de commit de la propia primaria se pierde con la región, el historial de antigüedad de mensajes tiene que guardarse fuera de ella. Si ya no está, recurre al punto más antiguo dentro de las ventanas de deduplicación de los consumidores y acepta los duplicados adicionales. Un umbral de alerta no basta, porque la alerta puede haber estado disparándose en ese momento.

La tercera propiedad es que **los duplicados en el failover son ciertos, no posibles**. Un consumidor que se reinicia antes de su posición real procesará eventos dos veces. Un relay que estaba a mitad de una publicación cuando la región falló puede publicar una segunda vez desde la base de datos superviviente.

Los huecos son el otro lado: eventos que la primaria confirmó y el standby nunca recibió. Si la base de datos del productor sobrevivió y es más reciente que el broker del standby, contiene filas marcadas como publicadas que el standby nunca vio. Para ellas, amplía la ventana de republicación del outbox como en el caso de pérdida del broker: marca como no publicadas las filas del último intervalo de retraso de replicación, más un margen, y envíalas de nuevo. El retraso de replicación dimensiona esta ventana, y solo esta ventana. Los duplicados no cuestan nada en el consumidor. Los huecos cuestan lo que intentas proteger.

Si la base de datos del productor se pierde ella misma con la región y se restaura desde una réplica más antigua, el caso se reduce a restaurar desde un backup, con las secuencias retrocedidas y los pedidos olvidados que eso conlleva.



Lee el diagrama desde la izquierda. Los dos pasos que tratan con lo desconocido, la ventana de republicación y el margen de la marca de tiempo, yerran ambos hacia la duplicación. Cada paso que sigue existe para hacer inofensiva la duplicación.

Migración de brokers

Una migración es un desastre que programas. Un equipo pasa de un broker a otro por costo, por el fin del soporte, por una oferta gestionada o por un cambio en el patrón de acceso, y el cambio tiene los mismos ingredientes que los desastres anteriores: datos en un lado, un lugar nuevo y vacío en el otro, posiciones que no se trasladan y un período en que dos sistemas están vivos a la vez. La diferencia es que tienes tiempo para ser cuidadoso y ninguna emergencia que te obligue a improvisar.

Las herramientas del libro hacen de una migración un cambio controlado. La razón es que varias de ellas se colocaron a propósito entre la aplicación y el broker.

El relay como único punto de conmutación

Con un transaccional outbox (Capítulo 4), un servicio productor nunca habla con el broker. Escribe en su propia base de datos, y el relay (Capítulo 5) mueve las filas al broker. Así que la elección del broker está codificada en un solo proceso. Cambiar adónde van los eventos significa cambiar adónde publica el relay, y el código del productor no cambia. Es un cambio menor que uno repartido por todos los servicios, un beneficio secundario de un patrón que el Capítulo 4 recomienda por otra razón.

El relay también es responsable del envelope. Toma el `event_id` de la fila del outbox y lo pone en el mensaje que publica. En el nuevo broker el mismo evento tiene el mismo `event_id`, y esa igualdad es lo que hace seguro el resto de la migración.

Dual-publish

El primer paso más seguro es que el relay publique cada fila en ambos brokers. Durante un tiempo, cada evento existe en el broker antiguo, donde lo leen los consumidores de hoy, y en el nuevo, donde todavía no lo lee nadie.

Hacer dual-publish es en sí mismo un dual write, y el Capítulo 4 explica por qué son peligrosos. Aquí el riesgo es menor, porque la fila del outbox es la fuente de verdad y un crash deja la fila para volver a enviarse.

La cuestión de diseño es qué espera el relay. Si marca una fila como publicada solo después de que ambas publicaciones tengan éxito, un shadow broker inaccesible deja filas sin publicar y detiene el flujo en vivo del que dependen los consumidores, por el bien de un broker que nadie lee. Así que lleva el seguimiento de los dos destinos por separado. La fila está publicada, en el sentido del Capítulo 5, cuando el broker antiguo la tiene, y una segunda marca registra el nuevo broker. Una pasada, en orden de `id`, alimenta al nuevo broker con las filas que carecen de la segunda marca, lo que mantiene el orden por clave, y la purga espera a ambos. El costo es una columna más y una pasada más, y a cambio una caída del nuevo broker aparece como un hueco creciente en la comparación y no como una caída de Parcelly.

Durante esta fase el broker antiguo sigue siendo aquel en el que confían los consumidores. El nuevo broker es un shadow, y su único trabajo es compararse con el original.

Mirroring y bridges

Una alternativa al dual-publish en el relay es copiar mensajes entre brokers con una herramienta: una herramienta de mirroring para log a log, o un bridge o conector para log a cola y cola a log. Esto no necesita cambios en el relay y además transporta la historia que existía antes de que empezara la migración, cosa que el dual-publish no hace.

Dos precauciones. Un mirror o un bridge es a la vez consumidor y productor, así que es at-least-once y produce duplicados, como todo par consumidor-productor. Y debe pasar el `event_id` sin cambios, en el cuerpo del mensaje o en un header. Un bridge que acuña su propio identificador convierte cada evento copiado en un evento nuevo, y el inbox no puede reconocer ninguno. Prueba esto con una muestra antes de confiar en ello. Si un mirror conserva el particionado, las claves y los headers, y en qué orden entre particiones, varía según la herramienta, y la respuesta está en su documentación, no en este libro.

Los offsets no se traducen

El offset de un grupo de consumidores en el broker antiguo no significa nada en el nuevo. Incluso entre dos brokers del mismo tipo, la copia tiene su propia numeración. Entre una cola y un log no hay ningún offset en

uno de los lados.

La respuesta es la que alcanzó la sección sobre la pérdida de una región: elegir un punto de partida seguro y apoyarse en la idempotencia. Para cada grupo, elige una marca de tiempo tal que cada evento que el grupo aún no había terminado de procesar en el broker antiguo se publicara después de ella. Toma el retraso del grupo en el momento del cambio, conviértelo a tiempo y añade un margen. Arranca el nuevo grupo en esa marca de tiempo en el nuevo broker. Relee eventos que el grupo antiguo ya manejó, y el inbox los omite.

La ventana de deduplicación se aplica aquí igual que a un replay: el evento más antiguo que releerá el nuevo grupo debe estar dentro de la ventana de cada consumidor.

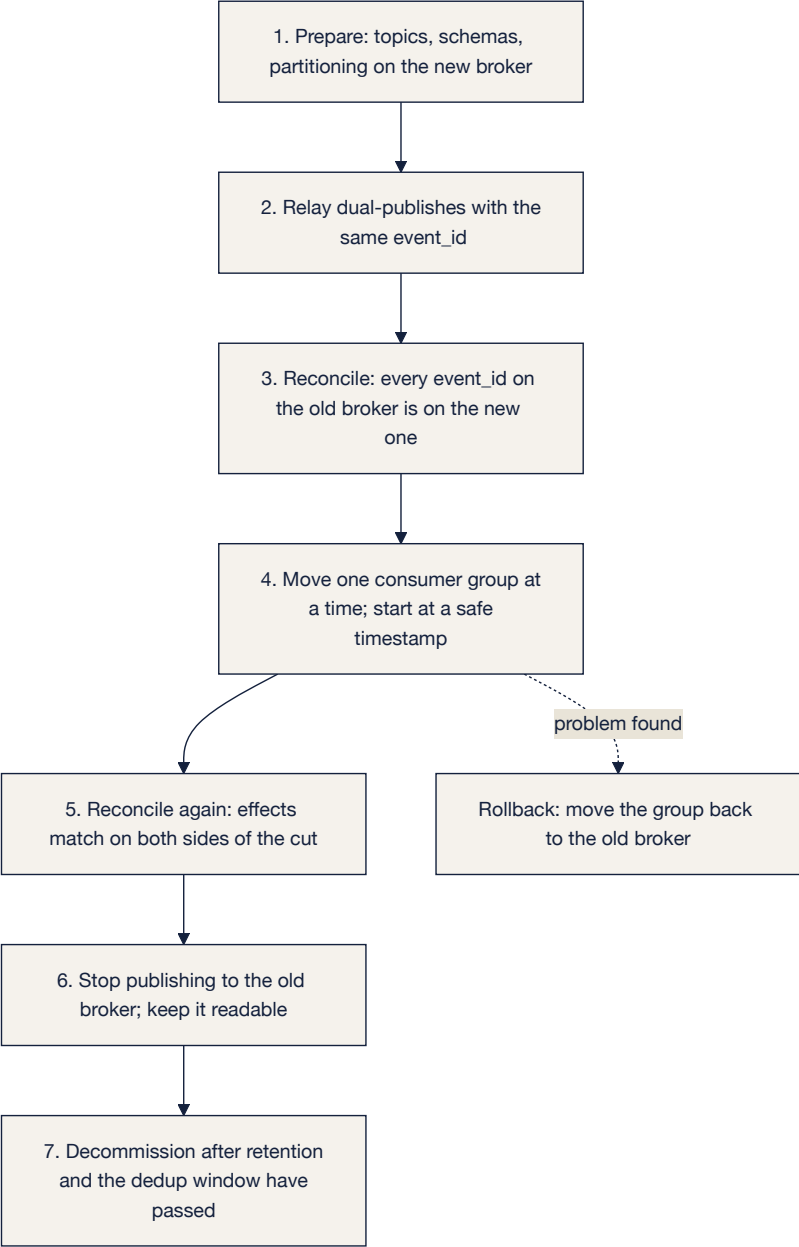
Orden y particiones

El Capítulo 9 dio la condición para el orden por clave: misma clave, mismo particionador, mismo número de particiones. Una migración cambia al menos una de las tres. El nuevo broker puede usar un particionador distinto, de modo que un `order_id` caiga en una partición en la que nunca había caído, y el nuevo topic puede crearse con un número distinto de particiones porque ese es el momento en que la gente reconsidera el número.

De ahí se siguen dos cosas. Calcula la partición en el relay a partir del `partition_key` almacenado, como sugirió el Capítulo 9, para que un solo fragmento de código sea responsable del mapeo, y prueba que los eventos de una clave caen en una partición en el nuevo broker. Y planifica el cutover de modo que ninguna clave tenga eventos en vuelo en ambos sitios. Si un consumidor empieza a leer el nuevo topic mientras el topic antiguo aún contiene eventos sin procesar para el mismo `order_id`, los eventos nuevos pueden manejarse antes que los antiguos: `OrderCancelled` antes de `OrderConfirmed` otra vez, el fallo del Capítulo 9. Hay dos formas de evitarlo. Drenar el backlog antiguo hasta cero antes de arrancar el nuevo grupo, lo que pausa el procesamiento durante lo que dure el drenaje. O hacer que los consumidores toleren el problema por diseño, con eventos que llevan estado y una versión de agregado que rechaza todo lo más antiguo, como describió el Capítulo 9. Drenaría primero en cualquier flujo en el que el orden importe y la pausa sea asumible.

El cutover

El orden de las operaciones es la parte por la que la gente pregunta: ¿primero los consumidores o primero los productores? Con dual-publish la pregunta casi se disuelve, porque los productores ya escriben en ambos antes de que se mueva un solo consumidor. El orden pasa a ser este.



La conciliación enmarca el movimiento. Antes de él, compara el conjunto de valores de `event_id` en los dos brokers durante una ventana: cualquier cosa que esté en el broker antiguo y falte en el nuevo es un defecto del dual-publish, encontrado mientras nadie depende del nuevo broker. Después, compara los efectos: un grupo movido a las 14:00 debería haber producido el mismo estado que habría producido en el broker antiguo, y la conciliación entre servicios del Capítulo 13 es lo que lo verifica. Una migración que se declara terminada porque los consumidores están sanos solo ha verificado que nada se cayó.

Mueve los grupos de consumidores de uno en uno, empezando por el menos crítico, digamos Notifications, y terminando por Ledger. Cada grupo es un paso pequeño y reversible por separado.

Rollback

Una migración sin plan de rollback es una puerta de un solo sentido. Con este diseño, el plan sale casi gratis, mientras se cumpla una condición: el broker antiguo sigue recibiendo todos los eventos. Mientras el relay hace dual-publish, el broker antiguo está completo, así que hacer rollback significa apuntar el grupo de nuevo al broker antiguo en una marca de tiempo con un margen. El inbox absorbe los eventos que el grupo manejó en el lugar nuevo.

La condición es la razón por la que el paso seis llega tarde. Una vez que el relay deja de escribir en el broker antiguo, un rollback necesita que el flujo se invierta, con el relay escribiendo de nuevo en el antiguo y los consumidores del broker antiguo arrancando desde el punto en que se detuvo. Eso sigue siendo posible porque el outbox conserva las filas, y es más lento. El rollback también está acotado por la retención del broker antiguo y por la ventana de deduplicación, así que yo desmantelaría solo cuando ambas hayan pasado desde el último evento que recibió. Retirar el broker antiguo el día en que se mueve el último consumidor elimina el rollback justo en el momento en que más probable es que aparezcan los problemas del nuevo broker.

Entre cola y log

Cuando la migración cruza familias, la semántica cambia, y un plan que la trate como un cambio de dirección es erróneo.

Aspecto	De cola a log	De log a cola
Historia	Se gana: el replay se hace posible (y la retención pasa a ser una decisión)	Se pierde: no hay replay, así que la recuperación depende del outbox y de los productores
Acknowledgment	El ack por mensaje pasa a ser un commit de offset; un mensaje lento ahora puede bloquear su partición (Capítulo 9)	Un commit de offset pasa a ser un ack por mensaje, con visibility timeouts y reentrega
Orden	El orden de grupo, donde la cola lo ofrece, pasa a ser orden por partición; elige la clave	El orden por partición hay que recrearlo, por ejemplo con grupos de mensajes donde la cola los admita
Dead letters	El redrive es algo que construyes o configuras; el log no tiene una dead-letter queue (DLQ) integrada, solo un dead-letter topic que creas tú (Capítulo 8)	Aparecen las funciones de DLQ de la cola, y sus límites y su retención son distintos
Retrasos y expiración	La entrega retrasada y la expiración por mensaje pueden no existir y necesitar un sustituto	El retraso y la expiración pueden ser nativos; desaparece la recuperación basada en retención

Cada fila es un lugar en el que un comportamiento del que dependían tus consumidores puede cambiar en silencio. Escribiría la lista de comportamientos de los que depende cada consumidor antes de elegir el destino, y probaría los que importan, como un poison message que no debe bloquear su partición, en un ensayo (Capítulo 15).

Ensayar la recuperación

Nada de esto vale mucho si la primera vez que lo ejecutas es bajo presión. Una restauración que nunca se ha hecho es una creencia. Un procedimiento de replay que nunca se ha ejecutado con un volumen realista no tiene una duración.

Dos ensayos merecen su costo. Una **prueba de restauración** toma un backup, lo restaura en un entorno desechable, comprueba qué le hace la restauración al outbox, al inbox y a los offsets, y después ejecuta el procedimiento de reanudación y lo cronometra. Un **game day** retira algo a propósito en un lugar seguro, un broker, una zona, la base de datos de un con-

sumidor, y hace que el equipo se recupere usando solo el runbook. El Capítulo 15 explica cómo diseñar estos ejercicios. La razón pertenece aquí: solo ejecutar la recuperación muestra qué pasos olvidaste.

Un runbook para los escenarios de este capítulo tiene que responder una lista corta de preguntas, escritas y no guardadas en la cabeza de alguien. El Capítulo 16 las incorpora a su guía de runbooks.

- ¿Qué store está afectado y cuál retrocedió en el tiempo?
- ¿Cuál es el punto de recuperación y cómo se determinó?
- ¿Qué grupos de consumidores hay que reiniciar, a qué marca de tiempo y con qué margen?
- ¿Qué efectos hay que suprimir durante el replay y con qué mecanismo?
- ¿Cada rango que se va a releer está dentro de la ventana de deduplicación de cada consumidor afectado?
- ¿Qué se pausa, en qué orden y quién lo reanuda?
- ¿Qué conciliaciones confirman que la recuperación funcionó y quién las lee?
- ¿Cómo se deshace todo si va mal?

Trade-offs y costos

La retención es un presupuesto de recuperación. Una retención más larga mantiene disponible durante más tiempo la recuperación barata, con los costos enumerados antes. Una retención más corta te empuja a republicar desde los productores.

Conservar filas del outbox cuesta almacenamiento y un job de limpieza. Las mismas filas que hacen posible una ventana de republicación son las que un diseño más ligero borra de inmediato. Con change data capture (CDC) y borrado inmediato, la ventana no existe (Capítulo 5). Si quieres la seguridad, conserva las filas publicadas durante un período declarado y monitorea la tabla.

El replay es un instrumento contundente. Relee todo lo del rango, incluido lo que estaba bien, y depende de que cada consumidor sea idempotente y de que cada efecto esté bajo compuerta. Para unos pocos registros

conocidos, la reparación es más barata y más segura. El replay se gana su lugar cuando el daño es amplio y nadie tiene una lista.

La supresión de efectos secundarios es código que hay que probar.

Una compuerta que existe pero que un handler nuevo esquivaba es peor que ninguna, porque genera confianza. El costo es una prueba permanente y una regla en la revisión de código.

Una migración con dual-publish cuesta el doble. Dos brokers funcionan a la vez, el relay hace más trabajo y sigue dos destinos, y los duplicados en el lado antiguo aumentan. A cambio obtienes verificación antes de que nadie dependa del nuevo broker, y un rollback. Si el sistema tolera una ventana de mantenimiento, un cutover directo es más barato: detén el relay, drena los consumidores antiguos, apunta el relay al nuevo broker y arranca allí los consumidores. Con un outbox eso necesita una interrupción y ninguna pérdida, porque las filas esperan. Renuncia a la verificación y al rollback, y yo no lo elegiría para el libro mayor.

Cuándo no construir todo esto. Un flujo en el que un hueco sale barato, como un feed interno de métricas, no necesita una ventana de republicación, un runbook ni un game day. Ajusta la recuperación al costo del fallo y deja por escrito qué flujos decidiste dejar sin cubrir. Que no figuren en el plan debería ser una decisión.

Errores comunes

- **Reiniciar un grupo sin comprobar sus efectos secundarios.** Es tentador porque el reinicio es un solo comando y los datos están ahí. El resultado es una segunda tanda de correos y un segundo intento de cobro. Decide cómo se suprimen los efectos antes de mover ningún offset.
- **Reiniciar demasiados grupos.** Hacer que todos empiecen de nuevo parece más minucioso. Cada grupo que reinicias relee la historia y lleva el riesgo de sus propios efectos. Reinicia solo los grupos que se perdieron algo.
- **Confiar en un header de replay.** Es tentador porque una marca en el mensaje parece la señal más barata. Un mensaje reproducido es el mensaje original y no lleva ninguna señal de que se esté leyendo de

nuevo. El modo pertenece al deploy del consumidor o a la capa de efectos.

- **Hacer un replay hacia un consumidor con el inbox del bug.** Tras un bug de consumidor, el inbox omite cada mensaje por hecho. Reconstruye un estado derivado en un store nuevo, o registra correcciones explícitas para uno append-only.
- **Un solo objetivo de recuperación para todo el sistema.** Es tentador porque es un único número que acordar. Es demasiado estricto para las notificaciones y demasiado laxo para Ledger. Fija objetivos por flujo y por store.
- **Restaurar una base de datos y reanudar de inmediato.** El servicio está sano y los dashboards están en verde, así que se reanuda el tráfico. El servicio restaurado publica entonces estado retrocedido, reutiliza identificadores y se pierde eventos que sus offsets se saltaron. Pausa, concilia, reinicia los offsets y solo entonces reanuda.
- **Reutilizar identificadores basados en secuencias tras una restauración.** Las secuencias son lo habitual y nada parece mal hasta que dos pedidos comparten un identificador. Una secuencia retrocedida reparte un identificador que ya pertenece a otro pedido. Usa identificadores opacos, o avanza la secuencia más allá del valor más alto jamás emitido.
- **Confiar en la traducción de offsets.** Es tentador porque la herramienta lo anuncia y ahorraría la aritmética de las marcas de tiempo. Una herramienta de replicación que copia mensajes puede o no mapear los offsets con exactitud. Planifica una recuperación que funcione sin ella, usando marcas de tiempo y el inbox.
- **Un bridge que acuña identificadores nuevos.** A menudo un bridge se configura para envolver cada mensaje en su propio envelope, y eso pasa una prueba con un solo consumidor. Los eventos copiados parecen nuevos, así que el inbox no reconoce nada y cada duplicado se convierte en un efecto. Pasa el `event_id` sin cambios y pruébalo.
- **Desmantelar el broker antiguo cuando se mueve el último consumidor.** Parece que la migración ha terminado. Elimina el rollback justo cuando es más probable que se necesite. Espera a que pasen la retención y la ventana de deduplicación.

- **No restaurar nunca un backup.** El job de backup está en verde. La primera restauración es entonces la de verdad. Restaura en un entorno desechable de forma periódica y cronométrala.

Ideas clave

- Cuando un store se pierde, se restaura o se reemplaza, lo que sobrevive es la semántica de fallos a su alrededor, es decir, el outbox, el inbox, el `event_id` estable, el contrato y la conciliación, y el propio broker es reemplazable.
- El replay es una herramienta de recuperación para la familia de log: detén un grupo de consumidores, registra sus offsets, reinícialo a una marca de tiempo medida desde donde el grupo estaba realmente, con un margen, y deja que el inbox absorba los duplicados, siempre que los datos sigan dentro de la retención.
- Un mensaje reproducido se parece al original, así que el consumidor que hace el replay debe conocer su modo y poner compuerta a sus efectos, derivar los identificadores posteriores de forma determinista o retener su outbox, y no confiar nunca en un header.
- La ventana de deduplicación forma parte de todo plan de recuperación, porque un rango más antiguo que la ventana readmite duplicados, y un bug de consumidor convierte el inbox de una protección en un obstáculo que exige una reconstrucción o correcciones explícitas.
- Los objetivos de punto y de tiempo de recuperación pertenecen a cada store y a cada flujo, y una base de datos restaurada hace retroceder su outbox, su inbox y sus secuencias mientras los offsets se quedan donde el broker los dejó, así que el procedimiento de reanudación pausa, concilia, reinicia los offsets y solo entonces reanuda.
- La replicación asíncrona entre regiones tiene un RPO igual a su retraso y transporta datos pero no offsets de consumidor, así que un failover arranca cada grupo desde la hora del evento de su propia posición en el momento del fallo, dimensiona una ventana de republicación del outbox según el retraso de replicación y trata los duplicados como ciertos y los huecos como el riesgo real.
- Una migración de broker usa el relay como único punto de conmutación, hace dual-publish con el mismo `event_id`, mueve un grupo de consumidores cada vez desde una marca de tiempo segura, concilia

antes y después, y mantiene legible el broker antiguo hasta que el roll-back deje de ser necesario.

Un procedimiento de recuperación que nunca se ha ejecutado es una hipótesis, y también lo es el código que se supone que hace que un sistema sobreviva al fallo. El Capítulo 15 explica cómo probar sistemas orientados a eventos, incluido cómo ensayar las restauraciones, los failovers y los replays que describió este capítulo.

Capítulo 15 — Cómo probar sistemas orientados a eventos

Imagina al equipo de Inventory de Parcelly el viernes anterior a un fin de semana largo. Su suite está en verde: 400 pruebas, unos minutos de ejecución, una cobertura sobre la que nadie discute. El handler de `OrderPlaced` tiene una prueba que construye un evento, llama al handler y comprueba que el stock bajó en dos. Pasa siempre. Las pruebas usan un sustituto en memoria del broker, que entrega cada mensaje una vez, en orden e inmediatamente.

El sábado un deploy reinicia los consumidores. Los handlers van más lentos de lo habitual mientras se calientan, y uno supera su `visibility timeout`, de modo que el broker entrega el mismo mensaje `OrderPlaced` a un segundo consumidor mientras el primero aún lo está procesando. Los dos leen el inbox, ninguno encuentra nada, y los dos siguen adelante y reservan el stock. La restricción única que habría hecho fallar a uno de ellos faltaba en el esquema real, y ninguna prueba lo notó, porque las pruebas se ejecutaban sobre una base de datos fake escrita a mano que nunca aplicaba restricciones y nunca ejecutaba dos handlers a la vez. La suite nunca se había equivocado en nada de lo que comprobaba. Nunca comprobó lo único que importaba.

Mientras tanto, una prueba de la suite de Orders falla una vez de cada 20 ejecuciones. El equipo ha dejado de mirarla. Espera dos segundos a que llegue un evento y a veces el evento tarda tres.

Las dos historias tienen la misma raíz: las pruebas describen un sistema que no existe. Este capítulo responde a una pregunta práctica. ¿Cómo pruebas código cuyos peores bugs aparecen solo bajo duplicación, retraso y reordenamiento, sin construir una suite lenta, flaky o ignorada?

El problema y por qué es difícil

El Capítulo 1 sostuvo que el happy path es el que puedes probar sin esfuerzo y el que te inclinas a creer. Los capítulos 2 a 14 construyeron después defensas contra fallos que el happy path nunca muestra. Las defensas son código, y el código que nunca se ejercita bajo el fallo que maneja es una esperanza. El Capítulo 3 introdujo un modelo de fallos con un campo “verified by” para cada entrada. Este capítulo es donde se rellena ese campo.

Tres propiedades hacen difícil esta tarea de probar.

Los fallos son entrelazados, no entradas. La prueba de una función aporta una entrada y comprueba una salida. El bug de la reserva duplicada necesita un momento concreto: dos entregas que se solapan en el tiempo, o un commit seguido de un crash y una reentrega. Ningún valor de entrada lo dispara. La prueba tiene que controlar cuándo ocurren las cosas, lo que significa que el arnés de pruebas debe ser responsable del comportamiento del broker, del reloj y de los puntos de crash.

La asincronía esconde el momento de la verdad. Después de que un handler se ejecuta, el efecto puede llegar a otro servicio segundos más tarde. Una prueba no puede hacer la aserción de inmediato, y no puede saber cuánto esperar. Dormir un tiempo fijo está mal en ambas direcciones: si es muy corto la prueba es flaky, y si es muy largo la suite es lenta.

Los fakes mienten justo donde importa. La idempotencia descansa sobre una restricción única. La atomicidad descansa sobre una transacción. El orden descansa sobre la asignación de particiones. Un fake que se salta esto te da una suite en verde y ninguna información. El costo del realismo es la velocidad, así que el oficio consiste en decidir dónde pagarlo.

Adaptar la pirámide de pruebas

La pirámide de pruebas clásica sigue valiendo: muchas pruebas rápidas en la base, pocas lentas en la cima. Lo que cambia es para qué sirve cada capa y de qué fallo es responsable de detectar cada una.

Capa	Qué se ejecuta	Real o fake	Qué detecta
Pruebas de handler	Un handler contra una base de datos real	Base de datos real, sin broker	Efecto erróneo, inbox ausente, atomicidad rota, mal manejo de duplicados y de orden
Pruebas de contrato	Las formas de los eventos entre productor y consumidor	Ningún servicio	Cambios de esquema que rompen a un consumidor
Pruebas de componente	Un servicio con un broker, outbox y relay reales	Broker real, otros servicios con stubs	Conexionado: del outbox al relay y al topic, acks, retries, dead-lettering
Pruebas de flujo	Unos pocos servicios juntos, con un fallo inyectado	Todo real, a pequeña escala	Comportamiento entre servicios, como compensaciones y timeouts
Probes de producción	Eventos sintéticos en el sistema en vivo	Real	Problemas de deploy y de configuración que ningún entorno de preproducción muestra

Dos decisiones de la tabla merecen una frase cada una. La capa de handlers usa un motor de base de datos real, porque el inbox, la escritura del outbox y la comprobación de versión dependen de restricciones y transacciones que un fake tendría que reimplementar, y reimplementarlas es como se dejó engañar la suite de Inventory. El broker, en cambio, se deja fuera en esa capa. Un handler es una función que va de un mensaje y un estado a un estado nuevo y algunos mensajes salientes. Si puedes llamarlo directamente, puedes probarlo rápido.

La forma de la pirámide también se sigue de dónde viven los fallos. La mayoría de las defensas de este libro son lógica del lado del consumidor o del productor, a la que la capa de handlers llega de forma barata. Solo el conexionado necesita un broker, y ahí el número de pruebas es pequeño porque cada una demuestra una propiedad de la configuración, no de la lógica de negocio.

Probar handlers

Una prueba de handler debería enunciar el contrato en el lenguaje del libro: dado este mensaje y este estado, el efecto ocurre una vez y el evento saliente se escribe en el outbox. Después la misma prueba tiene un segundo acto, en el que el mensaje llega de nuevo.

```
function testReservationIsIdempotent():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
    lines: [{ sku: "sku-17", qty: 2 }])

  handleOrderPlaced(db, message)
  // same event_id: the broker redelivered it
  handleOrderPlaced(db, message)

  assert stockAvailable(db, "sku-17") == 3
  assert reservationCount(db, "order-1") == 1
  assert outboxCount(db, "StockReserved", "order-1") == 1
```

La última aserción es la que los equipos olvidan. Un duplicado que no reserva dos veces pero emite `StockReserved` dos veces sigue causando daño más adelante, y comprobar el outbox lo detecta.

Esa prueba cubre los duplicados secuenciales, que maneja la comprobación de ruta rápida del inbox. El caso peligroso es el duplicado concurrente, en el que dos workers pasan la comprobación. Para probarlo, el arnés ejecuta dos handlers al mismo tiempo y usa una barrera para retener a ambos justo antes de la transacción, de modo que el entrelazado se fuerce en lugar de esperarlo.

```
function testConcurrentDuplicatesCommitOnce():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
    lines: [{ sku: "sku-17", qty: 2 }])
  barrier = newBarrier(parties: 2)
  setHook("before-transaction", barrier.wait)

  results = runConcurrently(
    handleOrderPlaced(db, message),
    handleOrderPlaced(db, message))

  // one commits; the other fails on the unique constraint and
  // would be redelivered, which is allowed
  assert stockAvailable(db, "sku-17") == 3
  assert reservationCount(db, "order-1") == 1
  assert results.committed == 1
```

Esta prueba falla si alguien elimina la restricción única y se apoya en la comprobación de leer y luego insertar, que es la regresión exacta más difícil de detectar en una revisión. El Capítulo 7 explicó por qué la restricción

es la red de seguridad. Una prueba como esta hace ejecutable esa explicación.

Los puntos de crash funcionan igual. Si el handler llama a `crashPoint("before-ack")` donde iría el ack, una prueba puede armar ese punto y lanzar un crash simulado allí.

```
function testCrashBetweenCommitAndAck():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
                      lines: [{ sku: "sku-17", qty: 2 }])
  armCrash("before-ack")

  crashed = false
  try:
    handleOrderPlaced(db, message)
  on SimulatedCrash:
    // the process died after commit; the broker saw no ack
    crashed = true
    assert reservationCount(db, "order-1") == 1
    // if the crash point was renamed, the test must fail, not pass
    assert crashed

    // a crash point fires once per arming; disarm it for the
    redelivery
    disarmCrash("before-ack")
    handleOrderPlaced(db, message)
    assert reservationCount(db, "order-1") == 1
    assert stockAvailable(db, "sku-17") == 3
```

Lee la aserción dentro del bloque `on`. Comprueba que el commit sobrevivió al crash, que es la mitad de la historia. La otra mitad es la reentrega que sigue. Una prueba de crash que comprobara solo la segunda mitad también pasaría con un handler que nunca hizo commit de nada.

Inyectar duplicados, reordenamiento y retraso

Las pruebas individuales codifican los casos que se te ocurrieron. El modelo de fallos también contiene los que no, así que añadiría una capa de pruebas que perturban un stream entero de eventos y comprueban que el resultado no cambia.

La idea es una propiedad de convergencia. Toma un stream grabado y correctamente ordenado para un agregado. Calcula el estado final esperado plegando los eventos directamente, sin broker de por medio. Después en-

treaga los mismos eventos a través del handler en un calendario perturbado, con algunos barajados dentro de una ventana y otros duplicados, y comprueba que el estado final es igual al esperado.

```
function testOrderViewConvergesUnderAnyDelivery(seed):
  // versions 1 to 6, true order
  events = recordedOrderStream("order-1")
  // reference, no broker, no handler
  expected = foldEvents(events)
  random = randomSource(seed)
  // a failure must be reproducible
  log("seed", seed)

  schedule = shuffleWithin(events, random, window: 3)
  schedule = duplicateSome(schedule, random, rate: 0.3)

  db = freshDatabase()
  for each message in schedule:
    handleOrderEvent(db, message)

  // assumes held events are released when a gap closes (Chapter 9)
  assert heldEventCount(db, "order-1") == 0
  assert loadOrderView(db, "order-1") == expected
```

Ejecútala con unos cientos de semillas en CI. Dos detalles la mantienen honesta. La semilla se registra, de modo que un fallo con la semilla 4172 es un fallo que puedes volver a ejecutar en un portátil en lugar de una anécdota. Y el resultado de referencia sale de un camino de código distinto del del handler, de modo que un bug del handler no pueda aparecer también en la expectativa.

El retraso es la tercera perturbación. Un mensaje retrasado es uno que llega después de sus sucesores, o después de que se haya disparado un timeout. Aquí lo que hay que controlar es el tiempo: los handlers y las sagas que leen directamente el reloj del sistema no se pueden probar en cuanto a timeouts sin esperar. Pasaría el reloj como dependencia y dejaría que la prueba lo avance. El Capítulo 10 enumeró los estados en que puede estar una saga. Una prueba de saga se convierte entonces en una tabla: para cada estado, entrega cada evento que podría llegar, incluidos los tardíos, y comprueba el estado resultante y la compensación emitida.

Estado de la saga	Evento que llega	Resultado esperado
AWAITING_RESULTS	PaymentAuthorized	Se queda en AWAITING_RESULTS, esperando el stock

Estado de la saga	Evento que llega	Resultado esperado
AWAITING_RESULTS, pago retenido	Se dispara el timeout	COMPENSATING; los comandos ReleaseStock y VoidAuthorization se escriben en el outbox
COMPENSATING o CANCELLED	StockReserved tardío	Ninguna transición de la saga. Un StockReserved tardío significa que Inventory reservó antes de que ReleaseStock le llegara, así que ese comando libera el stock. Si la liberación hubiera llegado primero, su tombstone habría rechazado la reserva (Capítulo 10)
COMPLETED	PaymentCaptured duplicado	Se ignora

La tercera fila es la que importa. La llegada tardía tras un timeout es el caso que las sagas hacen mal, porque el happy path y el camino del timeout se probaron cada uno por separado y la colisión nunca. La tabla tiene que escribirla alguien que se pregunte “¿qué más podría llegar en este estado?” y no alguien que siga el diagrama de flujo.

Nota: Si tus pruebas dependen de un calendario aleatorizado, conserva algunas semillas fallidas como casos de regresión fijos una vez encontradas. Un bug que se reprodujo una vez debe seguir reproduciéndose.

Pruebas de contrato y comprobaciones de compatibilidad

Las pruebas de handler construyen sus propios eventos, lo que significa que prueban al consumidor contra la idea que el consumidor tiene del productor. Cuando el productor cambia un campo, nada en la suite del consumidor lo nota. El Capítulo 12 describió las reglas de compatibilidad y un schema registry que las aplica. Eso cubre la estructura. Las pruebas de contrato cubren la parte que el registry no puede ver: de qué depende realmente cada consumidor.

En un **contrato dirigido por el consumidor**, cada consumidor escribe los campos y tipos que lee de un evento, y el pipeline del productor comprueba que su esquema sigue proporcionándolos. La dirección importa. El pro-

ductor no puede saber qué campos son críticos, y el consumidor sí. Un contrato es por tanto un archivo pequeño que pertenece al consumidor y se guarda donde el CI del productor pueda leerlo.

```
contract inventory-reads-order-placed:
    consumer:      inventory
    event_type:    OrderPlaced
    schema_version: 1
    requires:
        payload.order_id      text, required
        payload.lines[].sku    text, required
        payload.lines[].qty    integer, greater than zero
    tolerates:    unknown fields
    sample:       minimalEventFrom(requires)

function verifyContracts(contracts, producerSchemas):
    for each contract in contracts:
        schema = producerSchemas.versionOf(contract.event_type,
                                              contract.schema_version)
        for each requirement in contract.requires:
            // optional is not enough: a consumer that reads the field
            // must be able to rely on it being there
            if not schema.guarantees(requirement):
                raise ContractBroken(contract.consumer, requirement)
```

Hacen falta dos mitades. El pipeline del productor ejecuta `verifyContracts` contra cada contrato archivado. El pipeline del consumidor decodifica cada evento de muestra de sus propios contratos con su deserializador real y ejecuta el handler sobre él. Juntas detectan a un productor que deja de proporcionar un campo y a un consumidor que empieza a leer un campo que nunca declaró. La segunda detección se sostiene solo si la muestra se construye a partir del contrato y contiene solo los campos declarados; una muestra copiada del evento completo del productor deja que el consumidor lea un campo no declarado y aun así pase.

Las muestras deben cubrir cada versión que siga en retención, como recomendó el Capítulo 12. El replay lee lo que haya, y un consumidor que maneja solo la forma más reciente falla en el primer evento antiguo. Aquí es también donde pertenece a CI la comprobación de compatibilidad del registry: un cambio de esquema del productor debe hacer fallar el build antes de poder registrarse, no después de llegar al topic.

Sé honesto sobre el límite. Un contrato demuestra que `amount_minor` existe y es un entero. No demuestra que siga significando “el importe en unidades menores”. Cambiar el significado sin cambiar la estructura pasa

todas las pruebas de contrato, y solo lo nota una persona revisora, o una prueba de comportamiento en el consumidor. No trataría los contratos en verde como una afirmación sobre el significado.

Aserciones con consistencia eventual

Una prueba de extremo a extremo publica algo y luego hace una aserción sobre un efecto en otro lugar. El efecto llega al final, así que la aserción necesita una forma que lo tolere. La forma estándar es un polling con un deadline.

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionError(observation.description)
        sleep(interval)
```

La estructura importa más de lo que parece. La comprobación se ejecuta una vez más después de alcanzado el deadline, de modo que una condición cumplida durante la última espera aún pasa, el intervalo es corto en relación con el timeout, y el mensaje de fallo lleva la última observación. Un mensaje que dice “timed out” no dice nada. Uno que dice “order-1 stuck at AWAITING_RESULTS, 1 held event” te dice dónde mirar.

Las pruebas asíncronas flaky tienen una lista corta de causas, y la mayoría se arregla en la causa.

- **Esperas fijas.** Sustitúyelas por `eventually`. El timeout pasa a ser un techo y la prueba termina en cuanto se cumple la condición.
- **Timeouts ajustados a un portátil rápido.** Fija el techo con generosidad, ya que no cuesta nada cuando la condición se cumple pronto, y mantenlo separado de las expectativas de rendimiento. Un presupuesto de latencia es otra prueba con otro entorno.
- **Estado compartido entre pruebas.** Un mensaje sobrante en un topic compartido hace que la siguiente prueba vea un evento que no publi-

có. Da a cada prueba su propio correlation id y filtra por él, o su propio topic o cola.

- **Orden supuesto donde no se promete ninguno.** Si una prueba hace una aserción sobre el orden de dos eventos de particiones distintas, la hace sobre algo que el broker nunca prometió (Capítulo 9).
- **Afirmar que no pasó nada.** “No se envió un segundo correo” no se puede convertir en éxito mediante polling, ya que esperar más podría mostrar uno. Es preferible llevar el sistema a la quiescencia y luego hacer la aserción. La quiescencia es una propiedad de todo el camino, no de un consumidor: el outbox está vacío, el relay está inactivo, el retraso de cada consumidor implicado es cero, no hay ningún retry ni temporizador programado, y las colas de retry y de retraso están vacías. Solo entonces no queda nada que pueda producir el efecto, incluido un salto posterior cuyo mensaje sigue en vuelo.

Para el último caso, el arnés necesita una forma de decir “ejecuta hasta que no quede nada pendiente”. En una prueba de handler o de componente, el arnés puede bombear el consumidor hasta que la cola se drene y el reloj inyectado no tenga temporizadores por disparar. Esto es a la vez más rápido y más determinista que esperar al tiempo de reloj, y es la razón principal por la que mantengo la mayoría de las pruebas de comportamiento en la capa de handlers y uso el tiempo real solo donde lo que se prueba es el conexionado.

Inyección de fallos y caos

Todo lo anterior prueba código de forma aislada. La siguiente pregunta es si el sistema ensamblado se comporta como afirma el modelo de fallos. La inyección de fallos mete un fallo en un entorno en ejecución a propósito y comprueba la respuesta del sistema.

Empieza por el modelo de fallos del Capítulo 3 y elige entradas, no herramientas. Para cada una, escribe la hipótesis en los términos de la propia entrada: “si se mata a un consumidor entre el commit y el ack, la reentrada no cambia nada”. Luego inyéctala: mata el proceso en ese momento, pausa el relay, retrasa las respuestas del broker, corta la conexión entre un servicio y su base de datos, reinicia un nodo del broker a mitad de una pu-

blicación. Después, comprueba los invariantes, no los logs: el stock coincide con las reservas, el libro mayor tiene una entrada por captura, no hay ninguna fila de outbox atascada.

Añade a esa lista los ejercicios de recuperación, porque una defensa que nunca se ha ejercitado es una esperanza. Un backup que nunca se ha restaurado es el ejemplo clásico. Restaura uno en un entorno desechable y comprueba qué le hace una restauración al outbox, al inbox y a los offsets de los consumidores, lo cual cubre el Capítulo 14: filas de outbox resucitadas, filas de inbox retrocedidas de modo que los duplicados ya no se reconocen, offsets que ya no coinciden. Haz un failover a un cluster en espera, o ensáyalo, y compara el tiempo de recuperación y los datos perdidos con los objetivos de recuperación elegidos para ese flujo en el Capítulo 14. Y prueba el ponerse al día: pausa un consumidor el tiempo suficiente para acumular un backlog, reanúdalo mientras el tráfico sigue llegando y comprueba que el backlog se despeja antes del deadline del flujo y antes de que expire la retención. Eso comprueba el ritmo de drenaje del Capítulo 13 y los límites al trabajo en vuelo del Capítulo 6.

Lo ejecutaría en capas de radio de impacto creciente. Las pruebas de flujo en un entorno dedicado inyectan fallos según un calendario en CI, pequeños y repetibles. Un game day periódico en un entorno mayor los inyecta como ejercicio de equipo, en parte para encontrar bugs y en parte para averiguar si alguien se da cuenta. Las alertas del Capítulo 13 están tan bajo prueba como el código. Si mataste a un consumidor y no saltó ninguna alerta, encontraste un bug en el monitoreo.

El caos en producción es un posible paso siguiente, no un punto de partida. Tiene sentido solo cuando se ha demostrado que el sistema sobrevive al mismo fallo en preproducción, cuando se puede acotar el radio de impacto y cuando alguien puede detener el experimento al instante. Un equipo que no tiene eso no debería empezar por ahí, y no me avergonzaría de quedarme en los game days durante mucho tiempo. El valor de una práctica de inyección de fallos está en el ciclo de hipótesis, inyección y comprobación, no en lo dramática que sea la inyección.

Advertencia: Una ejecución de inyección de fallos que comparte un broker, una base de datos o una cuenta de proveedor de pagos con tráfico real es una caída con plan. Aísla el entorno, o restringe la inyección a tráfico sintético, antes de apagar nada.

El replay como prueba de regresión

Un log de eventos es una grabación de lo que realmente ocurrió, y el replay, la relectura deliberada de la historia, lo convierte en una suite de regresión. Toma una porción de eventos reales, pásalos por la nueva versión de un consumidor en un entorno desechable y compara el estado derivado con el que produjo la versión anterior. Las diferencias son o bien intencionadas, como en la corrección de un bug, o bien regresiones. El replay también tiene un uso operativo, recuperarse tras un hueco o una restauración releando la historia, y el Capítulo 14 lo cubre. Esta sección trata solo de usarlo para probar.

Esto detecta cosas que los fixtures escritos a mano nunca detectarían: formas de evento de tres versiones de esquema atrás, secuencias que nadie imaginó, un campo que siempre estaba presente en las pruebas y es nulo en una pequeña fracción de la historia. Es también lo más parecido a una prueba de la afirmación del Capítulo 12 de que los eventos antiguos aún se decodifican.

Un procedimiento viable tiene cinco pasos.

1. **Selecciona una porción.** Una ventana de tiempo, o la historia completa de una muestra de agregados. Una muestra que incluya cada versión de esquema, además de incidentes conocidos, es mejor que una aleatoria.
2. **Enmascárala.** Los eventos reproducidos contienen datos de personas compradoras. Anonimízalos antes de que salgan de su frontera original, o mantén el entorno desechable bajo los mismos controles que producción.
3. **Desactiva los efectos y los eventos emitidos.** El consumidor que hace el replay no debe cobrar tarjetas, enviar correos ni llamar a transportistas. Los efectos van a stubs que registran lo que se les pidió, y los registros de los stubs forman parte del resultado. Su outbox y su relay

deben publicar en un broker desechable, o ser capturados, y nunca apuntar a topics de producción, o el replay republica la historia hacia consumidores en vivo.

4. **Ejecuta las versiones antigua y nueva sobre la misma porción.**

Compara el estado derivado y los eventos emitidos, agregado por agregado.

5. **Clasifica cada diferencia.** Intencionada, inofensiva o una regresión.

Una diferencia que nadie sabe explicar es una regresión hasta que se demuestre lo contrario.

El replay se aplica a la familia de log, donde la historia sigue ahí. En una cola clásica, los mensajes consumidos desaparecen. El equivalente es una muestra capturada: un tap que copia una fracción de los mensajes, anonimizados, a un store que controlas. Es una herramienta más débil porque la captura solo es tan buena como el muestreo, pero sirve al mismo propósito.

El replay también tiene un límite de costo. Las historias grandes llevan tiempo y almacenamiento reales, y las herramientas de comparación son algo que tu equipo tiene que construir y mantener en funcionamiento. Reservaría el replay de historia completa para cambios que afecten a cómo se interpretan los eventos, y usaría una porción menor para los releases rutinarios.

Verificar en producción

Ningún entorno de preproducción reproduce la forma del tráfico, los datos, la configuración y las rarezas de producción. Algunos fallos, como un permiso ausente en un topic tras un deploy, un secret caducado o una cola mal enrutada, aparecen solo en el sistema real. La respuesta no es renunciar a las pruebas de preproducción sino añadir una última capa que se ejecute donde están los fallos.

Un **evento sintético** es un fragmento de tráfico inyectado deliberadamente y etiquetado de forma explícita que recorre el pipeline real y se comprueba al final. Para Parcelly, es un pedido realizado cada pocos minutos por un probe, que fluye desde `OrderPlaced` hasta `ShipmentScheduled`.

```

function runOrderProbe():
    probeId = newId()
    // recorded first, so a crash of the probe itself leaves a trail
    recordProbe(probeId, startedAt: now())
    try:
        // goes through the real API, outbox, relay and broker; the
flag    // travels in a header and is copied like correlation_id
        orderId = placeSyntheticOrder(probeId, headers: { synthetic:
"true" })
        eventually(
            check: flowReached(orderId, "ShipmentScheduled"),
            timeout: probeBudget,
            interval: probeInterval)
    finally:
        // runs on success, on timeout and on error alike
        cancelSyntheticOrder(probeId)

```

La etiqueta es el mecanismo de seguridad, y es un header `synthetic` dedicado, no un prefijo en `correlation_id`. Un identificador que además controla el comportamiento es una trampa, y el Capítulo 13 pide un único formato uniforme de id, `order-81724`, para que la búsqueda por correlación funcione en todas partes. El precio es que un salto que olvide copiar el header convierte un probe en tráfico real sin ningún error. Todos los servicios lo copian con el mismo helper que copia `correlation_id`, el probe usa además una cuenta de persona compradora sintética dedicada que Payments y Shipping reconocen de forma independiente, y una prueba elimina el header a propósito y comprueba que no resulta ningún cobro ni envío reales.

Cada servicio reconoce la marca y cambia sus bordes, no su lógica central: Payments habla con el sandbox del proveedor, Shipping se salta la entrega al transportista, Notifications suprime el envío y Ledger contabiliza en una cuenta de prueba que finanzas excluye de los informes. Estas ramas son código que se ejecuta solo para los probes, así que un error en ellas es un bug que solo expone producción. Inventory necesita el mismo tratamiento, o cada probe reserva stock real: reserva contra un SKU sintético dedicado, o salta la reserva para los pedidos etiquetados. Probaría las propias ramas en la capa de handlers, y probaría el probe rompiendo algo a propósito y comprobando que falla.

La limpieza tiene una trampa más. Un probe que agota el timeout no necesariamente ha fallado en terminar; sus eventos pueden seguir en vuelo, y un `StockReserved` o `PaymentAuthorized` tardío puede llegar después de la cancelación. Por eso el probe se registra de antemano y cada servicio se guía por la marca `synthetic`: un barrido periódico puede encontrar probes más antiguos que el presupuesto y cancelar lo que hayan dejado, y la cancelación es en sí idempotente, de modo que ejecutarla dos veces es inofensivo.

Un probe también necesita un presupuesto, y el presupuesto necesita un responsable. Un probe que falla envía un page a alguien, así que el presupuesto tiene que ser lo bastante generoso para evitar falsas alarmas y lo bastante ajustado para tener sentido. El Capítulo 13 cubre la latencia de extremo a extremo y cómo alertar sobre ella. El probe es la fuente más barata de esa medición, ya que ejercita todo el camino, incluidas las partes que ningún pedido real tocó en la última hora.

La verificación en producción incluye los jobs de conciliación del Capítulo 13. Una ejecución de conciliación es una prueba de invariantes sobre datos reales, ejecutada de forma continua, y detecta los fallos que a nadie se le ocurrió simular. La cuento como la última capa de la pirámide.

Trade-offs y costos

El realismo cuesta tiempo. Las bases de datos reales, los brokers reales y los fallos inyectados hacen más lentas las pruebas. La suite que era rápida lo era porque se saltaba lo que importaba. Presupuestaría explícitamente las capas lentas: pruebas de handler en segundos, pruebas de componente en unos pocos minutos, pruebas de flujo e inyección de fallos según un calendario y no en cada commit.

Las pruebas de contrato añaden coordinación. Cada consumidor es responsable de archivos que viven en el pipeline del productor, y alguien tiene que mantenerlos al día. Para un sistema con dos servicios y un equipo, una prueba compartida del esquema puede bastar. Las pruebas de contrato empiezan a compensar cuando los equipos del productor y del consumidor despliegan de forma independiente, y rinden poco antes de eso.

Los calendarios aleatorizados añaden trabajo de investigación. Una semilla que falla es a la vez un rompecabezas y un defecto real. Los equipos a los que no les gusta acaban reduciendo el número de semillas hasta que la prueba ya no encuentra nada. La cura es registrar las semillas y conservar regresiones fijas, no abandonar la técnica.

El caos y los probes tienen un radio de impacto. Ambos tocan sistemas en vivo o compartidos. Cuándo no usarlos: cuando el sistema aún no sobrevive al fallo de forma aislada, cuando nadie puede detener la ejecución de inmediato, o cuando el tráfico etiquetado no puede mantenerse separado del dinero real y de las personas compradoras reales.

El replay es caro de montar. La infraestructura de enmascaramiento, stubs de efectos y comparación es un proyecto en sí. Un sistema pequeño con una historia corta saca más de las pruebas de handler y los puntos de crash. No sé dónde está tu punto de equilibrio, ya que depende de la frecuencia con que cambian los consumidores y de lo costosa que sería una interpretación errónea de la historia, pero un equipo que ya ha tenido un incidente por un evento antiguo inesperado normalmente puede adivinarlo.

Errores comunes

- **Probar contra un fake de broker que nunca se porta mal.** Es tentador porque es rápido y las primeras pruebas pasan. Haz que el fake duplique, retrase y reordene, o pon el fallo en las pruebas de handler con llamadas explícitas, como en este capítulo.
- **Falsear las restricciones de la base de datos.** Un fake escrito a mano no aplica unicidad, aislamiento ni atomicidad, así que las pruebas pasan exista o no la defensa. Usa el motor real, en un contenedor o en una instancia desechable, para cualquier prueba que dependa de una restricción.
- **Dormir un tiempo fijo.** Parece la forma más sencilla de esperar. Haz polling de la condición con un techo e informa de la última observación al fallar.
- **No probar nunca el crash entre el commit y el ack.** Parece exótico, aunque ocurre en cada deploy. Añade un punto de crash a cada handler y una prueba que lo arme.

- **Probar solo el estado final, no el outbox.** Un duplicado que no cambia el estado puede aun así emitir un segundo evento. Haz la aserción sobre lo que se escribió en el outbox.
- **Pruebas de contrato que comprueban la presencia pero no la obligatoriedad.** Un campo opcional en el esquema del productor puede estar ausente. Haz que el contrato diga “required” y compruébalo.
- **Pruebas aleatorizadas sin semillas.** Un fallo que no se puede reproducir acaba borrado. Registra la semilla y conserva las semillas que fallan como casos fijos.
- **Confiar en un backup que nunca se restauró.** El job de backup informa de éxito, así que la defensa parece probada. Restáuralo en un entorno desechable de forma periódica y comprueba después el outbox, el inbox y los offsets.
- **Ejecutar probes contra dinero real.** El tráfico sintético que llega al proveedor de pagos en vivo o a un transportista real no es un probe. Enruta el tráfico etiquetado a sandboxes en el borde, y prueba ese enrutamiento.

Ideas clave

- Las pruebas que suponen un broker que entrega cada mensaje una vez y en orden prueban un sistema que no existe, así que cada defensa de este libro necesita una prueba que aplique el fallo contra el que defiende.
- Las pruebas de handler pertenecen a un motor de base de datos real, porque el inbox, el outbox y las comprobaciones de versión dependen de restricciones y transacciones que los fakes no reproducen.
- Los duplicados, el reordenamiento y el retraso se prueban mejor como una propiedad de convergencia sobre un stream perturbado, con la semilla registrada para que cualquier fallo pueda reproducirse.
- Los contratos dirigidos por el consumidor detectan roturas estructurales antes del deploy, pero no pueden ver un cambio de significado, así que una revisión humana de lo que se supone que dice un campo sigue formando parte del proceso.
- Sustituye las esperas fijas por polling contra una condición y un techo, haz aserciones sobre la quiescencia y no sobre el silencio, e inyecta el

reloj para que los timeouts puedan probarse sin esperar.

- La inyección de fallos debe partir de las entradas del modelo de fallos y comprobar los invariantes después, aumentando el radio de impacto solo a medida que el sistema demuestra que sobrevive a cada paso, e incluir ejercicios de recuperación para restauraciones, failovers y ponerse al día.
- El replay de la historia real y los probes sintéticos en producción detectan lo que los fixtures escritos a mano y los entornos de preproducción no pueden, y necesitan enmascaramiento, stubs de efectos y tráfico etiquetado para ser seguros.

Un sistema probado de este modo merece algo más de confianza, aunque nunca completa, y eso deja la pregunta de qué debería comprobar un equipo antes de confiar siquiera en un diseño. El Capítulo 16 reúne el libro en una lista de verificación de confiabilidad, y cierra con lo que la lista no puede hacer.

Capítulo 16 — Una lista de verificación de confiabilidad y reflexiones finales

Imagina una revisión de diseño en Parcelly un jueves tranquilo. El equipo está a punto de añadir un flujo nuevo: los vendedores podrán ofrecer reembolsos parciales y Payments emitirá un evento nuevo para ello. El diagrama es limpio. Alguien dibujó las flechas, puso nombre al evento y enumeró a los consumidores. La reunión está prevista para 30 minutos y parece que terminará en 20.

Entonces quien hace de tech lead plantea una pregunta sencilla: “¿Qué pasa si el reembolso se emite y el evento nunca se publica?”. La sala se queda en silencio, no porque nadie conozca el patrón, sino porque nadie lo había preguntado. El outbox existe en otra parte del código. El consumidor idempotente también. Si este flujo nuevo los usa es otra cuestión, y el diagrama no lo dice.

Así suele erosionarse la confiabilidad. El equipo conoce los patrones. Lo que se desvanece es el hábito de comprobar, en cada flujo nuevo, que cada patrón está realmente en su sitio. Por eso la pregunta de este último capítulo es práctica: ¿qué preguntarías, flujo por flujo, para saber si el sistema puede cumplir sus promesas en un mal día, y quién tiene que ser responsable de las respuestas?

El problema y por qué es difícil

La confiabilidad es una propiedad de las uniones, y las uniones no se ven en ningún archivo concreto. Un productor puede ser correcto, un consumidor puede ser correcto, y el flujo entre ambos puede perder eventos de todos modos, porque nadie examinó el traspaso. El Capítulo 1 planteó este argumento al principio del libro, y los capítulos siguientes han recorrido las uniones una por una.

La dificultad ahora es que el conocimiento está repartido en 15 capítulos. Nadie guarda tanto en la cabeza durante una revisión. Con la presión de los plazos, las preguntas que se saltan son aquellas cuya respuesta es “funciona hasta el día en que deja de funcionar”: la ventana de deduplicación, el responsable de los dead letters, la comprobación de compatibilidad, la alerta que se dispara por antigüedad y no por cantidad.

Una lista de verificación es la respuesta habitual, y tiene límites conocidos. No puede pensar. Quien no entiende un punto puede marcarlo igualmente, y una lista completada da entonces una falsa confianza. Usaría la lista siguiente como pie para una conversación, no como un gate que un formulario pueda superar. Cada punto remite al capítulo que explica por qué está ahí, de modo que quien no esté de acuerdo pueda encontrar el razonamiento y discutir la premisa.

Trata cada punto como una pregunta sobre un flujo concreto, como el flujo de pedidos de Parcelly, y no sobre el sistema en general. “Usamos un outbox” no es una respuesta. “Orders escribe `OrderPlaced` en su outbox en la misma transacción que la fila del pedido” sí lo es.

El método detrás de la lista

La lista no es una idea nueva. Es la tesis del libro aplicada a una revisión: los sistemas confiables orientados a eventos se construyen en torno a la **semántica de fallos**, no a los message brokers: para una unión dada, qué puede fallar, cómo lo observarías y qué garantía sobrevive.

Eso da tres preguntas, y cada punto de abajo es una de ellas formulada sobre una unión. Un punto que nombra un fallo pregunta qué puede fallar. Un punto que nombra una métrica, una alerta o una prueba pregunta cómo lo observarías. Un punto que nombra una configuración, una restricción o un patrón pregunta qué garantía sobrevive. Una unión cuyos puntos responden solo a una de las tres está a medio hacer, y una unión sin respuesta a la segunda pregunta fallará sin avisar a nadie.

La lista de verificación

La lista está agrupada según dónde vive el fallo. La columna de la derecha nombra el capítulo que conviene releer cuando una respuesta es “no” o “no estoy seguro”.

Productor

Verificación	Capítulo
El cambio de estado y el evento se escriben en una sola transacción local, sin publicación directa en la ruta de negocio.	4
La fila del outbox lleva el envelope completo, y el <code>event_id</code> se crea en el momento de la escritura y nunca se regenera.	4
El equipo ha dejado por escrito que la publicación es at-least-once y que los duplicados son el precio.	4
El relay publica primero y marca después, y selecciona por estado, no por un cursor de id creciente.	5
Si hay varios relays, reclaman filas de forma segura y mantienen las filas de un mismo agregado en un solo worker.	5
Se mide la antigüedad de la fila más antigua sin publicar, y solo se purgan las filas publicadas.	5

Si un equipo solo puede responder una pregunta en una revisión, querría que fuera la primera, porque toda garantía posterior supone que el evento existe. Cuando un flujo no necesita un outbox, porque sus pasos pueden compartir una transacción local, el equipo debería decirlo y explicar por qué (Capítulo 1).

Broker

Verificación	Capítulo
Para cada evento importante, el equipo ha registrado qué significa “publicado”: la configuración de replicación y de flush que hay detrás del ack del productor.	2, 3
El ack o el commit del offset ocurre después del efecto, de modo que la semántica es at-least-once y no un at-most-once accidental.	2
Los visibility timeouts, los timeouts de sesión y el tiempo de retry dentro del handler se fijan en conjunto, de modo que el trabajo lento no se reentregue mientras aún se ejecuta.	2, 6
La clave de partición identifica el agregado, y el equipo sabe qué orden ofrece su familia de brokers.	9
Todo productor, en cualquier lenguaje, usa el mismo partitioner y la misma codificación de clave, o el relay calcula la partición por sí mismo.	9
La retención supera la interrupción realista más larga más el tiempo que se tarda en notarla.	3, 13

Verificación	Capítulo
Los consumidores se detienen de forma ordenada en un deploy, de modo que el rebalanceo no sea una fuente rutinaria de duplicados.	3

La mayoría de estos puntos son configuraciones, no código. Se quedan en sus valores por defecto hasta que un incidente muestra cuáles eran, y para entonces es difícil rastrearlos.

Consumidor

Verificación	Capítulo
La fila del inbox y el cambio de negocio se confirman en una sola transacción local, con una restricción única sobre la clave.	7
La ventana de deduplicación dura más que la retención, el redrive y el replay planificado, y una restricción de clave de negocio cubre la tabla del dominio más allá de ella.	7
Los efectos secundarios externos llevan una clave de idempotencia determinista derivada del trabajo, no del intento.	7
Los fallos se clasifican como transitorios, permanentes o desconocidos, y los retries usan exponential backoff con jitter, un deadline y un presupuesto compartido.	6
El trabajo en curso tiene un tope, la concurrencia baja cuando la dependencia se ralentiza y el consumidor se pausa en lugar de fallar, de modo que la sobrecarga se vea como retraso; un límite en la entrada rechaza o aplaza el trabajo nuevo con una señal clara para quien llama.	3, 6
Un poison message termina en una DLQ con cuerpo, cabeceras, posición de origen, error, número de intentos y versión del consumidor, publicado antes de que se haga ack del original.	8
Se detecta un consumidor atascado en un registro, por el retraso sin throughput y por un contador de fallos por posición, y no entra en bucle hasta que alguien lo nota.	8, 13
El redrive conserva el <code>event_id</code> original, se ejecuta en lotes con límite de ritmo y tiene un interruptor de pausa.	8
El orden por agregado se protege con lanes por clave, verificaciones de versión y hold-back buffers que tienen timeouts.	9
Todo paso del flujo anterior al pivot tiene una compensación idempotente, y toda espera tiene un deadline.	10

Este es el grupo más largo porque la mayoría de los patrones del libro se concentran aquí. Si resulta pesado, es una descripción exacta: la entrega at-least-once pone la carga sobre el consumidor.

Contrato

Verificación	Ca- pítu- lo
Cada evento enuncia un hecho, se nombra en pasado y tiene un único servicio responsable.	11
El evento de integración es un contrato público deliberado, mapeado desde el evento de dominio, sin datos personales ni secrets en el payload.	11
Un comando se nombra en imperativo, tiene un único destinatario previsto, lleva un <code>command_id</code> , tiene una vía de respuesta para que un rechazo llegue a quien lo envió, y nunca se llama evento; quien lo envía espera con un deadline propio.	10, 11
El <code>correlation_id</code> se copia a lo largo del flujo y el <code>causation_id</code> apunta al desencadenante directo.	11
Los tipos en el cable del envelope, como marcas de tiempo, identificadores y dinero, están fijados para todos los lenguajes que lo leen o lo escriben.	11
Cada cambio se clasifica como compatible o incompatible, y la compatibilidad se comprueba en el pipeline de deploy.	12
Una persona revisa los cambios de significado, porque ninguna herramienta los detecta.	12
Los consumidores ignoran los campos que no usan y toleran los valores que no han visto, un salto que reenvía un mensaje conserva los campos desconocidos, y un evento cuyo <code>schema_version</code> no conocen se rechaza o se aparca.	12
Los cambios incompatibles se migran como una secuencia de pasos compatibles, con los rollbacks planificados en el mismo documento, y se conocen los consumidores que aún leen la versión antigua.	11, 12

Operaciones

Verificación	Ca- pítu- lo
El contexto de trace y el <code>correlation_id</code> sobreviven a cada salto asíncrono, de modo que la historia de un pedido pueda reconstruirse a partir de su identificador.	13

Verificación	Capítulo
Se miden el retraso del consumidor, la antigüedad de los mensajes, la antigüedad del outbox, la tasa de retries y las llegadas a la DLQ, y las alertas se disparan por antigüedad y por tasa.	13
Los jobs de conciliación comparan la fuente de verdad con el estado derivado de forma programada, y las reparaciones se revisan antes de ejecutarse.	13
Los invariantes de negocio, como “todo pago capturado tiene un asiento en el libro mayor”, se comprueban directamente.	13
Los handlers se prueban con duplicados, reordenamientos y retrasos, y las pruebas de contrato se ejecutan en CI.	15
Existen inyección de fallos y pruebas de regresión basadas en replay para los flujos que más importan.	15
Cada almacén tiene un objetivo de punto de recuperación (RPO) y un objetivo de tiempo de recuperación (RTO) declarados, y el equipo sabe qué almacén retrocede en el tiempo en una restauración.	14
Un procedimiento de replay nombra los grupos de consumidores, la marca de tiempo y su margen, los efectos secundarios que suprime y la ventana de deduplicación dentro de la cual debe caber cada rango reproducido.	7, 14
Un plan de migración de brokers usa el relay como único punto de conmutación, hace dual-publish con deduplicación en el consumidor y tiene pasos de cutover y de rollback.	14
Se han ensayado las restauraciones y los game days, y los runbooks de más abajo se escribieron a partir de lo que mostró el ensayo.	14, 15
Un modelo de fallos escrito nombra responsables, pruebas y fechas de revisión.	3

Preguntas para una revisión de diseño

La lista de verificación comprueba un diseño que existe. Estas preguntas ayudan a dar forma a uno que todavía no existe. Las haría en este orden, porque cada respuesta condiciona la siguiente.

1. **¿Cuál es el hecho y quién es su responsable?** Si no puedes nombrar el evento en pasado y nombrar un único servicio responsable, el diseño no está listo para un broker (Capítulo 11).
2. **¿Hace falta siquiera un evento?** Si los pasos pueden compartir una transacción local, o quien llama necesita una respuesta inmediata, una llamada síncrona puede ser la mejor herramienta (Capítulo 1).

3. **¿Qué pasa entre el commit y la publicación?** Pregunta por la ventana de caída, no por el patrón. “Publicamos justo después del commit” es una pérdida que espera a un deploy (Capítulo 4).
4. **¿Qué pasa cuando este mensaje llega dos veces, y qué cuando llega tarde?** Son fallos distintos con arreglos distintos (los capítulos 7 y 9).
5. **¿Cuál es el límite de “al final”?** Pon nombre a una duración, a una alerta y a un estado visible para el usuario durante la espera (Capítulo 1).
6. **¿Qué pasa cuando este mensaje nunca podrá tener éxito?** Pregunta dónde acaba, quién lo ve y cómo vuelve (Capítulo 8).
7. **¿Qué pasa cuando el consumidor, o la dependencia que hay detrás, es más lento que el productor?** Pregunta qué tiene un tope, qué se pausa y qué trabajo espera, y quién es responsable de esa decisión (los capítulos 3 y 6).
8. **Si falla un paso posterior, ¿qué deshace los anteriores?** Nombra la compensación y el pivot a partir del cual ya nada se deshace (Capítulo 10).
9. **¿Cómo cambiaremos este contrato dentro de un año?** Si nadie lo ha pensado, la respuesta se improvisará en un incidente (Capítulo 12).
10. **¿Cómo sabremos que está roto y cómo sabremos que está reparado?** Detectar y verificar son preguntas distintas (los capítulos 13 y 15).
11. **Si se pierde un almacén, el broker o una región, ¿cuánto perdemos y cuánto tardamos en volver?** Pide un RPO y un RTO por almacén, y el replay o la restauración que sigue (Capítulo 14).
12. **¿A quién se le envía un page y qué hace primero?** Si la respuesta es un nombre, tienes un responsable. Si es “el equipo”, quizá no.

Algunas de estas preguntas no tienen una respuesta limpia en un día dado, y no pasa nada. “Todavía no lo sabemos, y así vamos a averiguarlo” es un resultado mejor que una respuesta segura que nadie ha comprobado.

El lado organizacional

Todo patrón de este libro tiene una mitad humana. Un outbox necesita a alguien que vigile la antigüedad de su fila más antigua. Una DLQ necesita a alguien que la lea. Un schema registry necesita a alguien que decida qué significa “compatible”. Cuando falta la mitad humana, el patrón sigue existiendo en el diagrama y ya no cumple su función.

Responsabilidad

Dejaría por escrito tres tipos de responsabilidad.

- **Responsabilidad sobre el evento.** Cada evento tiene un servicio y un equipo responsables, que documentan qué significa el evento y quién lo consume (Capítulo 11).
- **Responsabilidad operativa.** Cada DLQ, parking lot, informe de conciliación y alerta tiene un responsable con nombre. Una alerta que no es de nadie es una notificación, y las notificaciones se ignoran.
- **Flujos entre servicios.** El flujo de pedidos de Parcelly atraviesa seis servicios. Algún equipo tiene que ser responsable del flujo en su conjunto, incluidos sus timeouts y sus flujos atascados, aunque ningún servicio por sí solo lo sea (Capítulo 10).

Runbooks

Un runbook es lo que escribes suponiendo que quien lo lee está cansado y no ha visto el sistema en seis meses. Para cada alerta, di qué significa, qué mirar primero, qué acciones son seguras, cuáles no y a quién llamar si los primeros pasos no ayudan.

Algunos runbooks están ligados a una alerta y otros a un suceso tan raro que nadie del equipo lo ha hecho recientemente. El Capítulo 14 añade el segundo tipo. Su sección sobre ensayar la recuperación enumera ocho preguntas que un runbook de recuperación tiene que responder, por escrito y no guardadas en la cabeza de alguien:

1. ¿Qué almacén está afectado y cuál retrocedió en el tiempo?
2. ¿Cuál es el punto de recuperación y cómo se determinó?
3. ¿Qué grupos de consumidores hay que reiniciar, a qué marca de tiempo y con qué margen?

4. ¿Qué efectos hay que suprimir durante el replay y mediante qué mecanismo?
5. ¿Está cada rango que se va a releer dentro de la ventana de deduplicación de cada consumidor afectado?
6. ¿Qué se pausa, en qué orden y quién lo reanuda?
7. ¿Qué conciliaciones confirman que la recuperación funcionó y quién las lee?
8. ¿Cómo se deshace todo si va mal?

La tabla enumera los sucesos para los que escribiría un runbook. La última columna marca cuáles de las ocho preguntas esperaría que respondiera cada uno. Es un punto de partida, y el sistema de cada equipo añadirá preguntas propias.

Suceso	La acción peligrosa	Preguntas
Un consumidor atascado en un poison message (Capítulo 8)	Saltarse la posición sin dejar registro de ello, o subir el límite de retries hasta que el mensaje pase	6, 7, 8
Un consumidor cuyo retraso se acerca a la retención (los capítulos 13 y 14)	Dejar que los mensajes caduquen sin leer, o reiniciar hacia delante sin que nadie lo decida	2, 3, 5, 7
Reinicio de offsets y replay (Capítulo 14)	Reproducir con los efectos secundarios activos, o más allá de la ventana de deduplicación	3, 4, 5, 6, 8
Redrive (Capítulo 8)	Devolver una DLQ entera de golpe a una dependencia que todavía se está recuperando	5, 6, 8
Reparación y backfill (Capítulo 13)	Ejecutar un script sin revisar contra el estado de producción	4, 7, 8
Restaurar una base de datos desde un backup (Capítulo 14)	Reanudar los consumidores sin reiniciar sus offsets, o reutilizar identificadores que la secuencia retrocedida vuelve a entregar	1, 2, 3, 4, 5, 6, 7, 8
Pérdida de datos o failover del broker (Capítulo 14)	Reanudar los grupos en posiciones que ya no significan lo que significaban	1, 2, 3, 5, 6, 7
Evacuación de una región (Capítulo 14)	Promover un standby sin saber cuánto se había retrasado	1, 2, 3, 6, 7, 8
Cutover de la migración de brokers (Capítulo 14)	Mover los consumidores antes de deduplicar las copias publicadas por duplicado, o sin forma de volver	3, 5, 6, 8

Dos de estos casos merecen un cuidado particular. El redrive va primero, porque uno descuidado convierte un fallo contenido en una inundación (Capítulo 8). La reparación y el backfill van después, porque un script que corrige estado derivado es un cambio en producción y merece una revisión, un dry-run y un registro de lo que hizo (Capítulo 13).

On-call

El on-call de un sistema orientado a eventos se diferencia del de un servicio de solicitud-respuesta en algo que importa: el fallo suele ser silencioso. Un servicio síncrono devuelve errores y una gráfica se pone roja. Un sistema orientado a eventos puede no devolver nada en absoluto, porque el mensaje está en una cola, esperando. Las alertas sobre antigüedad, retraso y trabajo sin procesar son la señal principal, y quien está de on-call necesita el acceso suficiente para mirar dentro de la cola, de la DLQ y del outbox sin esperar a que le den permiso.

Nota: Practica la reparación antes de necesitarla. Un equipo que ha hecho un redrive de una DLQ de prueba una vez, de día, con un colega mirando, afrontará la real de otra manera que un equipo que solo ha leído el runbook. El Capítulo 14 dice lo mismo sobre las restauraciones y los game days, y el Capítulo 15 explica cómo diseñar los simulacros.

Trade-offs y costos

Una lista de verificación tiene costos, y el primero es la atención. Una lista de 43 puntos aplicada a cada flujo es demasiado larga, y un equipo que los aplique todos a un flujo de notificaciones de bajo riesgo concluirá que el problema es la lista. Ponderaría la lista según el costo del fallo. Un flujo que mueve dinero o cambia existencias merece la lista entera. Un correo de recordatorio merece idempotencia y un responsable de la DLQ, y quizá poco más.

El segundo costo es la falsa seguridad. Una lista completada dice que alguien consideró cada punto, no que el sistema sea correcto. Los capítulos 13, 14 y 15 existen porque los diseños bien pensados siguen fallando de formas que sus autores no imaginaron. Trata la lista de verificación como la capa de defensa más barata, no como la última.

El tercer costo es la deriva. Una lista escrita una sola vez se convierte en el registro de lo que el equipo creía en un momento dado. Revísala tras cada incidente y elimina los puntos que han dejado de ganarse su lugar. Una lista corta que la gente usa vale más que una larga que la gente firma.

Cuándo no usarla: no uses la lista de verificación para sustituir al criterio, ni la exijas para prototipos desechables. Si un flujo se va a borrar el mes que viene, el costo de la lista completa supera el costo del fallo. Dilo explícitamente y anota la fecha en que el prototipo muere o recibe una revisión como es debido.

Errores comunes

- **Tratar la lista de verificación como un gate.** Un gate se puede auditar contando marcas, y esa es la tentación. El resultado es un formulario relleno y un diseño sin discutir. Usa los puntos como preguntas y registra las respuestas que sorprendieron a alguien.
- **Comprobar patrones en lugar de flujos.** “Tenemos un outbox” da sensación de avance. Pregunta si este flujo lo usa, y nombra la tabla y la transacción.
- **Aplicarlo todo en todas partes.** Parece exhaustivo, y después lleva al cansancio y al abandono. Pondera la lista según el costo del fallo y di qué puntos omites y por qué.
- **Dejar los puntos humanos para después.** Los responsables, los runbooks y las fechas de revisión parecen menos urgentes que el código, y son lo primero que falta cuando una alerta salta de noche. Asígnalos en la propia revisión de diseño.
- **No volver nunca a las respuestas.** Un flujo que pasó la revisión hace un año ha ganado desde entonces consumidores, versiones y personas. Pon una fecha de revisión a las respuestas, como sugiere el Capítulo 3 para el modelo de fallos.
- **Crear que la lista está completa.** Tu sistema tiene modos de fallo que este libro no nombra. Añádelos a tu propia lista cuando los encuentres.

Ideas clave

- La confiabilidad se erosiona por preguntas omitidas más a menudo que por patrones desconocidos, así que un conjunto fijo de preguntas formuladas en cada flujo nuevo protege un diseño mejor que la experiencia presente en la sala.
- Cada punto de la lista de verificación es una de las tres preguntas del libro, qué puede fallar, cómo lo observarías y qué garantía sobrevive, formulada sobre una unión, así que cada punto debe responderse para un flujo concreto y no para el sistema en general.
- Pondera la lista según el costo del fallo: los flujos que mueven dinero o existencias merecen todos los puntos, y los flujos de bajo riesgo merecen los pocos que evitan efectos duplicados y pérdidas silenciosas.
- Todo patrón de este libro tiene una mitad humana, así que los responsables, los runbooks, los destinatarios de las alertas y las fechas de revisión pertenecen a la misma revisión de diseño que el código.
- Los fallos de los sistemas orientados a eventos son silenciosos, así que el on-call depende de alertas sobre antigüedad, retraso y trabajo sin procesar, y los sucesos raros, una restauración, un replay, la pérdida de una región y una migración de brokers, necesitan runbooks que se hayan ejecutado de verdad.
- Una lista de verificación completada demuestra que alguien pensó en cada punto, no que el sistema sea correcto, y por eso la conciliación y las pruebas existen como capas adicionales.
- La confiabilidad no es una funcionalidad que se entrega una vez, sino una disciplina que se sigue practicando, en preguntas pequeñas, respuestas escritas y reparaciones ensayadas antes de necesitarlas.

Parcellly nunca llega a ser confiable como una funcionalidad llega a estar terminada. Se vuelve más confiable cada vez que alguien pregunta qué pasa cuando el mensaje llega dos veces, deja por escrito la respuesta y le asigna un responsable. Ningún broker ofrece eso, y ninguna biblioteca lo incluye. Es un hábito de atención, mantenido por personas que saben que todo sistema que construyan será maltratado algún día por una red, un deploy o el cansancio de un colega, y que han decidido diseñar para ese día sin dramatismo. Eso es lo que te pediría que te lleves de este libro: no un conjunto de patrones, sino el hábito de hacer la siguiente pregunta.

Apéndice: Glosario y referencia rápida de patrones

Este apéndice tiene dos partes. La Parte A es un glosario de los términos que usa el libro, en orden alfabético, con el capítulo donde cada término se define o se usa con más amplitud. La Parte B es una referencia rápida de los patrones, escrita como una tabla que puedes recorrer durante una revisión de diseño.

Ninguna de las dos partes sustituye a los capítulos. Una definición aquí es un recordatorio, y una fila de la tabla es una indicación. El razonamiento, los costos y los casos en que un patrón no debe usarse están en el capítulo que se nombra en la última columna.

A. Glosario

acknowledgment (ack). La señal del consumidor al broker de que un mensaje se ha procesado y puede retirarse de la cola, o de que la posición del consumidor puede avanzar. La forma negativa es un **nack**. Capítulo 2.

agregado. El límite de consistencia que posee una clave de partición. Los eventos de un mismo agregado están ordenados entre sí; los eventos de agregados distintos no. Capítulo 9.

antigüedad del mensaje. El tiempo transcurrido desde que se produjo el mensaje más antiguo que sigue esperando, comparado con un deadline de negocio y con el período de retención. Capítulo 13.

at-least-once. Una semántica de entrega en la que un mensaje se entrega una o más veces, de modo que los duplicados son posibles. Se cumple mientras el mensaje sigue retenido y el consumidor hace ack solo después del efecto. Un consumidor que se retrasa más allá de la retención sigue perdiendo mensajes en silencio. Es la opción por defecto del libro. El Capítulo 2 la define; el Capítulo 3 trata la pérdida.

at-most-once. Una semántica de entrega en la que un mensaje se entrega cero o una vez, de modo que un mensaje puede perderse pero nunca se entrega dos veces. Capítulo 2.

backfill. Reparación o reconstrucción masiva que alimenta a un consumidor activo con un gran conjunto de eventos, ejecutada en lotes pequeños y con límite de ritmo. Capítulo 13.

backpressure. Acotar el trabajo en curso en cada etapa para que la carga se mantenga finita. En un sistema mediado por un broker aparece como límites en la admisión del consumidor, como el prefetch o los registros por poll, y como retraso que el broker absorbe. La presión llega a un productor solo mediante el control de flujo del broker o una limitación explícita. El Capítulo 3 la sitúa en el modelo de fallos; el Capítulo 6 da los mecanismos.

bloqueo semántico. Una marca de estado en los datos que indica que una saga está trabajando sobre un registro, y que el resto del código está escrito para respetar. No es un bloqueo de base de datos. Capítulo 10.

causation_id. El campo del envelope que apunta al evento o comando que causó directamente este, mientras que `correlation_id` nombra todo el flujo. Capítulo 11.

change data capture (CDC). Leer el log de transacciones de una base de datos para convertir los cambios confirmados en mensajes. En el contexto del outbox, un relay basado en CDC lee las filas del outbox desde el log en lugar de hacer polling de la tabla. Capítulo 5.

circuit breaker. Una protección alrededor de una dependencia que detiene las llamadas cuando los fallos superan un umbral y luego deja pasar llamadas de prueba tras una espera. Capítulo 6.

clave de idempotencia. La clave que un consumidor o una API usa para reconocer trabajo repetido. Para el consumo de eventos es normalmente el `event_id`. Capítulo 7.

clave de negocio. Un identificador derivado del significado del trabajo, como el id de un pago más una operación, en lugar de derivarse del envelope. Protege frente a que el mismo trabajo llegue en dos mensajes distintos. Capítulo 7.

clave de partición. El identificador del agregado que se usa para encaminar eventos a una partición y así preservar el orden por agregado. Capítulo 9.

cola. Una estructura del broker en la que cada mensaje se consume y se confirma individualmente. Capítulo 2.

comando. Una solicitud de hacer algo, nombrada en imperativo, como `CapturePayment`. Un comando se dirige a un solo servicio y puede rechazarse. No es un evento. Capítulo 11.

commit de offset. El registro del consumidor, en un broker basado en log, de que ha procesado todo lo anterior a una posición. Sustituye al ack por mensaje. Capítulo 2.

compatibilidad hacia adelante. Un consumidor que usa el esquema antiguo puede leer eventos escritos con el esquema nuevo: lectores antiguos, datos nuevos. Capítulo 12.

compatibilidad hacia atrás. Un consumidor que usa el esquema nuevo puede leer eventos escritos con el esquema antiguo: lectores nuevos, datos antiguos. Capítulo 12.

compensación. El paso que deshace semánticamente un paso anterior de una saga, como liberar existencias o anular una autorización. El libro no la llama rollback. Capítulo 10.

comprobación de invariantes. Una consulta programada para una regla que el negocio da por cierta, como que cada pago capturado tenga exactamente un asiento en el Ledger. Cada fila devuelta es una discrepancia. Capítulo 13.

comprobación de versión. Un consumidor que compara la versión de un evento con la última versión que aplicó para ese agregado, para ignorar eventos obsoletos y detectar huecos. Capítulo 9.

conciliación. Comparar las vistas de la verdad de dos sistemas y reparar las diferencias. Capítulo 13.

consistencia eventual. La propiedad por la que las réplicas o los servicios convergen en el mismo estado si las actualizaciones se detienen, sin una cota de cuánto tarda. Capítulo 1.

consumidor. Un proceso que lee mensajes de un topic o una cola y actúa sobre ellos. Capítulo 2.

consumidor idempotente. Un consumidor cuyo procesamiento repetido del mismo mensaje tiene el mismo efecto que procesarlo una vez. Capítulo 7.

consumidores competidores. Varios consumidores que comparten una cola de modo que cada entrega va a uno de ellos. Un mensaje reentregado puede llegar a otro consumidor. Es el equivalente, en la familia de colas, de un grupo de consumidores. Capítulo 2.

contexto de trace. El trace id y el span id que sitúan una unidad de trabajo en un trace distribuido, capturados en la fila del outbox y enviados en las cabeceras del mensaje para que el span del consumidor pueda enlazarse con el del productor. Capítulo 13.

contrato dirigido por el consumidor. Un archivo propiedad de un consumidor que enumera los campos y tipos que lee de un evento, comprobado en el pipeline del productor. Capítulo 15.

correlation_id. El campo del envelope compartido por todos los eventos de un mismo flujo de negocio. Su compañero `causation_id` apunta al evento o comando que causó directamente este. Capítulo 11.

cutover. El paso de una migración de brokers en el que un grupo de consumidores empieza a leer del broker nuevo, dado grupo por grupo tras conciliar los dos brokers y con un rollback planificado. Capítulo 14.

dead letter. Un mensaje que se ha apartado porque falló su procesamiento. Como modificador, el término lleva guion: dead-letter queue. Capítulo 8.

dead-letter queue (DLQ). Un destino aparte donde un consumidor o un broker deja los mensajes de los que se ha rendido. En un broker basado en log, la misma idea es un **dead-letter topic**. Capítulo 8.

deadline. El momento a partir del cual un resultado ya no es útil. Los retries deben detenerse antes. Capítulo 6.

diferido. La clase de retry para un fallo que no dice nada sobre el mensaje, porque se sabe que la dependencia está caída, como cuando un circuit breaker está abierto. Un mensaje diferido no consume ningún intento y

nunca se envía a dead letters por ello. El consumidor se pausa y el mensaje espera. Capítulo 6.

dual write. Actualizar dos sistemas, como una base de datos y un broker, sin una transacción compartida. Capítulo 4.

dual-publish. Publicar el mismo hecho en dos formas o en dos destinos durante una migración: ambos campos o ambos topics durante un cambio de esquema (Capítulo 12), ambos brokers durante una migración de brokers (Capítulo 14). En una migración de brokers, el relay mantiene una segunda marca para el nuevo destino, de modo que el broker antiguo siga completo y el nuevo pueda conciliarse con él.

duplicado. Un mensaje procesado o entregado más de una vez. Capítulo 2.

envelope. El conjunto estándar de campos que viaja con cada evento, como `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` y `causation_id`, separado del payload. Capítulo 11.

event journal. Un registro de solo anexo de envelopes de eventos, sin payloads, que un servicio mantiene junto a su inbox para poder responder preguntas después de que el broker haya olvidado los eventos. Capítulo 13.

event_id. El campo del envelope que identifica un evento. Se crea cuando se escribe la fila del outbox y nunca se regenera, y es la clave de idempotencia habitual para el consumo. Capítulos 4 y 7.

evento. Un mensaje cuyo contenido es un hecho que ya ocurrió, nombrado en pasado. Capítulo 11.

evento de integración. Un evento publicado como un contrato público deliberado, mapeado desde un evento de dominio y libre de datos personales y de secrets. Capítulo 11.

evento sintético. Tráfico etiquetado que una sonda inyecta en el pipeline activo y que se comprueba al final del flujo. Capítulo 15.

evolución de esquemas. Cambiar la forma de los eventos con el tiempo sin romper a los lectores y escritores que existen. Capítulo 12.

exactly-once. Se usa en el libro solo como un efecto cualificado: entrega at-least-once más un consumidor idempotente, que el libro también llama effectively-once. No es una garantía de entrega. El procesamiento exactly-once de un broker es real dentro de las fronteras de ese broker y termina en ellas. Capítulo 2.

expand and contract. Un patrón de migración que convierte un cambio incompatible en varios pasos compatibles: admitir ambas formas, migrar a todos los lectores y después retirar la forma antigua. Capítulo 12.

failover. Un cambio diseñado a otra copia de una partición de broker, de una base de datos o de un clúster cuando falla la activa. Puede perder lo que aún no se había replicado. Capítulos 3 y 14.

fat event. Un evento que lleva estado suficiente para que los consumidores actúen sin volver a llamar al productor. Lo contrario es un **thin event**. Capítulo 11.

grupo de consumidores. Un conjunto de consumidores que se reparten las particiones de un topic, de modo que cada partición la lee un miembro a la vez. Capítulo 2.

head-of-line blocking. Un registro que un consumidor no quiere saltarse bloquea todos los registros que hay detrás en la misma partición o stream. Capítulo 8.

hold-back buffer. Una tabla duradera donde un consumidor retiene un evento cuyo predecesor no ha llegado, y lo libera cuando el hueco se cierra o vence un timeout. Capítulo 9.

inbox. La tabla de ids de mensajes procesados en el lado del consumidor, escrita en la misma transacción que los efectos del handler. Capítulo 7.

jitter. Aleatoriedad añadida a la espera de un retry para que muchos clientes no reintenten al unísono. “Exponential backoff with jitter” es la expresión estándar. Capítulo 6.

link (de trace). Una referencia desde un span del consumidor al contexto de trace del productor, usada en lugar de una relación padre-hijo cuando la entrega es tardía, repetida o por lotes. Capítulo 13.

log de eventos. La secuencia duradera, de solo anexo, de eventos. Capítulo 2.

mensaje. Una unidad de datos enviada a través de un broker: un evento, un comando o un documento. Capítulo 11.

message broker. La infraestructura que transporta mensajes entre servicios. En el libro aparecen dos familias: basada en log y basada en colas. Capítulo 2.

mirroring. Copiar mensajes entre brokers con una herramienta: una herramienta de mirroring entre dos logs, o un bridge o conector entre un log y una cola. También transporta el historial anterior a una migración y puede mantener alimentado un clúster standby, y los offsets de la copia son propios. Capítulo 14.

modelo de fallos. La lista escrita de las uniones de un sistema y, para cada una, qué puede fallar, cómo se observaría y qué garantía sobrevive, con responsables, pruebas y fechas de revisión. Capítulo 3.

objetivo de punto de recuperación (RPO). Cuántos datos puedes permitirte perder, medido como tiempo. Un backup cada seis horas implica un RPO de hasta seis horas, y cada almacén tiene el suyo. Capítulo 14.

objetivo de tiempo de recuperación (RTO). Cuánto tiempo puedes estar sin el servicio. Para una proyección reproducida es un cálculo de drenaje. Capítulo 14.

offset. La posición de un consumidor en una partición de un log de eventos. Capítulo 2.

outbox. La tabla que se escribe en la misma transacción local que el cambio de estado, desde la cual un relay publica eventos. El término completo es **transactional outbox**. Capítulo 4.

outbox del consumidor. Un outbox escrito por un consumidor que publica un mensaje de seguimiento, en la misma transacción que su fila del inbox y su cambio de estado. Una caída antes del commit no deja nada, y una caída después se absorbe en el inbox en la reentrega. El Capítulo 7 introduce la regla; el Capítulo 10 la aplica a los pasos de una saga.

parking lot. Un lugar de espera para dead letters que necesitan una decisión humana, distinto de la propia DLQ. Capítulo 8.

partición. Una unidad de un topic basado en log que está ordenada y que lee un solo consumidor de un grupo a la vez. Capítulos 2 y 9.

paso pivot. El punto de una saga a partir del cual el flujo ya no compensa y solo avanza. Capítulo 10.

período de gracia. La antigüedad que debe alcanzar un registro antes de que un job de conciliación lo compare, para que los eventos aún en tránsito no se reporten como discrepancias. Capítulo 13.

poison message. Un mensaje que falla cada vez que se procesa. En un broker basado en log, uno que tumba el proceso deja al consumidor atascado en una posición, de modo que se detecta por el retraso sin throughput y por un contador por posición. Capítulo 8.

productor. Un proceso o servicio que publica mensajes, y el responsable de los eventos que emite. Capítulo 11.

rebalanceo. La reasignación de particiones cuando miembros se unen a un grupo de consumidores o lo abandonan, o cuando se juzga que un miembro no responde. Dos consumidores pueden trabajar brevemente sobre la misma partición. Capítulo 3.

redrive. Devolver los dead letters al consumidor que falló con ellos, a su propia entrada y nunca al topic compartido del que vinieron. Se conservan la clave y el `event_id`, el número de intentos empieza de nuevo y `redrive_count` registra la ronda. Capítulo 8.

reentrega. El broker entregando un mensaje de nuevo, por ejemplo tras un ack perdido o un visibility timeout vencido. El replay es otra cosa. Capítulo 2.

reinicio de offsets. Mover la posición confirmada de un grupo de consumidores, normalmente hacia atrás hasta una marca de tiempo con un margen, para que se vuelva a leer el historial. Los offsets pertenecen a un clúster y no se traducen a otro. Capítulo 14.

relay. El proceso que mueve las filas del outbox al broker. Capítulo 5.

replay. Releer deliberadamente el historial de un log de eventos. El Capítulo 2 lo define; los capítulos 7, 12, 13, 14 y 15 lo usan.

retención. Cuánto tiempo conserva un broker un mensaje, se haya leído o no: un límite de tiempo o de tamaño en un log, una caducidad en una cola. Acota el replay, el redrive y cuánto tiempo puede estar caído un consumidor retrasado. El Capítulo 2 la define; los capítulos 3, 13 y 14 la usan.

retraso del consumidor. Cuánto va un consumidor por detrás de la cabeza de una partición o cola, medido en mensajes o en tiempo. El Capítulo 2 lo define; los capítulos 3 y 13 lo usan.

retry budget. Un límite a los retries como proporción del tráfico total en lugar de por llamada. Capítulo 6.

retry storm. Un aluvión de retries que añade carga a una dependencia en apuros y la mantiene caída. Capítulo 6.

revisión del registry. El identificador que un schema registry asigna a cada esquema registrado. Avanza con cada cambio registrado, incluidos los aditivos, y es un número distinto de la **versión del esquema** y de la **versión del agregado**. Capítulo 12.

saga. Un flujo de larga duración formado por transacciones locales con compensaciones, ya sea orquestado por un coordinador o coreografiado mediante eventos. Capítulo 10.

schema registry. Un servicio que almacena esquemas, asigna identificadores y comprueba un esquema propuesto frente a los anteriores. Capítulo 12.

semántica de fallos. Para una unión dada, qué puede fallar, cómo lo observarías y qué garantía sobrevive. Es la tesis del libro. El Prefacio la enuncia, el Capítulo 3 la aplica como modelo de fallos y el Capítulo 16 la convierte en la lista de verificación de revisión.

tolerant reader. Un consumidor que toma de un evento solo los campos que necesita e ignora el resto. Capítulo 12.

tombstone. La clave, la última versión y una marca de borrado que un consumidor conserva tras borrar la fila de un agregado, para que un evento tardío no pueda recrearla. Capítulo 9.

topic. Un stream con nombre de mensajes en cualquiera de las dos familias de brokers. Capítulo 2.

upcaster. Código que convierte un evento escrito bajo un esquema antiguo al actual antes de que la lógica de negocio lo vea. Capítulo 12.

ventana de deduplicación. El período durante el cual un consumidor recuerda los ids procesados. Un mensaje cuya fila del inbox se ha borrado puede procesarse de nuevo. Capítulo 7.

versión del agregado. Un contador de los cambios de un agregado, almacenado en la fila del outbox y transportado en el payload, como `order_version` en Orders. Ordena los eventos de un mismo agregado y no dice nada sobre la forma del payload. Es un número distinto de la **versión del esquema** y de la **revisión del registry**. El Capítulo 4 la define; el Capítulo 9 la usa.

versión del esquema. El entero `schema_version` del envelope, que el Capítulo 12 también llama la versión incompatible. Solo avanza con un cambio que un lector antiguo no puede manejar, y es un número distinto de la **revisión del registry** y de la **versión del agregado**. Capítulo 12.

visibility timeout. En un broker basado en colas, el tiempo que un mensaje entregado permanece oculto antes de reentregarse si no se ha confirmado. Capítulo 2.

B. Referencia rápida de patrones

Cada fila nombra el fallo que aborda el patrón, en los términos del libro, y lo que te cuesta. Un patrón mueve un fallo a un lugar donde puedes manejarlo; no lo elimina. El Capítulo 16 convierte estas filas en una lista de verificación de revisión.

Patrón	Fallo que aborda	Costos	Capítulo
Transactional outbox	Dual write entre una base de datos y un broker	Una tabla extra, un relay que operar, duplicados trasladados a los consumidores	4
Relay con polling	Mover las filas del outbox al broker con poca infraestructura	Retraso del polling, carga sobre la base de datos, diseño cuidadoso del cursor	5
Change data capture	Publicar las filas del outbox desde el log de la base de datos con poco retraso	Peso operativo, failover del conector, retención del log que gestionar	5

Patrón	Fallo que aborda	Costos	Ca- pítu- lo
Exponential backoff con jitter	Retries que sobrecargan una dependencia que se está recuperando	Latencia añadida, ajuste de las esperas y de los límites de intentos	6
Retry budget	Retries que amplifican una interrupción	Algunas llamadas fallan rápido cuando podrían haber tenido éxito	6
Backpressure y control de flujo	Una dependencia o un consumidor al que se da más trabajo del que puede absorber	Retraso, consumidores pausados y trabajo rechazado, una decisión de negocio sobre qué espera	6
Circuit breaker	Llamadas repetidas a una dependencia que falla	Estado que ajustar; llamadas rechazadas mientras el breaker está abierto	6
Retry topic o cola de retraso	Esperas de retry demasiado largas para aguardarlas dentro del handler	Más topics, más piezas móviles, orden más difícil	6
Consumidor idempotente con inbox	Duplicados de la entrega at-least-once	Escrituras en tabla por cada mensaje, retención que gestionar	7
Clave de negocio	El mismo trabajo llegando en mensajes distintos	Diseño de la clave por operación, y restricciones únicas	7
Clave de idempotencia pasada aguas abajo	Efectos secundarios externos duplicados	Necesita soporte del servicio aguas abajo	7
Dead-letter queue	Poison messages atascados en un bucle de reentrega	Un montón que necesita responsable, alarmas y una vía de reparación	8
Parking lot	Dead letters que necesitan una decisión humana	Trabajo manual y una regla clara de responsabilidad	8
Redrive	Recuperar dead letters una vez corregida la causa	Duplicados, reordenamientos y carga sobre el flujo principal	8
Outbox del consumidor	Un consumidor que cambia estado y publica un seguimiento, perdiendo uno de los dos	Un outbox en el lado del consumidor y un relay que operar	7, 10
Clave de partición por agregado	Eventos desordenados dentro de un mismo agregado	Particiones calientes, límites al paralelismo	9

Patrón	Fallo que aborda	Costos	Ca- pítu- lo
Comprobación de versión en el consumidor	Eventos obsoletos o desordenados que sobrescriben estado más reciente	Un campo de versión y una decisión para cada hueco	9
Hold-back buffer	Huecos en una secuencia que debe aplicarse en orden	Memoria, timeouts y el hueco que nunca se cierra	9
Saga (orquestrada o coreografiada)	Un flujo de negocio que abarca servicios sin una sola transacción	Estados, timeouts y compensaciones que diseñar y operar	10
Compensación	Deshacer pasos de saga ya completados tras un fallo posterior	Algunos pasos no pueden deshacerse; el paso pivot marca el límite	10
Bloqueo semántico	Cambios concurrentes sobre un registro en el que trabaja una saga	Operaciones bloqueadas mientras la saga se ejecuta	10
Tipos en el cable fijados por campo entre lenguajes	El mismo campo del envelope leído de forma distinta por dos runtimes	Disciplina en cada lenguaje y pruebas que cruzan lenguajes	11
Envelope de evento estándar	Eventos que las herramientas y los consumidores no pueden rastrear ni encaminar	Disciplina en todos los productores	11
Thin event o fat event	Elegir entre callbacks y estado transportado	Thin: acoplamiento en el momento de la lectura. Fat: contrato más grande y datos copiados	11
Schema registry con comprobaciones de compatibilidad	Productores que rompen consumidores en silencio	Un servicio que operar y un proceso que seguir	12
Tolerant reader y upcaster	Lectores que se rompen con campos nuevos o versiones antiguas	Código de conversión conservado para cada versión histórica	12
Expand and contract	Un cambio incompatible que no se puede evitar	Varios deploys, dual-publish, una migración más larga	12
Contexto de trace y correlation id	Perder la historia de un flujo a través de saltos asíncronos	Propagación en cada productor y consumidor	13
Conciliación y reparación	Divergencia silenciosa entre un sistema de origen y el estado derivado	Jobs de comparación, herramientas de reparación y revisión de los resultados	13

Patrón	Fallo que aborda	Costos	Ca- pítu- lo
Event journal	Preguntas sobre eventos que el broker ha olvidado	Una tabla escrita con cada mensaje y retención que gestionar	13
Backfill	Estado derivado que debe re-construirse o completarse a partir de datos existentes	Lotes con límite de ritmo y un interruptor de pausa, y la carga sobre un consumidor activo	13
Comprobaciones de invariantes	Roturas a nivel de negocio que ninguna métrica técnica muestra	Definir los invariantes y ser responsable de las alertas	13
Replay con reinicio de offsets	Un consumidor que se equivocó, o un almacén que retrocedió en el tiempo, y necesita el historial otra vez	Una ventana de deduplicación que cubra el rango, efectos secundarios suprimidos, y la retención como límite	14
Objetivos de punto y de tiempo de recuperación por almacén	Desastres manejados con improvisación, sin pérdida ni tiempo de inactividad acordados	Un número que medir y ensayar para cada almacén	14
Replicación entre clústeres para la recuperación	Pérdida de un clúster de brokers o de una región	Un standby que operar, un retraso de replicación que es el RPO y offsets que no se traducen	14
Dual-publish y cutover	Pasar a otro broker sin perder eventos ni duplicar efectos	Una segunda marca en el out-box, conciliación y una migración más lenta	14
Pruebas de contrato dirigidas por el consumidor	Deriva del contrato entre equipos	Mantenimiento de las pruebas y acuerdo sobre quién las ejecuta	15
Pruebas de handlers y de perturbación	Handlers nunca ejercitados con duplicados, retrasos o reordenamientos	Infraestructura de pruebas con base de datos real y suites más lentas	15
Inyección de fallos	Entradas del modelo de fallos nunca ejercitadas en un sistema en ejecución	Un entorno dedicado, un radio de impacto que acotar	15
Prueba de regresión basada en replay	Cambios que rompen el comportamiento sobre historial real	Datos de muestra curados y anonimizados que mantener	15
Eventos sintéticos en producción	Fallos silenciosos en la ruta activa	Datos de prueba que deben aislarse de los datos de negocio reales	15

Cómo leer la tabla

Muchas filas vienen en pares: un patrón que arregla un fallo y un segundo patrón que arregla el fallo que introdujo el primero. El outbox crea duplicados, así que el consumidor necesita un inbox. Los retries crean carga, así que necesitan un presupuesto. Una dead-letter queue crea un montón, así que necesita un responsable y un procedimiento de redrive.

Esa cadena es la forma del libro. Si adoptas una fila y te saltas su pareja, el fallo se ha movido pero no ha desaparecido. A mi juicio, la tabla es más útil leída de derecha a izquierda: parte del fallo que tu sistema tiene hoy, encuentra la fila que lo nombra y lee la columna de costos antes de leer el capítulo.

SOBRE EL LIBRO

Los sistemas confiables orientados a eventos se construyen en torno a la semántica de fallos, no a los message brokers.

En cada punto de unión, donde un componente entrega trabajo a otro, el libro plantea las mismas tres preguntas: qué puede fallar, cómo lo observarías y qué garantía sobrevive al fallo. La entrega at-least-once se trata como un hecho y no como un defecto, así que los handlers son idempotentes, y los duplicados, los mensajes perdidos, los retries y la llegada desordenada se diseñan de antemano en lugar de esperar que desaparezcan. Siguiendo a Parcellly, el libro recorre el transaccional outbox, los relays y el change data capture, los retries, la idempotencia, las dead letters, el orden, las sagas, la evolución de esquemas, la observabilidad y la recuperación, cada uno con el costo que conlleva.

SOBRE EL AUTOR

Muhammet Şafak desarrolla software desde 2008. Durante muchos años ha desempeñado funciones activas en los sectores de la salud, las finanzas y las telecomunicaciones, y sostiene que el desarrollo debe centrarse en el problema, no en la tecnología. Desde 2024 diseña y desarrolla sistemas de alto tráfico escalables de extremo a extremo.

Brinda mentoría a equipos de desarrollo de software pequeños y medianos, dirige proyectos de código abierto y deja por escrito sus decisiones, con sus razones y su costo.

Los sistemas confiables no se construyen con piezas confiables. Se construyen alrededor de las piezas que fallan.

I
ACCESO

II
EVENTOS

III
AGENTES

