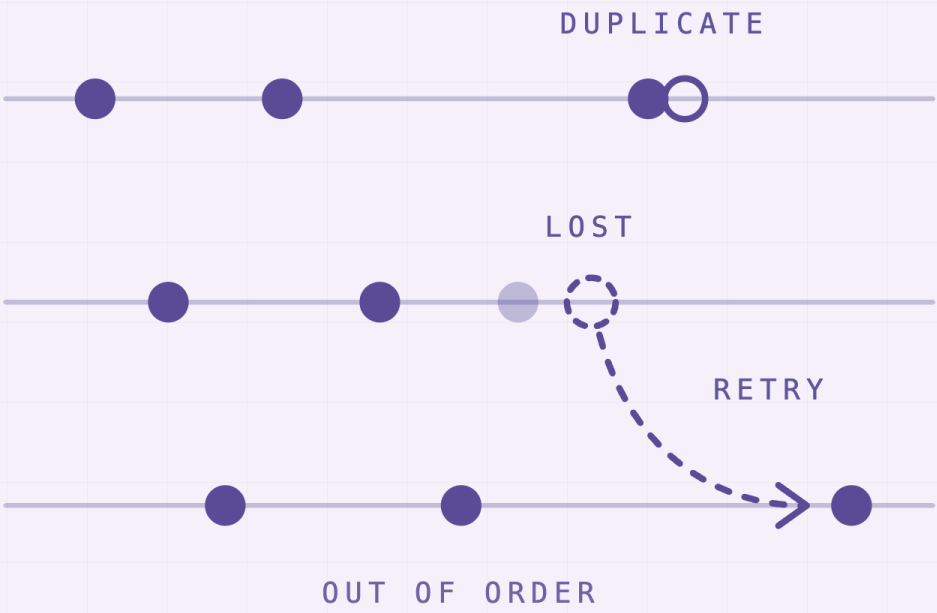
THE UNRELIABLE
SYSTEMS TRILOGY

II



Zuverlässige ereignisgesteuerte Systeme entwerfen

Muhammet Şafak

Zuverlässige ereignisgesteuerte Systeme entwerfen

Messaging, Retries, Idempotenz, Outbox, Nachrichtenreihenfolge und Schemaevolution

Muhammet Şafak

© 2026 Muhammet Şafak
Kostenlos herunterladen und lesen.

Inhalt

Vorwort 5

TEIL I — GRUNDLAGEN 13

Kapitel 1 — Die Lüge vom Happy Path 14

Kapitel 2 — Messaging-Semantik 27

Kapitel 3 — Ein pragmatisches Ausfallmodell 45

TEIL II — ZUVERLÄSSIG VERÖFFENTLICHEN 65

Kapitel 4 — Das Dual-Write-Problem und die Transactional Outbox 66

Kapitel 5 — Relays, Polling und Change Data Capture 87

TEIL III — ZUVERLÄSSIG KONSUMIEREN 110

Kapitel 6 — Retries richtig gemacht 111

Kapitel 7 — Idempotenz 138

Kapitel 8 — Poison Messages und Dead Letters 162

Kapitel 9 — Nachrichtenreihenfolge 187

TEIL IV — WORKFLOWS UND VERTRÄGE 210

Kapitel 10 — Sagas und langlaufende Workflows 211

Kapitel 11 — Ereignisdesign 235

Kapitel 12 — Schemaevolution 256

TEIL V — BETRIEB UND NACHWEIS 279

Kapitel 13 — Observability und Abgleich 280

Kapitel 14 — Wiederherstellung, Katastrophen und Broker-Migration 304

Kapitel 15 — Ereignisgesteuerte Systeme testen 332

Kapitel 16 — Eine Checkliste für Zuverlässigkeit und Schlussgedanken 350

Anhang: Glossar und Kurzreferenz der Patterns 364

Vorwort

Stellen Sie sich einen Dienstagnachmittag bei Parcelly vor, einem Marktplatz, den es nicht gibt. Der Checkout-Ablauf läuft seit Monaten stabil. Ein Käufer gibt eine Bestellung auf, der Service Orders schreibt sie in seine Datenbank, und ein Ereignis `OrderPlaced` geht an den Message-Broker. Payments autorisiert die Karte. Inventory reserviert den Bestand. Shipping plant das Paket ein. Jedes Dashboard ist grün.

Dann läuft ein Deployment durch Orders. Jede Instanz behandelt eine Bestellung auf dieselbe Weise: Sie schreibt die Bestellung per Commit in ihre Datenbank und veröffentlicht dann `OrderPlaced` beim Message-Broker. Für ein paar Sekunden werden Instanzen heruntergefahren, während andere starten. Eine Instanz führt den Commit einer Bestellung aus und wird gestoppt, bevor der Publish-Aufruf erfolgt. Die Bestellung ist sicher gespeichert, und das Ereignis verlässt das System nie. Nichts stürzt laut ab, und kein Alarm löst aus. Am nächsten Morgen sieht ein Käufer eine Bestellung, die als aufgegeben markiert ist, doch Payments hat nie eine Karte autorisiert, und Inventory hat nie einen einzigen Artikel reserviert. Orders glaubt, die Arbeit sei im Gange, und der Rest des Systems hat nie davon gehört.

Niemand hat einen Bug im üblichen Sinn geschrieben. Jede Codezeile tat, was sie sagte. Das System ist im Raum zwischen den Zeilen ausgefallen: zwischen einem Datenbank-Commit und einem Netzwerkaufruf, zwischen einer zugestellten und einer verstandenen Nachricht, zwischen „wir haben es gesendet“ und „sie haben es getan“.

Von diesem Raum handelt dieses Buch.

Warum es dieses Buch gibt

Die ereignisgesteuerte Architektur hat ein Imageproblem, und sie hat es sich von beiden Seiten verdient. Ihre Befürworter beschreiben lose gekoppelte Services, die unabhängig skalieren und sich ohne Abstimmung wei-

terentwickeln. Ihre Kritiker beschreiben verteiltes Spaghetti, bei dem niemand sagen kann, was passiert, wenn eine Bestellung aufgegeben wird. Beide beschreiben dasselbe System an unterschiedlichen Tagen.

Was die beiden Tage trennt, ist selten die Wahl des Brokers, des Frameworks oder auch nur des Architekturdiagramms. Es ist eine kurze Liste unspektakulärer Fragen. Was passiert, wenn diese Nachricht zweimal zugestellt wird? Was passiert, wenn sie nie ankommt? Was passiert, wenn sie nach der Nachricht ankommt, die ihr eigentlich folgen sollte? Was passiert, wenn der Consumer jedes einzelne Mal fehlschlägt? Was passiert, wenn der Producer ein Feld ändern muss und hundert Consumer noch den alten Code ausführen?

Diese Fragen sind die These dieses Buches. Zuverlässige ereignisgesteuerte Systeme werden um die **Ausfallsemantik** herum gebaut, nicht um Message-Broker: Für eine gegebene Nahtstelle geht es darum, was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall übersteht. Eine Nahtstelle ist jede Stelle, an der eine Komponente Arbeit an eine andere übergibt, etwa ein Datenbank-Commit mit anschließendem Publish oder eine Zustellung mit anschließendem Effekt. Die Patterns hier gibt es wegen des Netzwerks und der Datenbank, nicht wegen eines bestimmten Produkts, und ein Broker ist nur eine weitere Komponente mit eigenen Nahtstellen.

Teams treffen meist einzeln auf diese Fragen, in der Produktion, in der Reihenfolge, in der das System gerade ausfällt. Jede Antwort wird für sich gelernt, und die Erkenntnisse landen selten an einem Ort. Die Dokumentation eines Brokers sagt Ihnen, wie Sie veröffentlichen und konsumieren. Zu der Frage, wie ein korrekter Service rund um diese beiden Verben aussieht, schweigt sie weitgehend.

Dieses Buch versucht, dieser Ort zu sein. Es ist ein Leitfaden zu den Patterns, die ereignisgesteuerte Systeme zuverlässig machen: die Transactional Outbox, idempotente Consumer, Retries mit Backoff und Jitter, Backpressure, Dead-Letter-Queues, Strategien für die Reihenfolge, Sagas, sorgfältiges Ereignisdesign, Schemaevolution, Observability, Wiederherstellung nach Datenverlust und Broker-Wechsel sowie Tests. Ich habe versucht, jedes davon als Entscheidung mit einem Preis zu behandeln statt als Regel,

die es zu befolgen gilt. Ein Pattern, das für die Zahlungen von Parcelly richtig ist, kann für dessen Benachrichtigungen Verschwendung sein, und das Buch sagt es, wenn es darauf ankommt.

Ich möchte auch klar sagen, was das Buch nicht ist. Es ist kein Tutorial für einen bestimmten Broker. Es ist kein Katalog jedes jemals benannten Enterprise-Integration-Patterns. Es verspricht nicht, dass Ihr System korrekt wird, wenn Sie ihm folgen. Zuverlässigkeit ist keine Eigenschaft, die man hinzufügt; sie ist eine Menge von Ausfällen, für die Sie bewusst entschieden haben, wie Sie sie behandeln, und eine Menge, die Sie bewusst in Kauf nehmen. Mein Ziel ist es, diese Entscheidungen sichtbar zu machen.

Für wen dieses Buch ist

Das Buch richtet sich an **Backend- und Plattform-Entwickler**, die Services bauen oder betreiben, die über Ereignisse kommunizieren, und an die **Tech Leads**, die entscheiden müssen, wie sich diese Services verhalten sollen, wenn etwas schiefgeht. Wenn Sie schon einmal auf eine Dead-Letter-Queue gestarrt haben, die über Nacht gewachsen ist, darüber gestritten haben, ob ein Consumer idempotent sein muss, oder ein Topic geerbt haben, dessen Ereignisse niemand vollständig erklären kann, dann hatte ich Sie beim Schreiben vor Augen.

Sie müssen kein Messaging-Experte sein. Mit Folgendem sollten Sie vertraut sein:

- Relationale Datenbanken und Transaktionen: was ein Commit ist, was ein Unique Constraint bewirkt und was Isolation grob bedeutet.
- Services, die über ein Netzwerk kommunizieren, und das Grundvokabular von Timeouts sowie von Anfrage und Antwort im Stil von HTTP.
- Die allgemeine Vorstellung eines Message-Brokers: etwas, das Nachrichten von Producern annimmt und an Consumer weitergibt.

Sie brauchen keine Erfahrung mit einem bestimmten Produkt. Wenn ein Kapitel Kafka, RabbitMQ oder Amazon SQS nennt, dann als Beispiel für eine Familie von Systemen, und es beschreibt Verhalten, das allgemein gilt und nicht nur in einem bestimmten Release. Haben Sie keines davon ge-

nutzt, bleibt das Buch trotzdem verständlich. Kennen Sie eines gut, werden Sie feststellen, dass manches von dem, was Sie wissen, nur für dieses gilt, und das Buch versucht zu kennzeichnen, welche Teile das sind.

Wenn Sie Tech Lead sind und nicht jeden Tag Code schreiben, können Sie in jedem Kapitel das einleitende Szenario, den Abschnitt zu den Abwägungen und die zentralen Erkenntnisse lesen und haben damit den größten Teil des Nutzens für Ihre Entscheidungen. Der Pseudocode ist für alle da, die genau sehen müssen, wo die Transaktionsgrenze liegt.

Wie das Buch aufgebaut ist

Das Buch hat fünf Teile. Beim ersten Mal sollen sie der Reihe nach gelesen werden, danach lassen sie sich in beliebiger Reihenfolge nutzen.

Teil I, Grundlagen (Kapitel 1 bis 3). Wir beginnen damit, den Happy Path auseinanderzunehmen. Kapitel 1 zeigt, wie ein Ablauf, der einfach aussieht, ein Dutzend Entscheidungen verbirgt. Kapitel 2 legt fest, was Messaging-Garantien tatsächlich bedeuten und warum At-least-once-Zustellung die realistische Basis ist. Kapitel 3 baut ein pragmatisches Ausfallmodell auf, ein kleines Vokabular für die Arten, wie Dinge kaputtgehen, von Timeouts und Abstürzen bis zu Nachrichten, die immer fehlschlagen, und dem Verlust eines ganzen Stores. Der Rest des Buches nutzt es als gemeinsame Sprache.

Teil II, Zuverlässig veröffentlichen (Kapitel 4 und 5). Kapitel 4 behandelt das Dual-Write-Problem, genau die Lücke in der Eingangsszene, und führt die Transactional Outbox ein. Kapitel 5 behandelt die Mechanik, die Outbox-Zeilen zum Broker bringt, vergleicht Polling-Relays mit Change Data Capture und zeigt, was beides Sie kostet.

Teil III, Zuverlässig konsumieren (Kapitel 6 bis 9). Das ist der längste Teil, denn bei Consumern entstehen die meisten doppelten Effekte, festhängenden Warteschlangen und verlorenen Updates. Kapitel 6 behandelt Retries: wann man einen Retry durchführt, wann nicht, warum Backoff Jitter braucht und wie man verhindert, dass ein Consumer von seinen eigenen Retries überlastet wird. Kapitel 7 handelt von Idempotenz und der Inbox. Kapitel 8 befasst sich mit Poison Messages und Dead Letters. Kapitel 9 wendet sich der Reihenfolge zu und der Frage, wie wenig davon man sich in der Regel leisten kann einzufordern.

Teil IV, Workflows und Verträge (Kapitel 10 bis 12). Kapitel 10 führt Sagas ein, für Workflows, die mehrere Services umspannen und keine Transaktion teilen können. Kapitel 11 befasst sich mit dem Entwurf der Ereignisse selbst: was hineingehört, was draußen bleibt, wie sie benannt werden und wie ein Nachrichtenumschlag übersteht, in mehreren Sprachen geschrieben und gelesen zu werden. Kapitel 12 behandelt die Schemaevolution, die stille Disziplin, mit der Producer und Consumer sich unterschiedlich schnell ändern können.

Teil V, Betrieb und Nachweis (Kapitel 13 bis 16). Kapitel 13 handelt davon, zu sehen, was Ihr System tut, und vom Abgleich, der Gewohnheit, die Sicht eines Systems mit der eines anderen zu vergleichen. Kapitel 14 handelt von der Wiederherstellung nach verlorenen Daten, dem Verlust einer Region und einem Broker-Wechsel. Kapitel 15 behandelt, wie man Code testet, dessen schlimmste Bugs erst bei Duplizierung, Verzögerung und Umordnung auftreten. Kapitel 16 fasst alles in einer Checkliste für Zuverlässigkeit zusammen und schließt das Buch.

Ein Anhang enthält ein Glossar und eine Kurzreferenz der Patterns, damit Sie etwas nachschlagen können, ohne ein Kapitel erneut zu lesen.

Jedes Kapitel von 1 bis 15 hat dieselbe Form. Es beginnt mit einem kurzen Szenario, erklärt, warum das Problem schwierig ist, stellt das Pattern vor und geht dann zu zwei Abschnitten über, die ich für die wertvollsten halte: **Abwägungen und Kosten**, wo steht, wann man das Pattern nicht einsetzen sollte, und **Häufige Fehler**. Jedes Kapitel endet mit fünf bis sieben zentralen Erkenntnissen. Wenn Ihnen die Zeit knapp ist, bringen diese drei Teile Sie weiter, als Sie vielleicht erwarten.

Sie können die Kapitel selektiv lesen. Wenn Sie heute vor allem eine wachsende Dead-Letter-Queue plagt, gehen Sie direkt zu Kapitel 8 und folgen Sie den Verweisen zurück auf die Kapitel 6 und 7, soweit nötig. Wenn Sie einen neuen Service entwerfen, würde ich Teil II lesen, bevor Sie die erste Zeile Publishing-Code schreiben, denn mit der Outbox ist es viel leichter anzufangen, als sie nachzurüsten.

Lernen Sie Parcelly kennen

Durch das ganze Buch trägt ein fiktives System die Beispiele: **Parcelly**, ein Online-Marktplatz, auf dem unabhängige Verkäufer Waren anbieten und Käufer sie bestellen. Es hat sechs Services.

- **Orders** besitzt den Lebenszyklus der Bestellung und ist der Ort, an dem die meisten Workflows beginnen.
- **Payments** autorisiert und zieht Geld ein und spricht mit einem externen Zahlungsanbieter.
- **Inventory** verfolgt Bestände und hält befristete Reservierungen.
- **Shipping** plant Sendungen und übergibt Pakete an Paketdienste.
- **Notifications** versendet E-Mails, Push-Nachrichten und SMS und darf nie etwas anderes blockieren.
- **Ledger** führt eine Append-only-Aufzeichnung der Geldbewegungen in doppelter Buchführung.

Jeder Service besitzt seine eigene relationale Datenbank und liest keine fremde. Die Services kommunizieren über einen Message-Broker, indem sie Ereignisse mit Namen wie `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` und `ShipmentScheduled` austauschen. Die Ereignisse folgen einer einheitlichen Form, einem Nachrichtenumschlag mit einer Kennung, einem Typ, einer Schemaversion, Korrelationsfeldern und einem Payload, damit die Kapitel in einer Stimme darüber sprechen können.

Ich habe bewusst einen Marktplatz gewählt. Geld einziehen, Waren reservieren und Pakete bewegen ist eine Domäne, in der man die Kosten einer verlorenen Nachricht oder eines doppelten Effekts schnell spürt. Eine doppelte E-Mail ist ärgerlich. Ein doppelter Einzug bedeutet eine Erstattung, ein Support-Ticket und eine Entschuldigung. Eine verlorene Reservierung ist ein überverkaufter Artikel. Dieses Gefälle, von harmlos bis teuer, ist genau das, was Sie brauchen, um zu entscheiden, wie viel Zuverlässigkeitsmechanik jeder Ablauf verdient.

Parcelly ist Fiktion, und seine Incidents ebenso. Die Szenarien, mit denen jedes Kapitel beginnt, sind erfunden, um die Form eines Ausfalls zu veranschaulichen, und sie werden stets als Fiktion gerahmt. Für Urteile verwende ich die erste Person, etwa „Ich würde hier einen Unique Constraint hin-

zufügen“ oder „Meiner Ansicht nach lohnt sich die Komplexität nicht“, aber ich schmücke Überlegungen nicht als Kriegsgeschichten aus. Sie sollen sich darauf verlassen können, dass es, wenn das Buch sagt, etwas geschehe, an der Funktionsweise dieser Systeme liegt und nicht an einer Anekdote.

Das Buch ist außerdem bewusst **unabhängig vom Broker**. Wo ein Punkt davon abhängt, ob Ihr Broker logbasiert ist, wobei Consumer einen Offset in einer geordneten Partition verfolgen, oder warteschlangenbasiert, wobei Nachrichten einzeln bestätigt und erneut zugestellt werden, sage ich, welcher Fall gemeint ist. Wo es keine Rolle spielt, tue ich nicht so, als wäre es anders.

Wie Sie den Code lesen

Der Code in diesem Buch ist **Pseudocode**. Er gehört zu keiner Sprache, er lässt sich nicht kompilieren, und nichts darin ruft eine echte Bibliothek auf. Das ist Absicht. Jedes Snippet soll zeigen, wo eine Transaktion beginnt und endet, wo eine Nachricht bestätigt wird und was passiert, wenn ein Schritt auf halbem Weg fehlschlägt. Diese Tatsachen sind in jeder Sprache gleich, und Syntax würde ihnen nur im Weg stehen.

Die Blöcke sind kurz, höchstens 30 Zeilen. Wo ein Tabellenschema hilft, steht es in SQL-ähnlicher Notation. Diagramme verwenden Mermaid, und jedes wird durch einen Satz eingeleitet, der sagt, worauf Sie achten sollen. Wenn Sie ein Pattern in Ihren eigenen Stack übertragen, behandeln Sie den Pseudocode als Spezifikation von Reihenfolge und Atomarität, nicht als Implementierung.

Ein Hinweis zur Gewissheit

Ich werde in diesem Buch Empfehlungen aussprechen, und manche vertrete ich entschieden. Ich würde zum Beispiel keine Ereignisse mit einem schlichten Dual Write veröffentlichen, wenn der Verlust eines Ereignisses für das Geschäft zählt. Andere Empfehlungen sind Geschmacks- oder Kontextfragen, und ich versuche zu sagen, welche welche sind.

Es wird auch Stellen geben, an denen die ehrliche Antwort lautet: „Es kommt darauf an, und zwar auf Folgendes.“ Retry-Budgets, die Aufbewahrung von Dead Letters, Partitionszahlen und das richtige Maß an Reihenfolge hängen alle von Tatsachen über Ihr Geschäft und Ihr Team ab, die ich nicht sehen kann. An diesen Stellen nenne ich die Variablen, statt eine Zahl zu erfinden. Wenn Sie ein Kapitel beenden und wissen, welche Fragen Sie in Ihrem eigenen System stellen müssen, werte ich das als Erfolg.

Arbeit an der Zuverlässigkeit hat eine unglückliche Eigenschaft: Wenn sie gelingt, geschieht nichts Sichtbares. Bestellungen werden erfüllt, die Hauptbücher gehen auf, und niemand schreibt Ihnen eine Nachricht, um sich zu bedanken. Ich hoffe, dieses Buch hilft Ihnen, diese Art von Ruhe zu schaffen.

Beginnen wir mit dem Happy Path und mit den Gründen, ihm nicht zu trauen.

TEIL I

Grundlagen

Kapitel 1 — Die Lüge vom Happy Path

Stellen Sie sich Parcelly in seinem ersten Jahr vor. Orders ruft Payments über HTTP auf, wartet auf die Antwort, ruft dann Inventory auf, wartet wieder, ruft dann Shipping auf. Der Code liest sich von oben nach unten, ein Ausfall ist eine Exception, und ein Käufer, der eine Fehlerseite sieht, kann es erneut versuchen. Es ist langsam, wenn der Zahlungsanbieter langsam ist, und es kippt um, wenn Inventory nicht erreichbar ist, aber niemand ist darüber verwirrt, was geschehen ist.

Dann beschließt das Team, auf Ereignisse umzusteigen. Orders veröffentlicht `OrderPlaced` bei einem Message-Broker, und die anderen Services reagieren in ihrem eigenen Takt. Die Demo ist wunderschön. Ein Käufer klickt auf „Bestellung aufgeben“, und innerhalb einer Sekunde ist der Bestand reserviert, die Zahlung eingezogen, die Bestellung bestätigt und eine Sendung eingeplant. Alle applaudieren.

Stellen Sie sich nun den ersten Monat in der Produktion vor. Ein Käufer wird belastet, aber die Bestellung bleibt für immer in „pending“. Eine andere Bestellung wird zweimal versandt. Eine dritte hat einen Ledger-Eintrag ohne passende Zahlung, und niemand kann sagen, welcher Service recht hat. Der Support fragt: „Was ist mit Bestellung 81724 passiert?“, und die ehrliche Antwort lautet, dass vier Services je ein Stück davon kennen und keiner alles.

An der Demo war nichts falsch. Sie war unvollständig, denn eine Demo ist eine Geschichte, in der jeder Schritt gelingt. Die Frage, die dieses Kapitel beantwortet, lautet, was das Team aufgegeben hat, als es Request-Response verließ, und was „zuverlässig“ bedeuten muss, bevor jemand anfängt, es wieder aufzubauen.

Das Problem und warum es schwierig ist

Der Happy Path ist die Abfolge von Ereignissen, die eintritt, wenn jede Komponente läuft, jeder Netzwerkaufruf ankommt, jede Nachricht einmal und in Reihenfolge zugestellt wird und jeder Consumer zu Ende bringt, was er beginnt. Es ist der Pfad, den Sie auf ein Whiteboard zeichnen, und der Pfad, den eine Demo durchläuft. Er ist auch der seltenste Pfad in einem System, das monatelang läuft.

Die Ausfälle sind nicht exotisch. Deployments starten Prozesse neu. Verbindungen brechen mitten in einem Schreibvorgang ab. Ein Broker wechselt per Failover auf ein Replikat. Ein Consumer braucht länger als erwartet, und seine Nachricht wird an jemand anderen übergeben. Jedes dieser Ereignisse tritt in einem ausgelasteten System täglich irgendwo auf. Die übliche Form eines schlechten Produktionsmonats ist, dass eine Handvoll gewöhnlicher Ereignisse auf Code trifft, der für den Happy Path geschrieben wurde.

Schwierig wird es dadurch, dass ereignisgesteuerte Systeme das beseitigen, was Ausfälle sichtbar machte. Bei einem synchronen Aufruf wartet der Aufrufer, und das Ergebnis kommt über dieselbe Verbindung zurück: ein Resultat oder ein Fehler, direkt neben dem Code, der ihn verursacht hat. Bei einem ereignisgesteuerten Entwurf übergibt der Producer eine Nachricht an den Broker und macht weiter. Geht drei Services und zehn Sekunden später etwas schief, ist der Producer längst weg, und die Belege verteilen sich auf Logs, die noch niemand korreliert hat.

Die naheliegende Reaktion ist, jedes Teil zuverlässig zu machen: ein guter Broker, gut getestete Services, Retries bei jedem Aufruf. Jedes davon hilft, und keines genügt, denn die Ausfälle, die wehtun, sitzen in den Lücken zwischen den Komponenten. Eine Nachricht, die in die Datenbank geschrieben, aber nie veröffentlicht wird, oder die verarbeitet, aber nie bestätigt wird, ist kein Bug in einer der beiden Komponenten. Sie ist eine Eigenschaft der Nahtstelle. Der größte Teil dieses Buches handelt von Nahtstellen.

Für jede Nahtstelle stellt das Buch dieselben drei Fragen: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall übersteht. Die Antworten nenne ich die **Ausfallsemantik** der Nahtstelle. Die Dokumentation eines Brokers liefert Ihnen die Semantik der eigenen

Nahtstellen des Brokers. Die Nahtstellen um ihn herum, zwischen Ihrer Datenbank und Ihrem Publish-Aufruf oder zwischen einer Zustellung und ihrem Effekt, müssen Sie selbst herausarbeiten, und Kapitel 3 zeigt, wie.

Synchrone und asynchrone Kopplung

Der Umstieg auf Ereignisse wird meist als „Entkopplung“ beschrieben, ein Wort, das mehr verspricht, als es hält. Genauer ist: Die Kopplung ändert ihre Form.

Ein synchroner Aufruf koppelt zwei Services in **Zeit**, **Verfügbarkeit** und **Wissen**. Der Aufgerufene muss jetzt laufen, die Latenz des Aufrufers schließt die des Aufgerufenen ein, und der Aufrufer weiß, wer der Aufgerufene ist und was er antworten wird. Der Vorteil ist ein einfaches Ausfallmodell: Der Aufrufer weiß, dass der Aufruf fehlgeschlagen ist, und kann sofort entscheiden, ob er einen Retry durchführt, einen Fallback nutzt oder einen Fehler meldet. Der Preis ist, dass die Kette nur so verfügbar ist wie ihr schwächstes Glied. Eine langsame Paketdienst-API wird zu einem langsamen Checkout.

Eine Nachricht über einen Broker beseitigt die Kopplung in Zeit und Verfügbarkeit. Orders kann eine Bestellung annehmen, während Inventory neu startet, weil der Broker die Nachricht hält, bis Inventory zurückkehrt. Ein neuer Consumer, etwa ein Analytics-Service, kann zu lauschen beginnen, ohne dass sich Orders ändert. Dieser Gewinn ist real, und er ist der Grund, diesen Stil einzusetzen.

Aber die Kopplung ist nur gewandert, nicht verschwunden. Die Services sind nun über den **Vertrag des Ereignisses** gekoppelt: seinen Namen, seine Felder und seine Bedeutung. Sie sind über das **Verhalten des Brokers** gekoppelt: was er hinsichtlich Dauerhaftigkeit, Reihenfolge und Zustellung verspricht. Und sie sind über Annahmen zum Timing des jeweils anderen gekoppelt, die niemand aufgeschrieben hat. Kapitel 11 und Kapitel 12 behandeln den Vertrag, Kapitel 2 den Broker.

Kurz gesagt: Synchrone Systeme fallen laut und in einer Kette aus. Asynchrone Systeme fallen leise und in Stücken aus.

Der Preis der zeitlichen Entkopplung

Zeitliche Entkopplung bedeutet, dass Producer und Consumer nicht zur selben Zeit aktiv sein müssen. Sie ist das Wichtigste, was man mit Ereignissen kauft, daher lohnt es sich, den Preis genau zu benennen.

Der Producer erfährt das Ergebnis nicht mehr

Als Orders Payments noch synchron aufrief, erfuhr es, ob die Zahlung funktioniert hatte, bevor es dem Käufer antwortete. Mit Ereignissen veröffentlicht Orders `OrderPlaced` und antwortet „empfangen“. Ob die Zahlung gelingt, ist eine Frage für später, und die Antwort kommt, falls sie kommt, als weiteres Ereignis. Die Bestellung hat nun einen Zwischenzustand, „aufgegeben, aber noch nicht bestätigt“, und das System muss entscheiden, was der Käufer sieht, solange er andauert, und was geschieht, wenn er länger dauert als geplant.

Niemand besitzt den ganzen Ablauf

In der synchronen Version enthielt eine Funktion die Abfolge, und man konnte den Geschäftsprozess an einer Stelle lesen. In der ereignisgesteuerten Version ist der Prozess die Summe von Reaktionen, die über Services verstreut sind. Payments reagiert auf `OrderPlaced`, Orders reagiert auf `PaymentAuthorized` und `StockReserved`, Shipping reagiert auf `OrderConfirmed`. Der Ablauf existiert nur als emergente Eigenschaft, und niemand kann zeigen, wo er definiert ist.

Das ist in Ordnung, solange alles funktioniert, und teuer, wenn etwas nicht funktioniert, denn für „Wo ist diese Bestellung hängengeblieben?“ gibt es keine einzelne Stelle zum Nachsehen. Kapitel 10 kommt mit Sagas darauf zurück, und Kapitel 13 behandelt, was Sie beobachten müssen, um die Frage überhaupt beantworten zu können.

Ausfälle sind keine Ausnahmen mehr

In einer Aufrufkette wandert ein Fehler den Stack hinauf, bis etwas ihn behandelt. In einem Nachrichtenfluss gibt es keinen Stack. Ein fehlschlagender Consumer kann die Nachricht bestätigen und verlieren, sie ablehnen und erneut zustellen lassen oder sie für einen Menschen parken. Jedes davon ist eine Richtlinienentscheidung mit Folgen für Services, die den Fehler nie sehen. Meiner Ansicht nach ist Fehlerbehandlung in einem synchro-

nen System eine Frage des Kontrollflusses und in einem asynchronen System eine Frage des Entwurfs, weil sich die Behandlung über Zeit, Prozesse und manchmal Teams verteilt.

Zeit wird zur Variablen

Zwischen „veröffentlicht“ und „verarbeitet“ liegt eine Lücke ohne Obergrenze, solange man ihr keine gibt. In der Lücke kann ein Produkt ausverkauft sein, ein Käufer kann stornieren, ein Preis kann sich ändern. Ein Consumer muss entscheiden, ob ein Ereignis, das bei der Veröffentlichung wahr war, bei der Verarbeitung noch von Bedeutung ist.

> **Hinweis:** Deshalb werden Ereignisse in der Vergangenheitsform benannt. `OrderPlaced` ist eine Tatsache über einen Moment, der vorbei ist. Ein Consumer, der es als Aufforderung behandelt, sofort zu handeln, hat stillschweigend angenommen, dass keine Zeit vergangen ist. Ein Command, etwa `CapturePayment`, fordert einen Service auf, etwas zu tun, und kann abgelehnt werden; ein Ereignis berichtet, dass etwas bereits geschehen ist. Kapitel 11 entwickelt die Unterscheidung.

Was „irgendwann“ wirklich bedeutet

Die meisten Erklärungen von **Eventual Consistency** bestehen aus einem Satz: Wenn keine neuen Updates erfolgen, konvergieren alle Replikate auf denselben Wert. Das ist zutreffend und für jemanden, der einen Checkout-Ablauf baut, nahezu nutzlos, weil es offenlässt, wie lange „irgendwann“ dauert und was man tut, während man wartet.

„Irgendwann“ ist keine Dauer

Ein System, das in 200 Millisekunden konvergiert, und ein System, das nach einem nächtlichen Batch konvergiert, weisen beide Eventual Consistency auf. Für einen Käufer, der auf eine Bestätigungsseite starrt, sind das verschiedene Produkte. Ich würde für jeden relevanten Hop eine Zahl verlangen, und sei sie grob. Nehmen wir an, Orders muss binnen weniger Sekunden nach dem Einzug eine bestätigte Bestellung anzeigen, und Notifications darf eine Minute brauchen. Diese Zahlen sind Illustrationen, und Ihre werden anders aussehen, aber sie machen aus „irgendwann“ eine An-

forderung, die sich testen lässt und auf die sich ein Alarm setzen lässt. Eine Verzögerung ohne Grenze ist keine Entwurfsentscheidung. Sie ist das Fehlen einer solchen.

„Irgendwann“ ist keine Garantie

Konvergenz tritt nur ein, wenn jede relevante Nachricht irgendwann zugestellt wird, jeder Consumer sie verarbeitet und kein Schritt sie stillschweigend verwirft. Geht eine Nachricht zwischen einem Datenbank-Commit und einem Broker-Publish verloren, konvergieren die Sichten nicht. Sie bleiben verschieden, bis ein Mensch oder ein Reparatur-Job es bemerkt. „Irgendwann“ ist ein bedingtes Versprechen, und die Bedingung lautet, dass das System Mechanismen besitzt, die Verlust erkennbar und reparierbar machen. Die Wahl eines Brokers liefert sie nicht. Teil II und Teil III dieses Buches verhindern den Großteil der Verluste, und Kapitel 13 erkennt und repariert den Rest.

Was man tut, während man wartet

Im Fenster der Inkonsistenz sehen andere Leser einen Zwischenzustand. Betrachten Sie diese Momentaufnahme einer Bestellung, einen Augenblick nachdem der Käufer belastet wurde:

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

Jeder Service hat mit seinen eigenen Daten recht. Die Kombination wirkt inkonsistent und ist für ein paar Sekunden normal. Zum Problem wird sie erst, wenn sie so bleibt. Die nützliche betriebliche Frage lautet nicht „Sind die Services gerade konsistent?“, sondern „Wie lange befindet sich diese Bestellung schon in diesem Zustand, und ist das länger, als wir erlauben?“

Eine Arbeitsdefinition von Zuverlässigkeit

„Zuverlässig“ wird so lose verwendet, dass zwei Entwickler sich einig sein können, ein System solle zuverlässig sein, und Gegensätzliches meinen. Ich möchte eine Definition, die sich prüfen lässt. Für ereignisgesteuerte Systeme verwende ich vier Eigenschaften. Sie sagen, was geschützt wird; die drei Fragen von oben sind die Art, wie Sie eine Nahtstelle gegen sie prüfen.

Kein Verlust. Jedes Ereignis, das eine geschäftliche Tatsache darstellt, erreicht jeden Consumer, der es braucht. Verlust ist der schädlichste Ausfall, weil er still ist: Das System läuft weiter, und das einzige Symptom ist eine Abwesenheit, die niemand bemerkt, bis ein Käufer sich beschwert. Kein Verlust ist eine Eigenschaft des gesamten Pfades, nicht des Brokers. Ein dauerhafter Broker hilft nicht, wenn es dem Producer nie gelang, die Nachricht zu speichern, oder wenn der Consumer bestätigte, bevor er die Arbeit beendet hatte. Kapitel 2 untersucht, was Broker versprechen, und Kapitel 4 und Kapitel 5 behandeln die Producer-Seite.

Keine schädliche Duplizierung. Manche Nachrichten werden mehr als einmal zugestellt. Das ist kein Defekt eines bestimmten Brokers, sondern der Preis dafür, keine Nachrichten zu verlieren: Wenn ein Sender nicht weiß, ob eine Nachricht angekommen ist, besteht der sichere Zug darin, sie erneut zu senden. Ein zuverlässiges System versucht nicht, Duplikate im Transport zu verhindern. Es macht sie harmlos. Das Wort „schädlich“ ist wichtig. Zwei identische E-Mails sind ein Ärgernis, während ein zweiter Einzug derselben Zahlung ein finanzieller Incident ist. Kapitel 7 behandelt Idempotenz, das wichtigste Mittel.

Begrenzte Verzögerung. Jedes Ereignis wird innerhalb einer Zeit verarbeitet, die das Geschäft toleriert, oder das System sagt laut, dass das nicht geschah. Eine Nachricht, die korrekt, aber drei Tage zu spät zugestellt wird, ist für viele Zwecke eine verlorene Nachricht. Begrenzt heißt nicht schnell. Es heißt, dass es eine Grenze gibt, dass jemand sie gewählt hat und dass ihre Verletzung einen Alarm auslöst. Eine Zahlungsbestätigung braucht vielleicht Sekunden; ein Wochenauszug verträgt vielleicht Stunden. Kapitel 6 behandelt Retries und Backoff, Kapitel 13 behandelt Lag und Alarme.

Erklärbarkeit. Für jede Bestellung kann zu jedem Zeitpunkt jemand rekonstruieren, was geschehen ist: welche Ereignisse erzeugt und konsumiert wurden, was jeder Service entschieden hat und warum. Das ist die unscheinbarste Eigenschaft, und sie entscheidet, ob sich die anderen drei einhalten lassen. Ein System, das eine Nachricht verliert, aber genau zeigen kann, wo, ist reparierbar. Ein System, das sich auf eine Weise falsch verhält, die niemand nachverfolgen kann, ist es nicht. Kapitel 13 behandelt die Kennungen, Aufzeichnungen und Vergleiche, die dafür nötig sind.

Die vier ziehen in verschiedene Richtungen, weshalb eine Definition mit nur einer davon gefährlich ist. Wer mit unbegrenzten Retries darauf zielt, nichts zu verlieren, gefährdet die begrenzte Verzögerung. Wer langsame Nachrichten verwirft, um die Verzögerung zu begrenzen, gefährdet das Ziel, nichts zu verlieren. Wer Retries verweigert, um Duplikate zu vermeiden, bringt den Verlust zurück. Zuverlässigkeit ist ein bewusster Ausgleich, der pro Ablauf gewählt und aufgeschrieben wird.

Warum der Happy Path in der Produktion scheitert

Gehen Sie mit der Definition in der Hand zurück zu Parcellys erstem Monat und ordnen Sie die Ausfälle nach Eigenschaft. Einige wenige Mechanismen erklären den Großteil des Schadens.

Was Parcelly sah	Gefährdete Eigenschaft	Zugrunde liegende Ursache	Kapitel
Bestellung angenommen, aber danach passiert nie etwas	Kein Verlust	Ereignis per Commit in die Datenbank geschrieben, aber nie veröffentlicht	4
Bestellung zweimal versandt	Keine schädliche Duplizierung	Erneut zugestellte Nachricht wurde nochmals verarbeitet	7
Eine fehlerhafte Nachricht blockiert eine Partition	Begrenzte Verzögerung	Eine Nachricht schlägt jedes Mal fehl und wird endlos wiederholt	8
Bestellung „bestätigt“ vor „aufgegeben“	Erklärbarkeit	Zusammengehörige Ereignisse in falscher Reihenfolge verarbeitet	9
Ledger stimmt nicht mit Payments überein	Erklärbarkeit	Keine gemeinsame Kennung, keine Möglichkeit, die Sichten zu vergleichen	13

Die vierte Zeile ist die unbequeme. Umordnung dupliziert nichts. Sie beschädigt den Zustand, etwa wenn ein veraltetes Update ein neueres überschreibt, und keine der vier Eigenschaften benennt das direkt. Ich ordne sie unter Erklärbarkeit ein, weil das Symptom eine Historie ist, die niemand erklären kann, und Kapitel 9 behandelt den Zustandsschaden selbst.

Die erste Zeile verdient einen genaueren Durchgang, denn sie ist der Ausfall, der die Flitterwochen am häufigsten beendet.

```
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))
```

Dieser Block ist nicht sicher und wird als fehlerhafte Ausgangslage gezeigt. Der Datenbank-Commit und das Publish beim Broker sind zwei Systeme ohne gemeinsame Transaktion, was das Buch einen **Dual Write** nennt. Hält der Prozess zwischen den beiden Zeilen an, existiert die Bestellung, und niemand erfährt davon. Werden die Zeilen vertauscht, kann ein Ereignis für eine Bestellung veröffentlicht werden, die nie per Commit festgeschrieben wurde. Keine der beiden Reihenfolgen ist korrekt, und beide bestehen jeden Test, der den Prozess nicht genau im falschen Moment beendet.

Das ist kein seltener Bug. Es ist das Standardverhalten eines Entwurfs, der nie ein Ausfallmodell hatte. Kapitel 3 liefert das Modell, und Kapitel 4 liefert die Lösung.

Eine Landkarte des Buches

Der Rest des Buches beantwortet die Fragen, die die Definition aufwirft, in der Reihenfolge, in der ein Team ihnen meist begegnet.

Frage, die Sie stellen werden	Kapitel
Was verspricht der Broker tatsächlich, und was nicht?	2
Was kann kaputtgehen, und was sehe ich, wenn es passiert?	3

Frage, die Sie stellen werden	Kapitel
Wie ändere ich meine Datenbank und veröffentliche ein Ereignis, ohne eines zu verlieren?	4
Wie gelangen veröffentlichte Ereignisse von der Datenbank zum Broker?	5
Wie führe ich Retries durch, ohne alles schlimmer zu machen?	6
Wie mache ich wiederholte Verarbeitung harmlos?	7
Was tue ich mit einer Nachricht, die nie gelingen kann?	8
Wann kommt es auf die Reihenfolge an, und wie halte ich sie ein?	9
Wie koordiniere ich Arbeit über Services hinweg und mache sie rückgängig?	10
Was sollte ein Ereignis enthalten, und wie sollte es heißen?	11
Wie ändere ich ein Ereignis, ohne Consumer zu brechen?	12
Wie sehe, messe und repariere ich, was gerade passiert?	13
Wie führe ich die Wiederherstellung nach verlorenen Daten, einer verlorenen Region oder einem Broker-Wechsel durch?	14
Wie teste ich die Ausfallpfade?	15
Was prüfe ich vor dem Go-live?	16

Teil I, den dieses Kapitel eröffnet, baut das Vokabular und das Ausfallmodell auf. Teil II behandelt das zuverlässige Veröffentlichen, Teil III das zuverlässige Konsumieren, Teil IV Workflows und Verträge und Teil V Betrieb und Nachweis des Systems. Lesen Sie die Teile beim ersten Durchgang der Reihe nach. Wenn Sie heute gegen ein bestimmtes Problem kämpfen, ist die Tabelle ein Index.

Abwägungen und Kosten

Der Umstieg auf Ereignisse ist nicht falsch, und dieses Kapitel ist kein Plädoyer dagegen. Es ist ein Plädoyer dafür, den Umstieg mit offenen Augen zu vollziehen, denn die Kosten sind real und fallen an bestimmten Stellen an.

Sie zahlen mit der Komplexität des Zustands. Der synchrone Ablauf hatte zwei Ausgänge: erledigt oder fehlgeschlagen. Der ereignisgesteuerte Ablauf hat viele Zwischenzustände, und jeder muss benannt, Nutzern ge-

zeigt, überwacht und irgendwann aufgelöst werden. Jemand muss für diese Zustandsmaschine die Verantwortung tragen.

Sie zahlen mit betrieblichem Können. Die Fehlersuche in einer Aufrufkette braucht einen Stack-Trace. Die Fehlersuche in einem Nachrichtenfluss braucht Correlation-IDs, Lag-Dashboards und die Fähigkeit, über Timing nachzudenken. Ein Team ohne diese stellt fest, dass die Zeit bis zum Verständnis eines Incidents wächst, selbst wenn Incidents seltener werden. Die Kosten fallen bei der Person an, die Rufbereitschaft hat.

Sie zahlen mit Testaufwand. Der Happy Path lässt sich mit wenig Aufwand testen. Ausfallpfade brauchen duplizierte Nachrichten, umgeordnete Nachrichten und Neustarts in ungünstigen Momenten. Kapitel 15 zeigt, wie das ohne eine langsame, instabile Suite geht, aber der Aufwand ist nicht null.

Sie zahlen mit Latenz bei einfachen Dingen. Eine Operation, die ein Funktionsaufruf war, wird zu mehreren Hops über einen Broker. Braucht der Aufrufer die Antwort sofort, etwa beim Validieren eines Rabattcodes vor der Anzeige eines Preises, ist ein synchroner Aufruf das bessere Mittel.

Ich würde keine Ereignisse einsetzen, wenn zwei Schritte gemeinsam gelingen oder scheitern müssen und im selben Service und in derselben Datenbank liegen. Eine lokale Transaktion ist einfacher, stärker und billiger als jedes Pattern in diesem Buch. Ich würde auch zögern, wenn das Team keine Kapazität hat, einen Broker und das Monitoring darum herum zu betreiben, oder wenn das System so klein ist, dass eine einzelne auslieferbare Einheit mit einer Datenbank die Aufgabe erfüllt. Viele gute Systeme mischen beide Stile: einen synchronen Aufruf für eine Frage, die eine Antwort braucht, ein Ereignis für eine Tatsache, die andere interessieren könnte. Ich würde pro Interaktion wählen, nicht pro System.

Die Definition mit vier Eigenschaften hat ebenfalls eine Grenze. Sie ist eine Linse, keine Spezifikation. Sie sagt Ihnen nicht, welche Verzögerungsgrenze für Ihr Geschäft richtig ist oder welche Duplikate für einen bestimmten Consumer schädlich sind. Ich kenne Ihre Zahlen nicht, und diese Antworten kommen selten allein aus der Technik. Was ich anbieten kann, ist die Gewohnheit, nach ihnen zu fragen.

Häufige Fehler

- **Nur den Happy Path testen.** Es ist verlockend, weil die Feature-Anforderung ihn beschrieb und er schnell durchläuft. Behandeln Sie jeden Pfeil im Ablaufdiagramm als Frage nach dem Ausfall, und schreiben Sie pro Antwort mindestens einen Test (Kapitel 15).
- **Den Broker als Zuverlässigkeitsschicht behandeln.** Broker bewerben Dauerhaftigkeit, was dazu verleitet. Ein Broker deckt das Segment zwischen Annahme und Zustellung ab, und die Ausfälle, die am meisten schmerzen, liegen davor und danach. Listen Sie auf, was der Broker verspricht, und dann die Lücken rund um das Versprechen (Kapitel 2 und Kapitel 3).
- **„Irgendwann“ als Entwurf verwenden.** Es klingt nach einer Entscheidung und weicht der schwierigeren Frage aus, wie lange. Hängen Sie an jeden relevanten asynchronen Hop eine Grenze und einen Alarm, und sagen Sie, was der Nutzer während des Wartens sieht.
- **Annehmen, Duplikate seien zu selten, um sie zu behandeln.** In einer ruhigen Testumgebung treten sie nie auf. Entwerfen Sie Consumer unter der Annahme, dass jede Nachricht mehr als einmal ankommen kann (Kapitel 7).
- **Den Commit in der Datenbank und das Publish beim Broker als zwei schlichte Schritte ausführen.** Es liest sich natürlich und funktioniert in jeder Demo. Die Lösung ist die Transactional Outbox aus Kapitel 4.
- **Zuverlässigkeit als Uptime definieren.** Uptime lässt sich billig messen, aber ein Service kann den ganzen Monat laufen und trotzdem Nachrichten verlieren oder verspätet verarbeiten. Messen Sie die vier Eigenschaften direkt: was verloren ging, was dupliziert wurde, wie spät die Dinge waren und ob Sie es erklären konnten.

Zentrale Erkenntnisse

- Der Happy Path ist eine Geschichte, in der jeder Schritt gelingt, und ein System, das nur gegen ihn getestet wurde, hat gezeigt, dass es in dem einen Fall funktioniert, der selten lange zutrifft.
- Der Wechsel von synchronen Aufrufen zu Ereignissen beseitigt die Kopplung nicht; er verlagert sie von Zeit und Verfügbarkeit auf Verträ-

ge, Broker-Verhalten und unausgesprochene Timing-Annahmen.

- Ausfälle in ereignisgesteuerten Systemen sind leise und über Services verteilt, daher erfährt ein Entwurf ohne Ausfallmodell tendenziell in der Produktion von seinen Nahtstellen.
- „Irgendwann“ ist ein bedingtes Versprechen ohne angehängte Dauer, daher braucht jeder relevante asynchrone Hop eine Grenze, einen Alarm und einen definierten, für Nutzer sichtbaren Zustand für die Wartezeit.
- Ein zuverlässiges System hat keinen Verlust, keine schädliche Duplizierung, begrenzte Verzögerung und Erklärbarkeit, und weil diese sich gegeneinander aufwiegen, braucht jeder Ablauf einen bewussten, aufgeschriebenen Ausgleich.
- Die Nahtstellen zwischen Komponenten, etwa ein Datenbank-Commit mit anschließendem Broker-Publish, richten mehr Schaden an als die Komponenten selbst, und der größte Teil dieses Buches handelt von diesen Nahtstellen, untersucht mit drei Fragen: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall übersteht.
- Ereignisse sind nicht immer das richtige Mittel: Wenn Schritte sich eine lokale Transaktion teilen können oder der Aufrufer sofort eine Antwort braucht, ist ein synchroner Aufruf einfacher und stärker.

Das nächste Kapitel betrachtet den Broker selbst. Bevor wir um seine Lücken herum entwerfen können, müssen wir genau wissen, was eine Warteschlange oder ein Log verspricht, wenn sie sagt, eine Nachricht sei zugestellt worden, und was sie stillschweigend Ihnen überlässt.

Kapitel 2 — Messaging-Semantik

Stellen Sie sich einen Freitagabend bei Parcelly vor, mitten in einer Rabattaktion. Der Service Notifications versendet seit Monaten E-Mails mit „Ihre Bestellung ist bestätigt“, ohne dass es je eine Beschwerde gab. Während das Backlog wächst, fügt das Autoscaling Consumer hinzu und entfernt wenige Minuten später einige wieder. Am Ende des Abends liegen beim Support zwei Tickets vor. Ein Käufer hat dieselbe Bestätigungs-E-Mail dreimal erhalten. Ein anderer hat keine erhalten, obwohl die Bestellung bestätigt ist.

Das Team öffnet das Dashboard des Brokers und stellt fest, dass nichts kaputt ist. Für jeden Offset wurde ein Commit ausgeführt. Jeder Consumer ist gesund. Die Graphen sind flach. Und doch hat dasselbe System bei einem Käufer ein Duplikat und bei einem anderen Stille erzeugt, und beide Ergebnisse stehen im Einklang mit der Dokumentation.

Jemand sagt, der Broker habe „At-least-once-Zustellung“ versprochen, und jemand anderes sagt, er sei davon ausgegangen, dass der Broker das übernehme. Beide Aussagen stimmen, und keine erklärt, was den beiden Käufern widerfahren ist. Angenommen, Notifications liest ein Topic auf Log-Basis als eine Consumer-Group, wobei sein Client die Offsets automatisch per Commit festschreibt, und bei jeder Änderung der Gruppengröße lief ein Rebalance. Diese beiden Tatsachen können beide Tickets hervorbringen, und das Kapitel zeigt, wie.

Um das zu erklären, braucht das Team ein präzises Bild davon, was ein Broker verspricht, was nicht und welche Teile des Lebens einer Nachricht stattdessen der Anwendung gehören. Was durchläuft eine Nachricht tatsächlich zwischen einem Producer und einem abgeschlossenen Seiteneffekt, und wo beginnt und endet jede Garantie?

Das Problem und warum es schwierig ist

Die meisten Entwickler begegnen einem Broker über seine einfachste Beschreibung: Man veröffentlicht eine Nachricht, und ein Consumer empfängt sie. Diese Beschreibung verbirgt drei getrennte Übergaben, und jede Zuverlässigkeitsfrage in diesem Buch liegt in einer davon.

Die erste Übergabe geht vom Producer zum Broker. Die zweite vom Broker zum Consumer. Die dritte vom Consumer zum Effekt: dem Schreibvorgang in die Datenbank, der E-Mail, dem Aufruf bei einem Paketdienst. Ein Broker hat zu den ersten beiden viel zu sagen und zur dritten fast nichts, weil die dritte in Code und in Systemen stattfindet, die der Broker nicht sehen kann.

Die Schwierigkeit liegt darin, dass jede Übergabe mit einem Ack endet, und ein Ack ist eine Aussage über die Vergangenheit, die eine Partei über ein Netzwerk hinweg gegenüber einer anderen macht. Es sagt „Ich habe das empfangen“ oder „Ich bin damit fertig“. Es sagt nicht „und es kann nichts anderes mehr schiefgehen“. Geht ein Ack verloren, kann der Sender nicht erkennen, ob die Arbeit stattgefunden hat. Diese Unsicherheit ist die Wurzel der drei Zustellungssemantiken, und wir kommen darauf zurück, sobald wir das Vokabular haben.

Eine zweite Schwierigkeit besteht darin, dass das Wort „Broker“ zwei Familien umfasst, die sich auf relevante Weise unterschiedlich verhalten. Eine **Warteschlange** ist eine Broker-Struktur, in der jede Nachricht an einen Consumer zugestellt und einzeln per Ack bestätigt wird. Ein **Ereignis-Log** ist eine dauerhafte Append-only-Folge von Ereignissen, in der Consumer in ihrem eigenen Tempo lesen und eine Position festhalten. Aussagen, die für die eine Familie gelten, sind für die andere oft falsch. Das Buch nennt die Familie, wann immer die Unterscheidung zählt, und dieses Kapitel ist der Ort, an dem die Unterscheidung aufgebaut wird.

Eine dritte Schwierigkeit besteht darin, dass die Garantien in der Dokumentation eines Produkts einen begrenzten Geltungsbereich haben. „At-least-once zugestellt“ ist eine Aussage über die eigene Grenze des Brokers. Sie deckt nicht ab, was der Consumer nach dem Empfang der Nachricht tut, und sie hängt von Einstellungen ab, die der Producer beim Veröffentlichen gewählt hat. Wer die Garantie ohne ihren Geltungsbereich liest, landet bei dem Freitagabend von oben.

Warteschlangen und Logs

Das Warteschlangenmodell

In einer Warteschlange verfolgt der Broker den Zustand jeder einzelnen Nachricht. Eine Nachricht wartet, ist an einen Consumer zugestellt oder per Ack bestätigt und verschwunden. Mehrere Consumer können aus derselben Warteschlange lesen; diese Anordnung heißt **konkurrierende Consumer**, weil jede Nachricht jeweils an einen von ihnen zugestellt wird und sie um die nächste konkurrieren.

Bestätigt ein Consumer eine Nachricht per Ack, löscht der Broker sie. Die Aufbewahrung ist daher bewusst kurz: Eine Nachricht lebt, bis sie verarbeitet ist oder bis sie abläuft. Es gibt keine eingebaute Möglichkeit, bei der Warteschlange eine Nachricht anzufordern, die bereits bestätigt wurde, weil die Warteschlange sie nicht aufbewahrt.

Die Folge ist, dass eine Warteschlange gut zu Arbeit passt, die einmal erledigt und dann vergessen werden soll. Ein Parcelly-Job wie „dieses Rechnungs-PDF erzeugen“ ist ein naheliegender Kandidat: Man erledigt die Arbeit, danach ist die Nachricht nutzlos. Der Preis ist, dass es keine Historie gibt. Sendet ein Bug eine Stunde lang den falschen Text, sind die Nachrichten, die die Fakten trugen, verschwunden, und die Reparatur muss an anderer Stelle beginnen.

Das Log-Modell

In einem Ereignis-Log verfolgt der Broker keinen Zustand pro Nachricht. Er speichert Nachrichten der Reihe nach in Partitionen und bewahrt sie nach einer Aufbewahrungsrichtlinie auf, die unabhängig davon ist, wer sie gelesen hat. Eine **Partition** ist eine geordnete Teilfolge eines Topics, und ein **Topic** ist ein benannter Stream von Nachrichten. Jede Nachricht hat einen **Offset**, ihre Position in der Partition.

Consumer halten fest, wie weit sie gelesen haben. Auch diese Position heißt Offset, und der Consumer oder der Broker in seinem Auftrag zeichnet sie auf. Eine **Consumer-Group** ist eine Menge von Consumern, die sich die Arbeit an einem Topic teilen: Jede Partition ist jeweils einem Consumer der Gruppe zugewiesen. Verschiedene Gruppen lesen denselben Topic unabhängig voneinander, jede mit eigenen Offsets.

Die bestimmende Eigenschaft des Logs ist, dass Lesen nicht verbraucht. Zehn Gruppen können denselben Topic lesen, und eine neue Gruppe kann von vorn beginnen. **Replay**, das bewusste erneute Lesen der Historie, ist der Begriff, den dieses Buch für diesen Vorgang reserviert, und er ist nur möglich, weil das Log die Daten behält. Der Ledger von Parcelly ist ein gutes Beispiel für einen Consumer, der davon profitiert: Wird ein Bug in der Art gefunden, wie Einträge aufgebaut werden, kann der Service Ledger korrigiert und auf alte Ereignisse zurückgesetzt werden.

Der Preis ist, dass das Log die Buchführung auf die Seite des Consumers verlagert. Im klassischen Log-Modell kann der Broker eine fehlgeschlagene Nachricht nicht erneut zustellen und gleichzeitig zur nächsten übergehen, weil er Nachrichten nicht einzeln verfolgt. Ein Consumer, der die Nachricht bei Offset 500 nicht verarbeiten kann, muss entscheiden, was er tut, bevor er sicher zu 501 weitergehen kann.

Die Wahl zwischen beiden

Aspekt	Familie der Warteschlangen	Familie der Logs
Verfolgte Einheit	Jede Nachricht	Eine Position pro Partition
Aufbewahrung	Bis zum Ack oder zum Ablauf	Nach Zeit oder Größe, unabhängig von den Lesern
Mehrere unabhängige Leser	Braucht pro Leser eine Kopie der Nachricht	Natürlich, eine Position pro Gruppe
Replay der Historie	Nicht verfügbar	Innerhalb der Aufbewahrung verfügbar
Eine fehlschlagende Nachricht	Wird einzeln erneut zugestellt, die anderen laufen weiter	Blockiert die Partition, solange sie nicht behandelt wird
Reihenfolge	Bestenfalls FIFO im Speicher; geht über konkurrierende Consumer und erneute Zustellung hinweg verloren	Stark innerhalb einer Partition

Ich würde eine Familie nicht nach der Beliebtheit eines Tools auswählen. Ich würde mit drei Fragen beginnen: Muss jemand diese Historie noch einmal lesen, brauchen mehrere Consumer dieselben Fakten, und spielt die Reihenfolge für dasselbe Aggregat eine Rolle? Ein Ja auf eine davon

spricht für ein Log. Eine reine Arbeitswarteschlange, ohne Bedarf an Historie und mit Bedarf an unabhängigen Retries einzelner Nachrichten, spricht für eine Warteschlange.

Viele Teams betreiben am Ende beides. Ereignisse, aus denen andere Services Zustand aufbauen, kommen auf ein Log, und Aufgaben, die einmal geschehen sollen, kommen auf eine Warteschlange. Das ist ein legitimes Design, und es kostet den Aufwand, zwei Semantiken zu betreiben und zu verstehen. Kapitel 9 kommt darauf zurück, was Reihenfolge in jeder Familie bedeutet.

Push, Pull und der Zustellungsfluss

Broker unterscheiden sich auch darin, wer die Zustellung auslöst. In einem **Push**-Modell sendet der Broker Nachrichten an Consumer, sobald sie verfügbar werden. In einem **Pull**-Modell fragt der Consumer nach Nachrichten, wenn er bereit ist.

Broker auf Warteschlangenbasis arbeiten üblicherweise mit Push (RabbitMQ stellt an Consumer zu, während Consumer von Amazon SQS per Polling abfragen, sodass selbst das innerhalb der Familie variiert), mit einer Obergrenze dafür, wie viele unbestätigte Nachrichten ein Consumer gleichzeitig halten darf. Diese Obergrenze heißt oft Prefetch-Anzahl oder Fenster. Broker auf Log-Basis arbeiten üblicherweise mit Pull: Der Consumer fragt ab seinem aktuellen Offset den nächsten Batch an. Beide Modelle gibt es in beiden Familien, daher würde ich das als Tendenz behandeln und nicht als Gesetz.

Der Unterschied zählt vor allem dafür, was geschieht, wenn ein Consumer langsam ist. In einem Push-Modell ohne Obergrenze sendet der Broker weiter, der Speicher des Consumers füllt sich mit Nachrichten, die er noch nicht begonnen hat, und diese Nachrichten befinden sich in einem Zustand, in dem sie weder für einen anderen Consumer verfügbar noch erledigt sind. In einem Pull-Modell steuert der Consumer sein eigenes Tempo, und ein langsamer Consumer zeigt sich als wachsender Consumer-Lag und nicht als überlasteter Prozess. Der **Consumer-Lag** gibt an, wie weit ein Consumer hinter dem Head einer Partition oder Warteschlange zurückliegt, in Nachrichten oder in Zeit.

Die praktische Regel lautet: Ein Consumer sollte nie mehr Arbeit halten, als er beenden kann, bevor sein Anspruch auf die Arbeit abläuft. Kapitel 3 behandelt langsame Consumer und Backpressure als eigene Ausfälle, und Kapitel 6 behandelt, wie man die Last begrenzt. Merken Sie sich vorerst, dass „zugestellt“ und „in Bearbeitung“ verschiedene Zustände sind und dass in der Lücke dazwischen Zeit vergeht.

Ack, erneute Zustellung und Offsets

Ack in einer Warteschlange

Ein **Acknowledgment**, meist zu **Ack** verkürzt, ist die Aussage des Consumers, dass er mit einer Nachricht fertig ist und der Broker sie vergessen darf. Ein **Nack**, ein negatives Acknowledgment, besagt, dass der Consumer die Nachricht nicht verarbeiten konnte. Je nach Broker und Einstellungen schickt ein Nack die Nachricht in die Warteschlange zurück, verwirft sie oder leitet sie an eine Dead-Letter-Queue (DLQ) weiter.

Die zentrale Entwurfsfrage ist, wann der Ack erfolgt. Es gibt nur zwei vernünftige Positionen.

- **Ack vor der Verarbeitung.** Der Broker vergisst die Nachricht sofort. Stürzt der Consumer mitten in der Arbeit ab, geht die Nachricht verloren. Das ergibt At-most-once-Verhalten.
- **Ack nach der Verarbeitung.** Der Broker hält die Nachricht, bis die Arbeit erledigt ist. Stürzt der Consumer nach dem Abschluss, aber vor dem Ack ab, kommt die Nachricht zurück, und die Arbeit wird zweimal ausgeführt. Das ergibt At-least-once-Verhalten.

Es gibt keine Position, die beide Eigenschaften liefert, weil der Ack und die Arbeit zwei getrennte Aktionen auf zwei getrennten Systemen sind und ein Absturz zwischen sie fallen kann.

Visibility-Timeout und erneute Zustellung

Woran erkennt eine Warteschlange, dass ein Consumer abgestürzt ist? Das kann sie nicht direkt beobachten. Manche Broker überwachen die Verbindung und werten eine geschlossene Verbindung als Ausfall. Andere verwenden ein **Visibility-Timeout**: Wird eine Nachricht zugestellt, ist sie für andere Consumer eine festgelegte Zeit lang unsichtbar. Bestätigt der Con-

sumer per Ack, bevor die Zeit abläuft, wird die Nachricht gelöscht. Andernfalls wird die Nachricht wieder sichtbar und an jemand anderen zugestellt. Diese zweite Zustellung ist eine **erneute Zustellung**.

Der Timeout ist eine Schätzung, wie lange die Verarbeitung dauert, und beide Arten, falsch zu schätzen, kosten etwas. Ist er zu kurz, wird die Nachricht eines Consumers, der langsam, aber lebendig ist, an einen zweiten Consumer übergeben, während er noch arbeitet. Nun verarbeiten zwei Consumer dieselbe Nachricht, und der schließlich eintreffende Ack des ersten Consumers kann fehlschlagen, ignoriert werden oder je nach Broker erfolgreich sein und die Nachricht löschen, während der zweite Consumer noch daran arbeitet. In diesem letzten Fall verliert ein Fehlschlag des zweiten Versuchs die Nachricht. Ist er zu lang, lässt ein echter Absturz die Nachricht für diesen ganzen Zeitraum verborgen, und die Verzögerung zeigt sich als Latenz beim Käufer.

Angenommen, der Consumer von Inventory schließt eine Reservierung normalerweise in weit unter einer Sekunde ab, und jemand setzt den Timeout auf 30 Sekunden. Diese Zahl ist beispielhaft. Ein Datenbank-Schluckauf, der eine Reservierung 40 Sekunden dauern lässt, führt nun zu einer zweiten Zustellung desselben `OrderPlaced`, und Inventory versucht, den Bestand zweimal zu reservieren. Im Broker ist nichts kaputt. Der Timeout hat eine Annahme kodiert, und die Annahme galt nicht mehr.

Manche Systeme lassen einen Consumer den Timeout verlängern, während er arbeitet. Das hilft bei berechtigterweise langen Jobs und fügt eine neue Fehlerquelle hinzu: einen Consumer, der den Timeout für eine Nachricht endlos verlängert, die er nie abschließen wird.

Offsets in einem Log

In einem Log gibt es kein Ack pro Nachricht. Stattdessen führt der Consumer einen **Offset-Commit** aus: Er hält fest „Ich habe alles vor dieser Position verarbeitet“. Nach einem Neustart oder einer Neuzuweisung liest der nächste Consumer ab dem per Commit festgeschriebenen Offset.

Dieselben zwei Positionen gibt es auch hier. Schreibt man den Offset vor der Verarbeitung per Commit fest, verliert ein Absturz die Nachrichten nach der zuletzt abgeschlossenen bis zur festgeschriebenen Position. Schreibt man ihn nach der Verarbeitung fest, wiederholt ein Absturz die

Nachrichten zwischen dem letzten Commit und dem Absturz. Echte Consumer führen Commits batchweise aus, sodass die Zahl der wiederholten Nachrichten nach einem Absturz ein Batch ist und nicht eine.

Viele Client-Bibliotheken können automatisch einen Commit ausführen. Das ist bequem, und es ist zugleich die am wenigsten explizite Art, die eigene Zustellungssemantik festzulegen, denn je nach Client kann der Commit zu einem Zeitpunkt erfolgen, der nichts damit zu tun hat, wann die eigene Verarbeitung fertig war. Wird die Verarbeitung an einen anderen Thread übergeben, kann der Client Arbeit festschreiben, die nie abgeschlossen wurde. Meiner Einschätzung nach sollte ein Consumer, auf den es ankommt, explizit einen Commit ausführen, an einer Stelle, auf die das Team im Code zeigen und über die es nachdenken kann.

Consumer-Groups und Rebalance

Treten Consumer einer Consumer-Group bei oder verlassen sie sie, verteilt der Broker die Partitionen unter ihnen neu. Das nennt man **Rebalance**. Während eines Rebalance wird eine Partition, die ein Consumer las, einem anderen zugewiesen, und der neue Besitzer beginnt beim zuletzt per Commit festgeschriebenen Offset.

Gruppen auf Log-Basis stellen in anderer Form dieselbe Schätzung an wie der Visibility-Timeout in einer Warteschlange. Ein Consumer, der sich nicht innerhalb eines konfigurierten Intervalls beim Broker meldet, gilt als tot, seine Partitionen werden neu zugewiesen, und seine nicht festgeschriebene Arbeit wird vom neuen Besitzer erneut gelesen. Ein langsamer Consumer, der noch lebt, ist die alltägliche Ursache von Rebalance-Duplikaten.

Hier kommen die Duplikate aus dem Eingangsszenario her. Betrachten Sie einen Consumer, der die Nachrichten 100 bis 150 verarbeitet, aber nur bis 120 einen Commit ausgeführt hat. Ein Deployment startet ihn neu, und die Partition wechselt zu einem zweiten Consumer, der ab 120 liest. Die Nachrichten 120 bis 150 werden zweimal verarbeitet, und kein Teil des Systems hat versagt. Ein Rebalance ist ein normales Ereignis, das bei jedem Deployment, jeder Hochskalierung und jedem Absturz auftreten kann, daher ist erneute Zustellung ein alltägliches Vorkommnis und kein seltener Unfall.

Die Stille hat die spiegelbildliche Ursache. Hat der Client Offsets für Nachrichten per Commit festgeschrieben, die der Consumer nur an einen Hintergrund-Sender übergeben hatte, und wurde der Prozess gestoppt, bevor der Sender fertig war, beginnt der nächste Besitzer nach diesen Nachrichten. Sie werden nie wieder gelesen, und keine Log-Zeile sagt es. Dasselbe Rebalance kann also die Nachricht eines Käufers wiederholen und die eines anderen verlieren.

Es gibt eine verwandte Gefahr: Für kurze Zeit weiß der alte Besitzer womöglich noch nicht, dass er die Partition verloren hat, und arbeitet unter Umständen weiter daran. Zwei Consumer können dann dieselbe Partition gleichzeitig verarbeiten. Gut entworfene Clients versuchen, das zu verhindern, aber ich würde keinen Consumer entwerfen, der davon abhängt, dass die Verhinderung perfekt ist.

Zustellungssemantik

Mit der Mechanik im Rücken sind die drei Zustellungssemantiken Folgen dessen, wo man den Ack platziert.

At-most-once

At-most-once bedeutet, dass jede Nachricht null- oder einmal zugestellt wird. Man erreicht es, indem man vor der Verarbeitung per Ack bestätigt oder einen fehlgeschlagenen Publish nicht wiederholt. Eine Nachricht kann verloren gehen, und sie kann nie dupliziert werden.

Es ist eine vernünftige Wahl, wenn der Verlust einer Nachricht weniger kostet als der Umgang mit Duplikaten. Ein periodischer Ping „aktuelle Position“ eines Parcelly-Paketdienstes ist ein Fall dafür: Der nächste Ping macht den verlorenen unerheblich. Für `PaymentCaptured` ist es die falsche Wahl, denn dort bedeutet eine verlorene Nachricht, dass Geld bewegt wurde und niemand es aufgezeichnet hat.

At-least-once

At-least-once bedeutet, dass jede Nachricht ein- oder mehrmals zugestellt wird. Man erreicht es, indem man nach der Verarbeitung per Ack bestätigt und so lange wiederholt, bis ein Ack eintrifft. Eine Nachricht geht nie verloren, solange der Broker sie hält, und sie kann erneut zugestellt werden.

Das ist die Semantik, auf der das Buch aufbaut. Sie ist die einzige der drei, bei der der Ausfall sichtbar und behebbar ist: Ein Duplikat lässt sich erkennen und ignorieren, während eine verlorene Nachricht unter Umständen tagelang unentdeckt bleibt. Der Preis ist, dass jeder Consumer ein **Duplikat** tolerieren muss, also eine Nachricht, die mehr als einmal verarbeitet oder zugestellt wurde. Wie das geht, behandelt Kapitel 7.

„Exactly-once“

Zustellung ist ein physischer Vorgang über ein unzuverlässiges Netzwerk, und der Ack des Empfängers kann wie jede andere Nachricht verloren gehen. Ein Sender, der kein Ack erhalten hat, hat zwei Optionen: erneut senden, was ein Duplikat riskiert, oder nicht, was einen Verlust riskiert. Kein Protokoll nimmt diese Wahl den beiden Parteien allein ab.

Exactly-once ist dennoch ein reales und nützliches Ergebnis, und es ist ein Effekt, keine Zustellungsgarantie. Trifft At-least-once-Zustellung auf einen Consumer, dessen wiederholte Verarbeitung dieselbe Wirkung hat wie einmalige Verarbeitung, verhält sich das Gesamtsystem, als sei jede Nachricht genau einmal behandelt worden. Das Rezept lautet At-least-once-Zustellung plus ein **idempotenter Consumer**.

Manche Broker bewerben Exactly-once-Verarbeitung, und die Behauptung ist innerhalb einer Grenze berechtigt. Ein Broker auf Log-Basis kann einen Zyklus aus Lesen, Verarbeiten und Schreiben atomar machen, wenn Eingabe, Ausgabe und Offset-Commit alle im selben Broker liegen. Innerhalb dieser Grenze wird bei einem Absturz alles gemeinsam zurückgerollt. Die Garantie setzt außerdem voraus, dass jeder nachgelagerte Leser nur festgeschriebene Daten liest, und sie stützt sich auf eine stabile Producer-Identität, damit eine alte Instanz nach einem Neustart ausgesperrt wird. Die Grenze endet dort, wo Ihre Seiteneffekte beginnen. Sobald der Consumer in eine relationale Datenbank schreibt, einen Zahlungsanbieter aufruft oder eine E-Mail sendet, deckt die Transaktion des Brokers den Effekt nicht mehr ab, und das Duplikatproblem kehrt zurück.

Aus diesem Grund bleibt die Voreinstellung des Buches At-least-once plus idempotente Consumer. Sie funktioniert mit jedem Broker und jedem Seiteneffekt, sie hängt nicht davon ab, dass ein Feature aktiviert ist, und sie übersteht die Migration auf ein anderes Produkt, ein Schritt, den Kapitel 14 behandelt. Wenn Sie eine ganze Pipeline innerhalb der Transaktions-

grenze eines Brokers halten können, kann das Feature die Arbeit verringern, und ich würde die Consumer trotzdem so entwerfen, dass sie nicht darauf angewiesen sind.

> **Hinweis:** Auch Producer können Duplikate erzeugen, noch bevor ein Consumer beteiligt ist. Ein Producer, bei dem das Warten auf den Publish-Ack in ein Timeout läuft, veröffentlicht in der Regel erneut, und der Broker hält nun möglicherweise zwei Kopien eines Ereignisses. Beide Kopien tragen dieselbe `event_id`, sofern der Producer sie einmal beim Erzeugen des Ereignisses vergeben und nicht für den Retry eine neue geprägt hat. Manche Broker entdoppeln die eigenen Retries eines Producers, per Sequenznummer innerhalb einer Producer-Session oder per Deduplizierungs-ID innerhalb eines Zeitfensters. Das deckt einen Retry innerhalb einer Verbindung ab. Es deckt keinen Prozess ab, der neu startet und erneut veröffentlicht, etwa ein Outbox-Relay, weshalb der Consumer weiterhin die `event_id` prüft. Kapitel 4 behandelt, wie man veröffentlicht, ohne Ereignisse zu verlieren, und Kapitel 7, wie man Duplikate erkennt.

Was der Broker tatsächlich garantiert

Alles bisher Beschriebene betrifft die Seite vom Broker zum Consumer. Die Seite vom Producer zum Broker hat eigene Grenzen, und sie hängen stärker von der Konfiguration ab als vom Produktnamen.

Producer-Acks

Veröffentlicht ein Producer, kann der Broker zu unterschiedlichen Zeitpunkten bestätigen, und die brauchbare Art, die Einstellungen zu unterscheiden, ist die Zahl der Knoten, die die Nachricht halten, wenn der Producer „fertig“ hört. Ohne Ack erfährt der Producer nichts über die Speicherung. Beim Ack durch den empfangenden Knoten hat ein Knoten die Nachricht angenommen, gleichgültig, ob er sie auf die Platte geschrieben hat. Beim Ack nach der Replikation existiert die Nachricht auf einer geforderten Zahl von Knoten. Das Schreiben auf die Platte ist eine eigene Frage, die weiter unten folgt.

Die Wahl des Producers bestimmt, was ein Absturz anrichten kann. Bei den ersten beiden Einstellungen kann ein Broker-Knoten, der direkt nach dem Ack stirbt, die Nachricht mitnehmen. Beim replizierten Ack kommt die Antwort später, und das Verlustfenster ist kleiner. Der Preis ist höhere Latenz bei jedem Publish und die Abhängigkeit davon, dass genug Knoten verfügbar sind, um den Schreibvorgang abzuschließen.

Broker, die Publisher Confirms anbieten, in der Familie der Warteschlangen, und Ack-Stufen, in der Familie der Logs, beschreiben dieselbe Idee. Prüfen Sie, was der Ack, auf den Sie warten, in Ihrer Konfiguration tatsächlich bedeutet. Meiner Einschätzung nach ist das eine der wertvollsten Dokumentationsseiten, die ein Team einmal lesen und aufschreiben kann.

Replikation und fsync

Replikation schützt vor dem Verlust eines Knotens: Die Nachricht existiert auf mehreren. Sie schützt nicht davor, dass alles gemeinsam ausfällt, und ihre Garantie hängt davon ab, wie viele Kopien existieren mussten, bevor dem Producer „fertig“ gesagt wurde. Prüfen Sie sowohl die Ack-Stufe als auch die Mindestzahl an Replikaten, die sie verlangt, denn „alle Replikate“ kann einen bedeuten, wenn die anderen zurückgefallen sind. Prüfen Sie auch, ob ein Failover ein Replikat befördern darf, das nicht auf dem neuesten Stand war, denn das kann Daten verlieren, von denen dem Producer gesagt wurde, sie seien sicher.

Es gibt einen verwandten und oft missverstandenen Punkt zur Platte. Das Schreiben in eine Datei legt die Daten normalerweise im Cache des Betriebssystems ab, und die Daten erreichen die physische Platte später, es sei denn, das Programm erzwingt es mit einem synchronen Flush, meist fsync genannt. Den Flush bei jeder Nachricht zu erzwingen ist langsam, daher tun das viele Broker nicht für jeden Schreibvorgang. Sie verlassen sich darauf, dass die Replikation die Lücke abdeckt, und führen den Flush nach einem Zeitplan aus. Ob der Ack eines bestimmten Brokers bedeutet, dass die Daten auf der Platte liegen, hängt von seiner Konfiguration ab, und ich würde es nicht annehmen.

Ich weiß nicht, welche dieser Einstellungen zu Ihrem System passt, und es hängt davon ab, was ein verlorenes Ereignis kostet. Ein verlorenes `PaymentCaptured` und ein verlorenes „Seite aufgerufen“ sind verschiedene Probleme, und es wäre vernünftig, sie unterschiedlich zu konfigurieren.

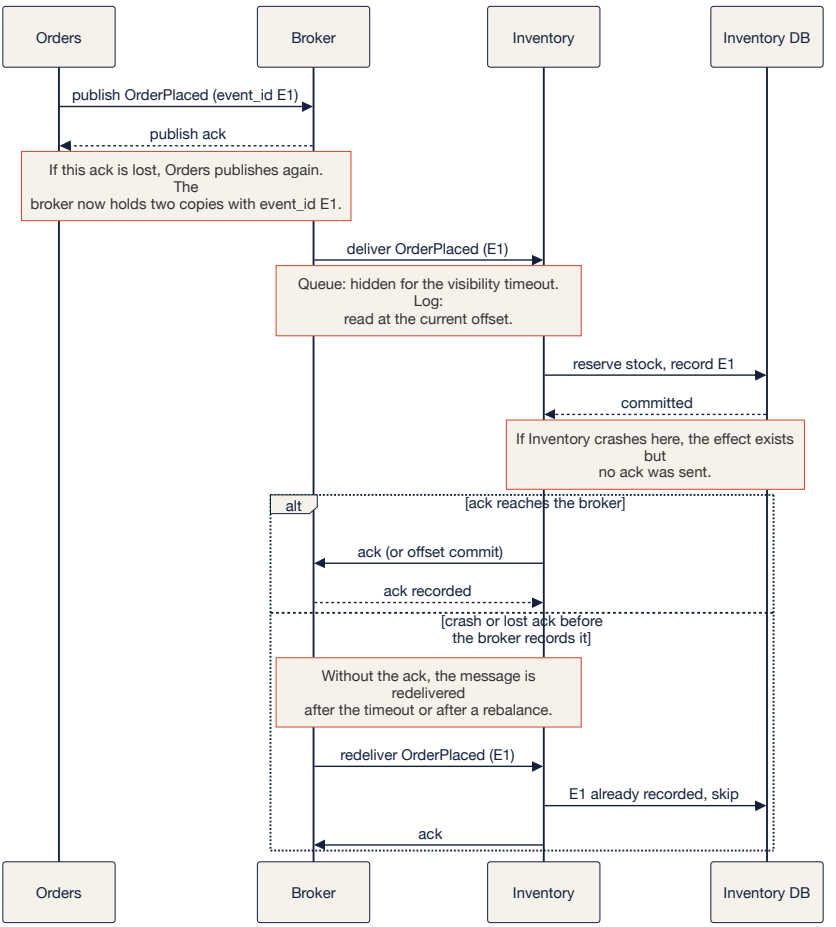
Die Grenze der Garantie

Fasst man dies zusammen, wird die Grenze klar. Ein Broker kann versprechen, dass eine Nachricht, die er angenommen hat, gehalten und Consumern angeboten wird, innerhalb seiner Grenzen für Aufbewahrung oder Ablauf und nur so dauerhaft, wie es die oben genannten Einstellungen für Ack und Replikation erlauben. Er kann nicht versprechen, dass eine Nachricht überhaupt beim Broker ankommt, denn das hängt davon ab, was der Producer getan hat, als sein Publish in ein Timeout lief. Er kann auch nicht versprechen, dass der Effekt eines Consumers genau einmal eingetreten ist.

Zwei Lücken bleiben in der Hand der Anwendung. Die erste liegt auf der Producer-Seite: Eine in der Datenbank von Orders gespeicherte Bestellung und ein an den Broker veröffentlichtes `OrderPlaced` sind zwei Schreibvorgänge, und ein Absturz kann zwischen sie fallen. Kapitel 1 hat das den Dual Write genannt, und Kapitel 4 beseitigt ihn. Die zweite liegt auf der Consumer-Seite, zwischen dem Effekt und dem Ack. Kapitel 7 schließt diese Lücke. Betrachtet als die drei Übergaben, mit denen dieses Kapitel begann, nennt der Broker Ausfallsemantik für die ersten beiden, und die beiden Lücken sind das, was in Ihrer Hand bleibt: der Dual Write vor der ersten Übergabe und die Lücke zwischen Effekt und Ack in der dritten.

Das Leben einer Nachricht

Das folgende Diagramm verfolgt eine Nachricht `OrderPlaced` vom Service Orders bis zu einem abgeschlossenen Effekt in Inventory. Achten Sie auf die beiden Stellen, an denen ein Ausfall den Sender ohne Antwort lässt, in den Notizen markiert: der verlorene Publish-Ack und der verlorene Consumer-Ack. Alles, was das Kapitel beschrieben hat, geschieht innerhalb dieser einen Sequenz.



Der Zweig `else` des Diagramms ist der Kern des Kapitels. Die erneute Zustellung ist kein Fehler, und keine Komponente hat sich falsch verhalten. Das Einzige, was verhindert, dass der Bestand zweimal reserviert wird, ist, dass Inventory `E1` in derselben Transaktion wie die Reservierung festgehalten und bei der zweiten Zustellung erneut geprüft hat.

So sieht ein Consumer mit dieser Eigenschaft als Pseudocode aus. Er verwendet eine **Inbox**, eine Tabelle verarbeiteter Werte von `event_id` in der eigenen Datenbank des Consumers, und setzt den Ack hinter den 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)
```

Dies ist das Design, das At-least-once sicher macht, und es wird hier nur eingeführt, um zu zeigen, wohin der Ack gehört. Kapitel 7 baut es aus, einschließlich der Fälle, in denen die Prüfung der Inbox allein nicht genügt.

Abwägungen und Kosten

Die Semantik zu wählen heißt wählen, welchen Ausfall man lieber handhaben möchte. Keine der Optionen beseitigt Ausfälle, und jede verlagert sie an eine andere Stelle.

At-least-once kostet Sie Idempotenzarbeit in jedem Consumer. Jeder Consumer braucht eine Möglichkeit, wiederholte Arbeit zu erkennen, meist eine Inbox-Tabelle in der eigenen Datenbank. Das bedeutet Speicher, einen Unique Constraint auf dem Hot Path und eine Bereinigungsrichtlinie. Für ein Team mit einem Dutzend Consumern ist es ein Pattern, das ein Dutzend Mal anzuwenden ist. Ist der Effekt eines Consumers von Natur aus idempotent, etwa wenn ein Status auf einen festen Wert gesetzt wird, kann die Inbox überflüssig sein, und ich würde trotzdem aufschreiben, warum.

At-most-once kostet Sie Vollständigkeit. Es ist einfach und schnell, und es ist nur akzeptabel, wenn jemand vorab entschieden hat, dass eine verlorene Nachricht billiger ist als ein Duplikat. Diese Entscheidung sollte explizit sein und festgehalten werden. Die übliche Form des Fehlers ist ein Team, das es standardmäßig gewählt hat, durch Einstellungen für automatischen Ack, ohne es zu merken.

Transaktionale Verarbeitung innerhalb eines Brokers kostet Sie Portabilität. Sie kann die Behandlung von Duplikaten innerhalb ihrer Grenze verringern, und sie bindet das Design an die Features eines Produkts, daran, dass jeder Beteiligte innerhalb dieser Grenze bleibt, und an den betrieblichen Aufwand, sie zu betreiben. Sobald Sie zu einer relationalen Datenbank oder einer externen API wechseln, brauchen Sie ohnehin Idempotenz. Ich würde das Feature nicht ablehnen, und ich würde Korrektheit nicht allein darauf aufbauen.

Starke Producer-Acks kosten Latenz und Verfügbarkeit. Das Warten auf mehrere Replikate macht jeden Publish langsamer und lässt den Schreibvorgang fehlschlagen, wenn zu wenige Knoten erreichbar sind. Für Ereignisse, die nicht verloren gehen dürfen, ist dieser Tausch meist richtig. Für Telemetrie mit hohem Volumen und geringem Wert muss er es nicht sein.

Logs kosten Buchführung auf Consumer-Seite, und Warteschlangen kosten Historie. Wählen Sie ein Log, gehören Ihnen die Richtlinie für den Offset-Commit und der Umgang mit einer Nachricht, die eine Partition blockiert. Wählen Sie eine Warteschlange, verzichten Sie auf Replay. Brauchen Sie unabhängige Retries einzelner Nachrichten und keine Historie, ist ein Log womöglich mehr Maschinerie, als die Arbeit verdient.

Wann man den Rat dieses Kapitels nicht anwendet: Ein System mit einem einzigen Producer, einem einzigen Consumer und der Toleranz für gelegentlichen Verlust, etwa ein interner Metrik-Feed, braucht weder eine Inbox noch starke Replikation. Die Kosten der Garantien sollten den Kosten des Ausfalls entsprechen. Der Fehler ist nicht, eine schwächere Garantie zu wählen; der Fehler ist, nicht zu wissen, dass man es getan hat.

Häufige Fehler

- **„Zugestellt“ als „verarbeitet“ behandeln.** Es ist verlockend, weil das Dashboard Zustellungen und Acks zeigt und das die Zahlen sind, die

gesund aussehen. Zugestellt heißt, der Broker hat die Nachricht übergeben. Verarbeitet heißt, der Effekt wurde festgeschrieben. Verfolgen Sie beides, und lassen Sie Ihren Ack dem Effekt folgen.

- **Per Ack bestätigen, bevor die Arbeit erledigt ist.** Es ist verlockend, weil es erneute Zustellung vermeidet und die Warteschlange aufgeräumt aussehen lässt. Es verwandelt jeden Absturz eines Consumers in eine verlorene Nachricht. Bestätigen Sie per Ack, nachdem der Effekt festgeschrieben ist, und behandeln Sie die Duplikate, die daraus entstehen.
- **Den automatischen Offset-Commit standardmäßig eingeschaltet lassen.** Es ist verlockend, weil es in einer Demo funktioniert und keinen Code braucht. Je nach Client kann er zu einem Zeitpunkt erfolgen, der nichts mit Ihrer Verarbeitung zu tun hat, sodass ein Absturz einen beliebigen Ausschnitt verlieren oder wiederholen kann. Führen Sie den Commit explizit aus, nach dem Effekt.
- **Den Visibility-Timeout nach Gefühl festlegen.** Es ist verlockend, weil der Standardwert meist in Ordnung scheint. Ein Timeout, der kürzer ist als ein langsamer, aber gesunder Lauf, verursacht gleichzeitige doppelte Verarbeitung. Orientieren Sie ihn an der längsten berechtigten Verarbeitungszeit, die Sie beobachten können, plus Reserve, und verlängern Sie ihn bei langen Jobs gezielt.
- **Glauben, das Exactly-once-Feature eines Brokers decke Ihre Datenbank ab.** Es ist verlockend, weil die Dokumentation die Formulierung verwendet. Die Garantie ist auf die eigene Grenze des Brokers beschränkt. Alles außerhalb, etwa eine Zeile in Ihrer Datenbank oder eine E-Mail, braucht Idempotenz.
- **Annehmen, eine veröffentlichte Nachricht sei eine gespeicherte Nachricht.** Es ist verlockend, weil der Publish-Aufruf zurückgekehrt ist. Je nach Ack-Stufe existiert die Nachricht womöglich nur im Speicher eines Knotens. Wissen Sie, worauf Ihr Producer wartet, und schreiben Sie es auf.
- **Annehmen, ein Rebalance sei selten.** Es ist verlockend, weil es leise geschieht. Deployments, Skalierungsereignisse und Abstürze können je nach Gruppenprotokoll und Einstellungen zur Mitgliedschaft alle einen auslösen, und jeder Rebalance kann einen Batch von Nachrichten wiederholen. Entwerfen Sie, als geschehe erneute Zustellung täglich, denn der Preis eines Irrtums ist ein doppelter Effekt.

Zentrale Erkenntnisse

- Eine Nachricht durchläuft drei Übergaben, Producer zu Broker, Broker zu Consumer und Consumer zu Effekt, und die Garantien eines Brokers decken nur die ersten beiden ab.
- Warteschlangen verfolgen jede Nachricht und löschen sie beim Ack, während Logs Nachrichten unabhängig von den Lesern aufbewahren und jede Consumer-Group ihren eigenen Offset führen lassen, was Replay erst möglich macht.
- Wo man den Ack oder den Offset-Commit platziert, bestimmt die Semantik: vor der Verarbeitung ergibt At-most-once, nach der Verarbeitung ergibt At-least-once, und keine Platzierung vermeidet sowohl Verlust als auch Duplikate.
- Erneute Zustellung ist Routine und keine Ausnahme, weil Visibility-Timeouts ablaufen, Consumer abstürzen und Rebalances (die Deployments häufig verursachen) Batches wiederholen.
- Exactly-once ist ein Effekt, der aus At-least-once-Zustellung und einem idempotenten Consumer entsteht, und die eigene Exactly-once-Verarbeitung eines Brokers endet an der Grenze dieses Brokers.
- Ein Producer-Ack ist nur so stark wie die Einstellungen für Replikation und Flush dahinter, daher sollte das Team wissen und festhalten, was „veröffentlicht“ für jedes wichtige Ereignis bedeutet.
- Jede semantische Option verlagert Ausfälle, statt sie zu beseitigen, und die richtige Wahl ist die, deren verbleibenden Ausfall Sie sich zu handhaben leisten können.

Die Messaging-Semantik sagt Ihnen, was der Broker an einem normalen Tag tut und nicht tut. Kapitel 3 wendet sich den unnormalen Tagen zu und baut ein praktisches Modell dessen, was tatsächlich kaputtgeht: Netzwerkpartitionen, Abstürze, langsame Consumer, Uhren und Menschen.

Kapitel 3 — Ein pragmatisches Ausfallmodell

Stellen Sie sich Payments bei Parcelly mitten in einem Einzug vor. Orders hat ein Command `CapturePayment` an Payments gesendet, und Payments hat den externen Zahlungsanbieter aufgerufen, um das Geld einzuziehen. Die Verbindung zum Anbieter verstummt. Dreißig Sekunden später gibt der HTTP-Client von Payments auf und löst ein Timeout aus.

Payments muss nun eine Frage beantworten, die es nicht beantworten kann. Hat der Anbieter die Zahlung eingezogen, bevor die Verbindung starb, oder hat er die Anfrage nie erhalten? Beide Verläufe erzeugen exakt dieselbe Beobachtung: Stille, dann ein Timeout. Wiederholt Payments den Aufruf und der erste Einzug war erfolgreich, wird der Käufer doppelt belastet. Meldet Payments einen Fehlschlag und der erste Einzug war erfolgreich, storniert Orders eine Bestellung, die bereits bezahlt ist.

Nichts an dieser Geschichte ist exotisch. Es gibt keinen Bug, keinen Störungsbericht, keinen Entwickler, der einen Fehler gemacht hätte. Ein Netzwerk hat getan, was Netzwerke tun, und das System hatte keine Möglichkeit zu erkennen, in welcher von zwei Welten es sich befand.

Das meiste, was in einem ereignisgesteuerten System schiefgeht, hat diese Form: Ein Schritt ist geschehen oder nicht, und der Beleg fehlt. Dieses Kapitel benennt die Arten, wie dieser Beleg verloren geht, damit Sie für jede gezielt entwerfen können. Die Frage, die es beantwortet, lautet: Was genau kann in einem System wie dem von Parcelly kaputtgehen, was sehen Sie, wenn es geschieht, und wo in diesem Buch wird jeder Ausfall behandelt?

Das Problem und warum es schwierig ist

Ein Ausfallmodell ist eine schriftliche Liste dessen, was Ihrer Annahme nach schiefgehen kann, und dessen, was Ihrer Annahme nach nicht schiefgehen kann. Jedes System hat eines, ob es jemand aufgeschrieben hat oder nicht. Die ungeschriebene Fassung enthält meist zwei Einträge: „Der Service stürzt ab“ und „Das Netzwerk ist langsam“. Beide sind real, aber es sind die Ausfälle, die sich von selbst bemerkbar machen.

Der einfache Fall ist ein Ausfall, den man erkennen kann, etwa eine abgewiesene Verbindung oder ein Datenbankfehler. Jeder sagt Ihnen, dass der Schritt nicht abgeschlossen wurde.

Der schwierige Fall ist der Ausfall, der Sie im Ungewissen lässt. Ein Timeout sagt nicht „Die Operation ist fehlgeschlagen“. Er sagt „Ich habe aufgehört zu warten“. Die Operation kann fehlgeschlagen sein, sie kann erfolgreich gewesen sein, oder sie läuft noch und gelingt gleich, nachdem Sie aufgegeben haben. Es genügen ein Timeout und ein Seiteneffekt am anderen Ende, um das zu spüren.

Deshalb trägt der naheliegende Ansatz nicht, Code für den Happy Path zu schreiben und um jeden Aufruf ein `try` und einen Retry zu legen. Ein Retry ist eine Entscheidung über die Zukunft, getroffen ohne Kenntnis der Vergangenheit. Hatte der frühere Versuch Erfolg, ist der Retry ein Duplikat. Verzichten Sie auf den Retry, geht womöglich eine Nachricht verloren. Keine der Entscheidungen ist an sich falsch; jede ist falsch in irgendeiner Welt, die Sie von der, in der Sie waren, nicht unterscheiden konnten.

Es gibt einen zweiten Grund. Ausfälle treten kombiniert auf: Ein Deployment startet einen Consumer neu, während ein Broker ein Failover durchführt und ein Producer einen Retry ausführt. Sie einzeln aufzulisten ist weiterhin der richtige Anfang, aber die Produktion liefert sie gemeinsam.

Die Arbeit dieses Kapitels ist also Klassifikation. Für jeden Ausfall stelle ich die drei Fragen aus Kapitel 1: Was kann an dieser Nahtstelle ausfallen, wie würden Sie es beobachten, und welche Garantie übersteht es? In den folgenden Einträgen beantwortet „Was tatsächlich wahr ist“ die erste, „Was Sie sehen“ die zweite, und das am Ende jedes Eintrags genannte Pattern beantwortet die dritte. Zusammengenommen sind diese drei Antworten die Ausfallsemantik, und der Rest des Buches fragt immer wieder danach.

Zehn Ausfälle, auf die es ankommt

Die folgende Liste ist die Menge, mit der ich bei einem System aus Services, relationalen Datenbanken und einem Message-Broker beginnen würde. Sie ist nicht vollständig, und keine Liste ist es.

Netzwerkpartitionen und Timeout-Ungewissheit

Das Eingangsszenario ist der Musterfall. Zwei Parteien sind über ein Netzwerk verbunden, und das Netzwerk verwirft, verzögert oder vertauscht Datenverkehr. Eine **Netzwerkpartition** ist die extreme Form, in der zwei Gruppen von Maschinen einander überhaupt nicht erreichen. Ein Timeout ist die alltägliche Form, und sie ist weitaus häufiger.

Was jede Seite beobachten kann, ist asymmetrisch. Der Aufrufer sieht nur, dass keine Antwort eintraf. Der Aufgerufene hat die Arbeit möglicherweise erledigt und die Antwort verloren, oder er hat die Anfrage nie gesehen. Dasselbe gilt für jeden Hop bei Parcelly: einen Service, der einen externen Paketdienst aufruft, einen Producer, der an den Broker sendet, einen Consumer, der gegenüber dem Broker per Ack bestätigt.

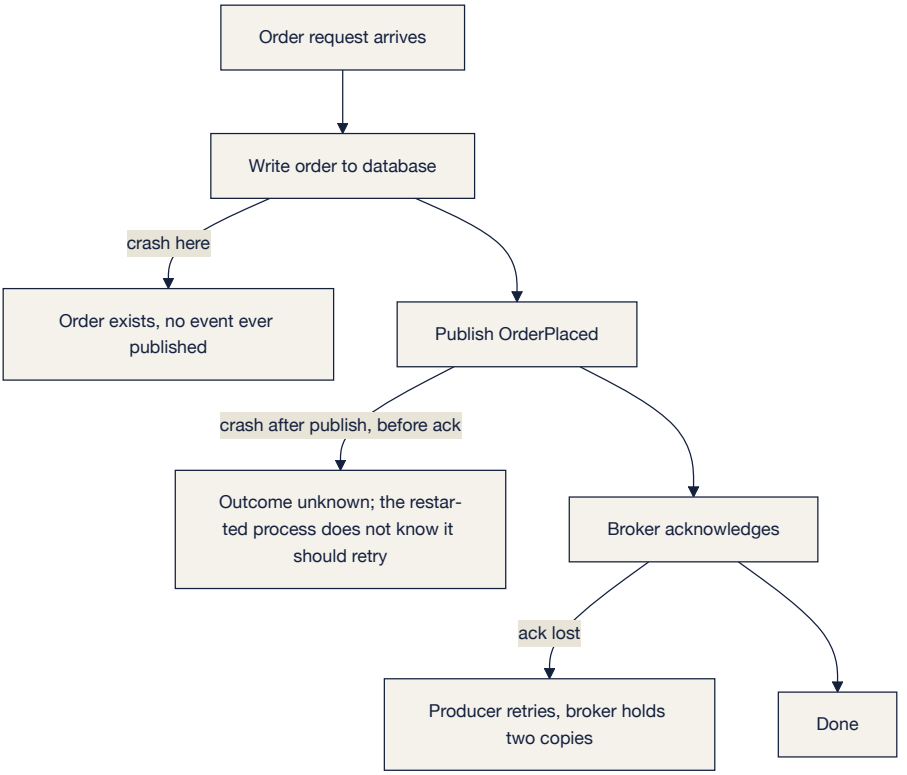
Die einzig ehrliche Antwort darauf ist, die Operation wiederholungssicher zu machen. Sendet Payments dem Anbieter bei jedem Einzug einen stabilen Idempotenzschlüssel, und behandelt der Anbieter einen wiederholten Schlüssel als dieselbe Anfrage, wird ein Retry nach einem Timeout harmlos. Das Design hängt nicht mehr von der Antwort auf „Ist es passiert?“ ab, weil beide Antworten im selben Endzustand münden. Kapitel 6 behandelt, wann und wie man wiederholt, und Kapitel 7 behandelt die Idempotenz, die Retries sicher macht.

Manche externen Systeme bieten keinen solchen Schlüssel. Dann fragen Sie das System, was geschehen ist, bevor Sie erneut handeln. Das ist ein Abgleich-Schritt, und Kapitel 13 behandelt ihn.

Absturzfenster des Producers

Betrachten Sie Orders bei der Verarbeitung einer neuen Bestellung. Der Service muss die Bestellung in seiner Datenbank speichern und `OrderPlaced` an den Broker veröffentlichen. Das sind zwei Systeme, und keine Transaktion umspannt sie. Das ist der Dual Write aus Kapitel 1, und das Interessante ist, was jedes Absturzfenster hinterlässt.

Der Prozess kann zwischen den beiden Schreibvorgängen abstürzen, und die Reihenfolge der Schreibvorgänge entscheidet, welchen Schaden Sie bekommen.



Das Diagramm zeichnet die Reihenfolge „Datenbank zuerst“ und zeigt drei Fenster: einen Absturz vor dem Publish, einen Absturz danach mit unbekanntem Ausgang und ein verlorener Ack. Erfolgt der Datenbank-Commit zuerst und der Prozess stirbt vor dem Publish, existiert die Bestellung, und niemand weiter unten erfährt davon. Payments autorisiert nie, Inventory reserviert nie, und der Käufer wartet auf eine Bestätigung, die nie kommt. Nichts meldet einen Fehler. Die Bestellung liegt einfach da.

Die umgekehrte Reihenfolge, die das Diagramm nicht zeichnet, hat ihr eigenes Fenster. Veröffentlichen Sie zuerst, hinterlässt ein Absturz vor dem Commit einen schlimmeren Zustand: ein Ereignis, das eine Tatsache verkündet, die nie wahr wurde. Nachgelagerte Services handeln für eine Bestellung, die nicht existiert. Und in beiden Reihenfolgen gilt: Geht der Ack des Brokers auf dem Rückweg verloren, wiederholt der Producer den Publish, und der Broker hält womöglich zwei Kopien desselben Ereignisses.

Von außen sehen Sie Ruhe: Bestellungen ohne Ereignisse, Ereignisse ohne Bestellungen und gelegentliche Duplikate. Nichts in den Logs sagt „verloren“. Kapitel 4 führt die Transactional Outbox ein, die das Fenster schließt, indem sie die Zustandsänderung und die Absicht zu veröffentlichen zu einer lokalen Transaktion macht. Kapitel 5 behandelt, wie die Outbox-Zeilen anschließend zum Broker gelangen.

Broker-Failover

Broker werden repliziert, damit eine Maschine sterben kann, ohne die Daten mitzunehmen. Stirbt die Maschine, die die führende Kopie einer Partition oder Warteschlange hält, übernimmt eine andere. Das ist ein **Failover**, und es ist ein vorgesehenes Verhalten, keine Fehlfunktion.

Entscheidend ist, was dem Producer vor dem Failover gesagt wurde. Ob eine Nachricht, die kurz vor dem Tod eines Leaders bestätigt wurde, überlebt, hängt davon ab, wie der Broker für das Bestätigen konfiguriert war: gar nicht, nachdem der empfangende Knoten sie angenommen hat (gleichgültig, ob er sie schon auf die Platte geschrieben hat), oder nachdem eine geforderte Zahl von Replikaten sie hat. Die stärksten Einstellungen kosten Latenz und Durchsatz. Die schwächsten Einstellungen können bestätigte Nachrichten bei einem Failover verlieren. Das unterscheidet sich zwischen Broker-Familien und zwischen Konfigurationen, daher behaupte ich hier kein bestimmtes Verhalten für ein bestimmtes Produkt; der Punkt ist, dass die Garantie eine Grenze hat und die Grenze eine Einstellung ist, die jemand gewählt hat.

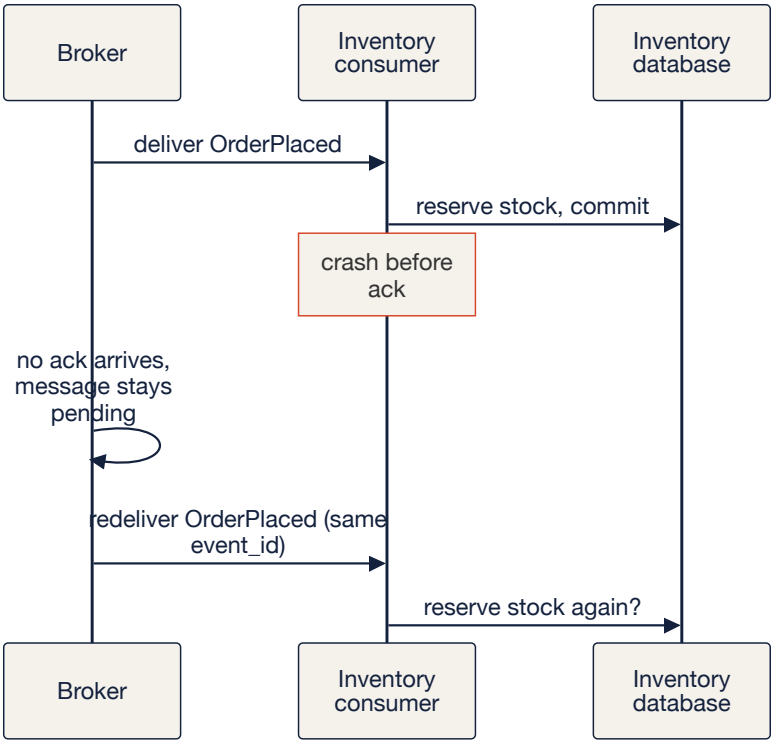
Während eines Failovers sehen Producer außerdem Fehler und Timeouts, wiederholen und veröffentlichen unter Umständen Duplikate von Nachrichten, die der alte Leader bereits angenommen hatte. Kapitel 2 beschreibt, wo die Garantien des Brokers enden. Die Regel hier lautet, „vom Broker bestätigt“ als starken Beleg zu behandeln und nicht als Beweis.

Was Sie sehen: eine Häufung von Publish-Fehlern, einen Einbruch beim Durchsatz der Consumer, dann eine Welle erneuter Zustellungen. Ist eine Nachricht verloren gegangen, sehen Sie gar nichts, und deshalb ist der Abgleich aus Kapitel 13 wichtig.

Absturz eines Consumers: verarbeitet, aber nicht per Ack bestätigt

Nun die Consumer-Seite. Inventory empfängt `OrderPlaced`, reserviert in seiner Datenbank den Bestand und führt den Commit aus. Dann stirbt der Prozess, bevor er die Nachricht per Ack bestätigen kann.

Aus Sicht des Brokers wurde die Nachricht nie abgeschlossen. Bei einem Broker auf Warteschlangenbasis bemerkt der Broker die geschlossene Verbindung, oder der Visibility-Timeout läuft ab, und die Nachricht wird erneut zugestellt. Bei einem Broker auf Log-Basis wurde der Offset des Consumers nie per Commit festgeschrieben, daher beginnt der nächste Consumer, dem die Partition gehört, an der alten Position und liest die Nachricht erneut. In beiden Familien trifft die zweite Zustellung für Arbeit ein, die bereits erledigt ist.



Die letzte Zeile des Diagramms ist das ganze Problem. Ein Consumer, der bei jeder Zustellung bedingungslos Bestand reserviert, reserviert ihn zweimal. Das ist die gewöhnliche Quelle von Duplikaten, und es ist der Grund,

warum die Voreinstellung des Buches At-least-once-Zustellung in Kombination mit idempotenten Consumern ist. Ein Consumer kann an verschiedenen Punkten abstürzen, und jeder Punkt hat eine andere Folge:

```
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.
```

Das ist eine kompakte Fassung des Handlers aus Kapitel 2, gekürzt, um die Absturzpunkte zu zeigen. Die Wirkung von Absturzpunkt A ist nichts. Die Wirkung von Absturzpunkt C ist nichts. Der interessante ist B, und ihn übersteht nur ein Consumer, der in derselben Transaktion wie die Arbeit festhält, was er verarbeitet hat. Kapitel 7 baut das ordentlich auf.

Was Sie sehen: doppelte Seiteneffekte wie zwei Reservierungen, zwei E-Mails, zwei Ledger-Einträge, und nirgends einen Fehler, weil jeder Verarbeitungsversuch erfolgreich war.

Langsame Consumer und Backpressure

Nicht jeder Ausfall ist ein Absturz. Ein Consumer, der bloß langsam ist, kann ebenso viel Schaden anrichten, und es fällt schwerer, ihn zu bemerken, weil nichts kaputt ist.

Angenommen, Notifications ruft einen E-Mail-Anbieter auf, der pro Anfrage 10 Sekunden braucht statt 100 Millisekunden. Notifications funktioniert weiterhin, aber jede Nachricht belegt nun einen Worker 100-mal so lange. Nachrichten treffen schneller ein, als sie fertig werden, daher wächst der Consumer-Lag, den Kapitel 2 definiert hat, stetig.

Langsamkeit löst dann sekundäre Ausfälle aus. Bei einem Broker auf Warteschlangenbasis erscheint eine Nachricht, die ihren Visibility-Timeout überschreitet, wieder, während der erste Worker sie noch bearbeitet, sodass ein zweiter Worker dieselbe Nachricht beginnt. Die Duplikate fügen einem ohnehin langsamen System Last hinzu. Bei einem Broker auf Log-Basis kann ein Consumer, der sich nicht rechtzeitig beim Broker zurückmeldet, für tot erklärt und seine Partitionen neu zugewiesen werden, womit der nächste Ausfall dieser Liste beginnt.

Backpressure ist die Disziplin, die laufende Arbeit in jeder Stufe zu begrenzen, damit die Last endlich bleibt. In einem brokervermittelten System fängt der Broker die Differenz auf, was der Vorteil der zeitlichen Entkopplung ist, doch der Puffer ist endlich und geht auf zwei verschiedene Arten zur Neige. Läuft die Platte des Brokers voll, drosselt oder blockiert er womöglich Producer oder stellt den Betrieb ganz ein, und welches davon Ihr Broker tut, sollten Sie herausfinden, bevor der Tag kommt. Was immer es ist, der Producer sieht es, und das ist echter Backpressure: laut und sichtbar. Fällt ein Consumer weiter zurück als die Aufbewahrungsdauer eines Logs, oder überlebt eine Nachricht in der Warteschlange ihren Ablauf, löscht der Broker Nachrichten, die nie gelesen wurden. Das ist keine Verzögerung. Das ist stiller Verlust: Es wird kein Fehler ausgelöst, kein Consumer schlägt fehl, und die ungelesenen Ereignisse `PaymentCaptured` sind weg. Lange bevor eine der beiden Grenzen erreicht ist, kann ein Backlog bereits eine geschäftliche Frist verletzen, etwa eine Zahlungsautorisierung, die abläuft, bevor sie eingezogen wird.

Weil der Puffer des Brokers das Problem nur aufschiebt, muss die Last an mehreren Stellen zugleich begrenzt werden. Ein Consumer kann die Zahl der Nachrichten in Bearbeitung begrenzen oder den Batch, den er vorab holt. Er kann die Parallelität seiner Worker begrenzen, sodass ein langsamer E-Mail-Anbieter eine feste Zahl von Workern bindet und nicht alle. Er kann seine Aufrufe in Richtung der langsamen Abhängigkeit mit einer Rate begrenzen oder den Konsum ganz pausieren, während sich die Abhängigkeit erholt. Eine Warteschlange kann eine Längenbegrenzung mit einer expliziten Überlaufrichtlinie haben, sodass Sie wählen, was verworfen oder abgewiesen wird, statt es den Broker entscheiden zu lassen.

Auch die Producer-Seite und die Topologie setzen Grenzen. Auf der Producer-Seite fängt die Outbox einen Broker auf, der Publishes ablehnt, weil die Zeilen in der Datenbank warten, bis der Relay sie zustellen kann. Und bei einem Broker auf Log-Basis begrenzt die Zahl der Partitionen, wie viele Consumer einer Gruppe parallel arbeiten können, sodass zusätzliche Instanzen ab einer im Voraus festgelegten Zahl nicht mehr helfen. Jede dieser Grenzen tauscht Durchsatz gegen Stabilität, und jede braucht eine Metrik, sonst wissen Sie nicht, dass gerade sie Sie ausbremst.

Was Sie sehen: steigenden Lag und steigendes Alter der Nachrichten, ein wachsendes Backlog, steigende Zahlen erneuter Zustellungen. Kapitel 6 behandelt Backoff, Limits für Nachrichten in Bearbeitung und das Pausieren des Konsums, Kapitel 8 behandelt, was mit Nachrichten zu tun ist, die nicht rechtzeitig verarbeitet werden können, und Kapitel 13 behandelt die Metriken für Lag und Alter, die Sie frühzeitig warnen.

Nachrichten, die jedes Mal fehlschlagen, und Abhängigkeiten, die für jede Nachricht ausfallen

Bisher war jeder Ausfall eine Frage des Zufalls: ein Absturz, eine verlorene Antwort, ein ungünstiger Moment. Manche Ausfälle sind deterministisch. Eine **Poison Message** ist eine Nachricht, die bei jedem Versuch fehlschlägt, gleich welches Timing. Angenommen, bei Inventory trifft ein `OrderPlaced` mit einer SKU ein, die in seinem Katalog nicht existiert. Inventory löst einen Fehler aus, die Nachricht wird erneut zugestellt, der Fehler wiederholt sich, und nichts am nächsten Versuch unterscheidet sich vom letzten.

Die Ursache kann auch außerhalb der Nachricht liegen. Ist der Anbieter von Payments ausgefallen, schlägt jeder Einzug fehl, und jede Nachricht sieht giftig aus, solange der Ausfall dauert. Die beiden Fälle verlangen gegensätzliche Reaktionen. Eine Nachricht, die nie gelingen wird, sollte den Hauptfluss schnell verlassen, und zwar in eine **Dead-Letter-Queue (DLQ)**, einen separaten Ort, an dem Nachrichten, die nicht verarbeitet werden können, auf einen Menschen oder einen Reparaturjob warten. Eine Nachricht, die fehlschlägt, weil eine Abhängigkeit ausgefallen ist, sollte gar nicht in die Dead Letters, denn sie würde nach der Wiederher-

stellung gelingen, und ein ganzes Backlog während eines Ausfalls in die Dead Letters zu schicken verwandelt ein vorübergehendes Problem in ein manuelles.

Was dazwischen schiefgeht, folgt einem Muster. Retries bei einer Poison Message verbrauchen Versuche und Kapazität ohne jede Aussicht auf Erfolg. Bei einem Broker auf Log-Basis oder einer Warteschlange, die die Reihenfolge wahrt, kann eine einzige solche Nachricht alles dahinter blockieren. Viele Consumer, die gegen eine einzige ausgefallene Abhängigkeit wiederholen, fügen dem, was sich gerade zu erholen versucht, Last hinzu: ein Retry-Storm. Und ein Bug im Code kann ohne jeden Fehler fehlschlagen: Der Consumer bestätigt per Ack, kehrt zurück und schreibt den falschen Wert, sodass im Fehlerpfad nichts davon je läuft.

Was Sie sehen: dieselbe `event_id`, die immer wieder in den Logs erscheint, Zahlen erneuter Zustellungen, die nie sinken, eine festsitzende Partition, eine DLQ, die während eines Ausfalls wächst, und beim stillen Bug gar nichts, bis eine Abgleichsprüfung widerspricht. Kapitel 6 behandelt, wie man erkennt, bei welchen Ausfällen sich ein Retry lohnt, Kapitel 8 behandelt Poison Messages und den Dead-Letter-Pfad, und die Kapitel 13 und 14 behandeln, wie man den Schaden eines Bugs findet und repariert, der keinen Fehler ausgelöst hat. Der Preis jeder Klassifikation ist, dass sie falsch sein kann, daher sollten Sie die Entscheidung sichtbar und umkehrbar halten.

Uhren und Zeitstempel, denen Sie nicht trauen können

Jede Maschine hat eine Uhr, und keine zwei Uhren stimmen exakt überein. Sie driften, sie werden durch Synchronisierung korrigiert, und eine Korrektur kann eine Uhr rückwärts setzen. Eine pausierte virtuelle Maschine wacht mit einer Uhr auf, die um genau die Dauer ihres Schlafs falsch geht.

Die Folge für Ereignisse ist konkret. `occurred_at` ist eine Tatsache darüber, wann etwas geschah, laut einer einzelnen Maschine. Vergleicht man das `occurred_at` eines Ereignisses von Orders mit dem eines Ereignisses von Payments, erfährt man ebenso viel über ihre Uhren wie über die Reihenfolge der Ereignisse. Sortieren Sie nach Zeitstempel, um zu entscheiden, welches Update gewinnt, gewinnt ein Service, dessen Uhr einige Sekunden vorgeht, jeden Konflikt, den er verlieren müsste.

Timeouts haben eine verwandte Falle. Eine Frist, die als Wall-Clock-Zeit ausgedrückt ist, „diese Autorisierung ist bis 14:00 Uhr gültig“, wird von einer Maschine verglichen, deren Vorstellung von 14:00 Uhr von der des Ausstellers abweichen kann. Zum Messen von Zeitspannen auf einer Maschine ist eine monotone Uhr, die sich nur vorwärts bewegt, das richtige Werkzeug. Für die Reihenfolge über Maschinen hinweg würde ich Wall-Clock-Zeit überhaupt nicht verwenden.

Was Ereignisse verlässlich ordnet, ist die Struktur des Systems: bei einem Broker auf Log-Basis eine nach dem Aggregat geschlüsselte Partition, die eine totale Ordnung für Ereignisse mit diesem Schlüssel liefert, solange es für jeden Schlüssel einen einzigen Producer-Pfad gibt und sich die Zahl der Partitionen nicht ändert, oder eine explizite Versionsnummer am Aggregat, die Consumer vergleichen können. Kapitel 9 behandelt die Reihenfolge ausführlich. Behandeln Sie `occurred_at` als Information für Menschen und für Analysen, und seien Sie misstrauisch, wenn es als Eingabe für die Korrektheit dient.

Was Sie sehen: Updates, die in falscher Reihenfolge angewendet werden, „unmögliche“ Abfolgen in Logs, in denen eine Wirkung ihrer Ursache vorauszugehen scheint, und Bugs, die nur auf einem Host auftreten.

Deployments und Rebalances

Ein Deployment ist ein geplanter Ausfall. Instanzen werden gestoppt und gestartet, und in der Zwischenzeit wird Arbeit mittendrin unterbrochen. Sendet Ihr Deployment-Prozess ein Beendigungssignal und der Consumer beendet sich mit Nachrichten in Bearbeitung, haben Sie den Fall „verarbeitet, aber nicht per Ack bestätigt“ absichtlich herbeigeführt, wiederholt, bei jedem Release.

Ein **Rebalance** ist die Version der Log-Familie, und es kann auch ohne Deployment auftreten. Treten Mitglieder einer Consumer-Group bei oder verlassen sie, oder wird ein Mitglied als nicht reagierend eingestuft, werden Partitionen neu zugewiesen. Bei der Übergabe können kurzzeitig zwei Consumer glauben, dieselbe Partition zu besitzen, und ein Consumer, der langsam war, kann Nachrichten weiterverarbeiten, nachdem seine Partition längst woanders liegt. Das Gegenstück in der Familie der Warteschlangen sind konkurrierende Consumer, deren Visibility-Timeouts ablaufen, während die Arbeit weiterläuft.

Das Schadensmuster ist eine Häufung von Duplikaten zur Deployment-Zeit und manchmal ein kurzer Stillstand, während Partitionen neu zugewiesen werden. Ein Deployment, das in rascher Folge durch alle Instanzen rollt, kann mehrere Rebalances hintereinander auslösen, und im schlimmsten Fall verbringen Consumer mehr Zeit damit, Arbeit neu zu verteilen, als sie zu erledigen. Zwei Maßnahmen verringern das, zum Preis von mehr Konfiguration, die man verstehen muss: kooperative Neuzuweisung, bei der nur die Partitionen entzogen werden, die wechseln müssen, und eine stabile Mitgliedsidentität, damit eine neu gestartete Instanz ihre alten Partitionen ohne vollständiges Neumischen zurückfordern kann.

Sorgen Sie dafür, dass Consumer sauber anhalten, indem sie Nachrichten in Bearbeitung vor dem Beenden abschließen oder freigeben, und gehen Sie davon aus, dass der saubere Pfad manchmal scheitert, sodass der idempotente Consumer die eigentliche Verteidigung bleibt. Ich würde außerdem schrittweise Deployments zeitlich staffeln, damit sie keine Neuzuweisungen in enger Schleife auslösen. Was Sie sehen: Duplikate und Lag-Spitzen, die mit den Release-Zeitpunkten zusammenfallen.

Menschliche Fehler

Die letzte Kategorie ist die, die die meisten Architekturdokumente auslassen. Menschen betreiben diese Systeme, und Menschen machen unter Druck Fehler, meist beim Reparieren eines früheren Ausfalls.

Stellen Sie sich den Bereitschaftsingenieur bei Parcelly vor, der 40.000 Dead Letters findet und den Knopf drückt, um sie alle per Redrive erneut einzuspielen. Wurde die Ursache behoben, laufen sie durch. Wurde sie nicht behoben, schlagen sie sofort wieder fehl und überfluten das System, das der Ingenieur heilen wollte. Wurden einige bereits durch einen manuellen Workaround erledigt, sind sie Duplikate von Arbeit, die jemand von Hand getan hat.

Andere Formen sind ebenso alltäglich. Jemand bearbeitet eine Zeile direkt, um einen Kunden zu entsperren, und umgeht die Outbox, sodass kein Ereignis veröffentlicht wird und nachgelagerte Services auseinanderlaufen. Jemand rollt einen Producer mit geändertem Payload aus, bevor Consumer ihn lesen können. Jemand leert eine Warteschlange in der falschen Umgebung. Das geschieht, wenn die Reparaturwerkzeuge breit sind und die Sicherungen fehlen.

Ich würde Aktionen von Operatoren als Teil des Ausfallmodells behandeln und wie jede andere Komponente entwerfen: Redrive in kleinen Batches mit einem Pausenschalter, Reparaturaktionen, die denselben Pfad nehmen wie normale Änderungen, damit Ereignisse ausgesendet werden, und jede manuelle Aktion in einem Prüfpfad festgehalten. Kapitel 8 behandelt Dead Letters und Redrive, Kapitel 12 behandelt Änderungen am Payload, und Kapitel 13 behandelt den Abgleich, mit dem Sie herausfinden, dass eine menschliche Korrektur schiefging.

Was Sie sehen: eine plötzliche Spitze direkt nachdem jemand etwas angefasst hat, und Daten, die zwischen Services nicht übereinstimmen, ohne dass eine fehlgeschlagene Nachricht es erklärt.

Verlust eines ganzen Stores

Jeder bisherige Ausfall nahm an, dass die Stores überleben: Die Datenbank behält, was sie per Commit festgeschrieben hat, und der Broker behält, was er bestätigt hat. Der letzte Eintrag gibt diese Annahme auf. Ein Cluster oder eine Region geht verloren. Ein Broker verliert Daten über das hinaus, was seine Replikation abdeckt. Die Datenbank eines Consumers wird aus einem Backup wiederhergestellt, das sechs Stunden alt ist. Die Datenbank eines Producers wird auf einen Zeitpunkt vor den letzten paar hundert Bestellungen zurückgesetzt.

Das ist nicht ein Ausfall, denn der Schaden hängt davon ab, welcher Store in der Zeit zurückgegangen ist. Wird die Datenbank von Inventory wiederhergestellt, vergisst sie Reservierungen, die nach dem Wiederherstellungszeitpunkt vorgenommen wurden, und der Broker hält womöglich die Ereignisse nicht mehr, die sie neu erzeugen würden. Wird die Datenbank von Orders wiederhergestellt, vergisst sie Bestellungen, auf die nachgelagerte Services bereits reagiert haben, und Payments hält Einzüge für Bestellungen, die Orders nicht mehr kennt. Keine Seite meldet einen Fehler. Jede ist in sich konsistent und falsch über die andere.

Eine Eigenschaft hilft. Weil die Inbox in derselben Datenbank liegt wie die Effekte, setzt eine Wiederherstellung beide auf denselben Zeitpunkt zurück. Nachrichten, die vor dem Wiederherstellungszeitpunkt verarbeitet wurden, stehen noch in der Inbox, daher überspringt ein Replay sie, was korrekt ist, weil ihre Effekte ebenfalls überlebt haben. Nachrichten, die

nach dem Wiederherstellungszeitpunkt verarbeitet wurden, sind die gefährdeten, und sie erneut abzuspielen ist sicher: Die Inbox erinnert sich nicht mehr an sie, und ihre Effekte gingen mit ihr verloren.

Externe Effekte sind die Ausnahme. Ein Einzug beim Zahlungsanbieter wird nicht mit Ihrer Datenbank zurückgerollt, daher brauchen diese Aufrufe eigene Idempotenzschlüssel, sonst belastet der Replay den Käufer ein zweites Mal. Die Regel lautet, vor dem Wiederaufnehmen des Konsums abzugleichen: Vergleichen Sie, was jeder Service glaubt, reparieren Sie die Unterschiede, und lassen Sie erst dann den Fluss laufen.

Was Sie sehen: zunächst nichts, und das ist das Problem. Die Services sind gesund und die Dashboards grün, während die Summen zwischen ihnen voneinander abweichen. Kapitel 13 behandelt den Abgleich, der die Abweichung findet, und Kapitel 14, Wiederherstellung, Katastrophen und Broker-Migration, behandelt, wie man wiederherstellt, abspielt und fortsetzt. Dafür zu entwerfen kostet Backups, die tatsächlich getestet werden, eine Aufbewahrung, die Ihre Wiederherstellungszeit überdauert, und die Disziplin, eine Wiederherstellung zu proben, bevor man sie braucht.

Die Ausfallkarte

Hier ist dasselbe Material an einer Stelle. Halten Sie es bei Ihren Design-Reviews griffbereit.

Ausfall	Was Sie sehen	Was tatsächlich wahr ist	Wo er behandelt wird
Netzwerkpartition oder Timeout	Aufrufer erhält keine Antwort	Die Operation ist womöglich gelaufen, womöglich nicht	Kapitel 6, Kapitel 7, Kapitel 13
Absturz des Producers zwischen Schreiben und Publish	Bestellungen ohne Ereignisse oder Ereignisse ohne Bestellungen	Zustand und Ereignis widersprechen sich	Kapitel 4, Kapitel 5
Verlorener Ack des Brokers	Doppelte Ereignisse im Broker	Der erste Publish war erfolgreich	Kapitel 7
Broker-Failover	Publish-Fehler, dann erneute Zustellungen; selten stiller Verlust	Die Garantie hängt von den Einstellungen für Acks ab	Kapitel 2, Kapitel 13

Ausfall	Was Sie sehen	Was tatsächlich wahr ist	Wo er behandelt wird
Absturz des Consumers nach der Arbeit	Doppelte Seiteneffekte, keine Fehler	Arbeit erledigt, Nachricht noch ausstehend	Kapitel 7
Langsamer Consumer	Wachsender Lag und wachsendes Alter der Nachrichten	Consumer lebt, fällt zurück	Kapitel 6, Kapitel 8, Kapitel 13
Aufbewahrung oder Ablauf bei Rückstand	Nichts; die fehlenden Ereignisse treffen nie ein	Ungelesene Nachrichten wurden vom Broker gelöscht	Kapitel 8, Kapitel 13
Poison Message oder ausfallende Abhängigkeit	Dieselbe Nachricht wird erneut zugestellt; festsitzende Partition; DLQ wächst bei einem Ausfall	Der Ausfall ist deterministisch, oder die Ursache liegt außerhalb der Nachricht	Kapitel 6, Kapitel 8
Bug im Consumer ohne Fehler	Nichts; die Summen weichen später ab	Der Handler war erfolgreich und schrieb das falsche Ergebnis	Kapitel 13, Kapitel 14
Uhrenabweichung	Updates in falscher Reihenfolge, unmögliche Abfolgen	Zeitstempel widersprechen der Kausalität	Kapitel 9
Deployment oder Rebalance	Duplikate und Lag zur Release-Zeit	Die Zuständigkeit für die Arbeit wechselte mittendrin	Kapitel 7, Kapitel 13
Fehler eines Operators	Spitze nach einer manuellen Aktion; stille Abweichung	Eine Reparatur änderte den Zustand außerhalb des normalen Pfads	Kapitel 8, Kapitel 13
Verlust eines ganzen Stores	Zunächst nichts; Services widersprechen sich nach einer Wiederherstellung	Ein Store ging in der Zeit zurück, oder seine Daten sind weg	Kapitel 13, Kapitel 14

Zwei Dinge fallen auf. Erstens erscheinen Duplikate in der Spalte „Was Sie sehen“ bei drei verschiedenen Ursachen (verlorener Ack, Absturz des Consumers, Deployment) und erneute Zustellungen bei einer vierten. Deshalb verdient ein einziges Pattern, der idempotente Consumer, ein ganzes Kapitel. Zweitens kehrt die letzte Spalte immer wieder zu den Kapiteln 13 und 14 zurück. Manche Ausfälle lassen sich nicht verhindern oder im Moment ihres Auftretens nicht einmal erkennen, daher braucht das System auch eine Möglichkeit, hinterher zu bemerken, dass zwei Sichten der Wahrheit voneinander abweichen.

Das Ausfallmodell aufschreiben

Ein Ausfallmodell ist nur nützlich, wenn es aufgeschrieben, aktuell und geteilt ist. Meiner Ansicht nach ist die Praxis wertvoller als jeder einzelne Eintrag, weil das Aufschreiben die Frage „Was nehmen wir hier an?“ erzwingt und in den Annahmen die Überraschungen stecken.

Ich würde es als kurzes Dokument im Repository führen, neben der Architekturbeschreibung, mit einem Eintrag pro Ausfall und jedes Mal denselben Feldern. Die Felder spiegeln die drei Fragen: „where“ und „we assume“ sagen, was ausfallen kann, „observable as“ sagt, wie Sie es sehen würden, und „defense“ sagt, welche Garantie überlebt. Ein brauchbarer Eintrag sieht so aus:

```
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
```

Die Felder, auf die es ankommt, sind die letzten drei. „Verified by“ macht den Eintrag von einer Überzeugung zu einer Behauptung, die sich testen lässt, und Kapitel 15 zeigt, wie man sie testet. „Owner“ bedeutet, dass jemand alarmiert wird. „Last reviewed“ räumt ein, dass das Modell verrottet: Services kommen hinzu, Broker werden ausgetauscht, und eine Annahme, die letztes Jahr stimmte, stimmt heute womöglich nicht mehr.

> **Hinweis:** Das Modell sollte auch festhalten, was Sie für unmöglich halten. „Wir nehmen an, dass die Datenbank nie eine festgeschriebene Transaktion verliert“ ist ein legitimer Eintrag. Es ist besser, die Annahme aufzuschreiben und zu wissen, was Sie täten, wenn sie falsch wäre, als sie implizit zu lassen.

Nutzen Sie das Modell an drei Stellen. Gehen Sie in Design-Reviews jeden neuen Fluss gegen die Liste durch und fragen Sie, welchen Ausfällen er ausgesetzt ist. Finden Sie in Incident-Reviews heraus, zu welchem Eintrag

der Incident gehört, und fügen Sie einen hinzu, falls keiner passt. Machen Sie beim Testen aus jedem Eintrag ein Szenario. Ein Ausfall, der im Modell steht, aber in keinem Test, ist eine Hoffnung.

Abwägungen und Kosten

Ein Ausfallmodell zu schreiben kostet Zeit und das Unbehagen, festzuhalten, was man nicht weiß. Ein kleines Team mit einem Service und einer Warteschlange braucht nicht jeden Eintrag dieser Liste. Beginnen Sie mit den drei oder vier Ausfällen, die am meisten wehtun würden.

Das Modell hat außerdem eine Grenze. Jede Verteidigung in diesem Buch nimmt an, dass manche Ausfälle außerhalb des Geltungsbereichs liegen: dass die Datenbank dauerhaft ist, dass die Replikation des Brokers wie konfiguriert funktioniert. Diese Annahmen sind vernünftig, und jede ist eine Stelle, an der das Modell falsch liegen kann. Sie aufzulisten lässt Sie entscheiden, ob sie akzeptabel sind, was nicht geht, solange sie unsichtbar sind. Ein Bug im eigenen Code ist etwas anderes. Er liegt im Geltungsbereich, als Ausfall, den man erkennen und von dem man sich erholen muss, weil er eintreten wird, und das Modell muss sagen, wie Sie ihn bemerken und reparieren würden, was er geschrieben hat.

Ein Modell mit 40 Einträgen wird nicht gelesen, und eines mit dreien übersieht Fälle. Ich würde eine Liste anstreben, die ein neues Teammitglied in einer Sitzung lesen kann.

Für jeden Ausfall mit maximaler Stärke zu entwerfen ist ebenfalls nicht das Ziel. Ein Broker-Failover ohne jeden Verlust zu überstehen kostet Latenz und Geld, und einen Doppelausfall zu überstehen kostet mehr. Eine E-Mail von Notifications, die gelegentlich zweimal versendet wird, rechtfertigt nicht dieselbe Verteidigung wie ein Ledger-Eintrag, der nie dupliziert werden darf. Die Aufgabe des Modells ist, Sie nach Risiko ausgeben zu lassen. Wenn ich die geschäftlichen Kosten eines Ausfalls nicht benennen kann, würde ich zögern, technische Kosten zu zahlen, um ihn zu verhindern.

Das Modell macht das System nicht zuverlässig. Es macht Ihre Annahmen überprüfbar.

Häufige Fehler

- **Nur Abstürze modellieren.** Ein Absturz ist verlockend zu modellieren, weil er sauber und erkennbar ist. Die teuren Ausfälle sind die mehrdeutigen, etwa Timeouts und langsame Consumer. Abhilfe: Schreiben Sie für jeden Aufruf auf, was der Aufrufer tut, wenn keine Antwort eintrifft.
- **Ein Timeout als Fehlschlag behandeln.** Es fühlt sich natürlich an, „keine Antwort“ als „ist nicht passiert“ zu lesen. Abhilfe: Behandeln Sie es als unbekannt, und entwerfen Sie die Operation so, dass ihre Wiederholung sicher oder prüfbar ist.
- **Dem Ack des Brokers als Beweis trauen.** Das Wort „bestätigt“ klingt endgültig. Abhilfe: Wissen Sie, was Ihre Einstellung für Acks tatsächlich garantiert, und gleichen Sie für die Fälle ab, in denen sie es nicht tut.
- **Nach Zeitstempel ordnen.** `occurred_at` liegt direkt im Envelope, und das Sortieren danach braucht eine Zeile. Abhilfe: Ordnen Sie innerhalb eines Partitionsschlüssels nach der Version des Aggregats, und heben Sie Zeitstempel für Menschen auf.
- **Deployments ignorieren.** Releases fühlen sich wie Routine an, nicht wie Ausfall. Abhilfe: Zählen Sie jedes Deployment als geplanten Absturz eines Consumers, und testen Sie das Herunterfahren, während Nachrichten in Bearbeitung sind.
- **Operatoren auslassen.** Reparaturwerkzeuge werden weniger geprüft als Produktcode. Abhilfe: Nehmen Sie Redrive, Replay und manuelle Änderungen ins Modell auf, mit Batch-Limits, Pausenschaltern und Prüfpfaden.
- **Eine wiederhergestellte Datenbank als sauberen Neustart behandeln.** Eine Wiederherstellung fühlt sich wie das Ende des Incidents an. Abhilfe: Gleichen Sie nach jeder Wiederherstellung mit den anderen Services ab, bevor der Konsum wieder aufgenommen wird, und prüfen Sie externe Effekte, die nicht zurückgerollt wurden.
- **Einen Ausfall wiederholen, der sich nicht ändern wird.** Retries fühlen sich gewissenhaft an. Abhilfe: Trennen Sie Ausfälle, die vom Timing abhängen, von denen, die von der Nachricht abhängen, und schicken Sie die zweite Art früh auf den Dead-Letter-Pfad.

- **Das Modell einmal schreiben.** Ein Dokument, das in einer Entwurfsphase entstand, ist eine Momentaufnahme. Abhilfe: Bestimmen Sie eine verantwortliche Person, prüfen Sie es nach Incidents, und datieren Sie jeden Eintrag.

Zentrale Erkenntnisse

- Ein Timeout bedeutet „Ich habe aufgehört zu warten“, nicht „Es ist fehlgeschlagen“, daher sollten Sie Operationen entwerfen, die korrekt bleiben, wie auch immer sich das Unbekannte auflöst.
- Jede Nahtstelle hat ihre eigenen Absturzfenster, und der Schaden ist still. Ein Dual Write hinterlässt Bestellungen ohne Ereignisse oder Ereignisse über Bestellungen, die nicht existieren, und ein Consumer, der nach dem Commit, aber vor dem Ack abstürzt, erhält bei jeder erneuten Zustellung ein Duplikat, was idempotente Consumer zur Pflicht macht.
- Der Ack eines Brokers ist ein Beleg mit einer Grenze, die durch die Einstellungen für Replikation und Flush gesetzt wird, daher sollten Sie die Grenze kennen und einplanen, was außerhalb liegt, abzugleichen.
- Langsame Consumer und Deployments verursachen Duplikate und Stillstände, ohne dass eine Komponente kaputt wäre, daher sollten Sie die Last begrenzen, Lag und Alter der Nachrichten messen und Consumer sauber anhalten.
- Eine Nachricht, die jedes Mal fehlschlägt, oder eine Abhängigkeit, die für jede Nachricht ausfällt, braucht eine andere Reaktion als ein zufälliger Ausfall, daher sollten Sie klassifizieren, bevor Sie wiederholen, und das Dead-Lettering dem vorbehalten, was sich nicht erholen wird.
- Wall-Clock-Zeitstempel stimmen über Maschinen hinweg nicht überein, daher sollten Sie Ereignisse innerhalb eines Partitionsschlüssels nach der Version des Aggregats ordnen und `occurred_at` für Menschen verwenden.
- Operatoren und Wiederherstellungen sind Teil des Systems: Ein wiederhergestellter Store ist in der Zeit zurückgegangen, daher gleichen Sie ab, bevor Sie den Konsum wieder aufnehmen, und führen ein schriftliches Ausfallmodell, das Verantwortliche, Tests und Review-Daten benennt, damit Reparaturen nicht zum nächsten Incident werden.

Der erste Ausfall, den es zu beseitigen gilt, ist der im Producer, wo eine Zustandsänderung und ihr Ereignis auseinanderfallen können, ohne dass es jemand bemerkt. Kapitel 4 eröffnet Teil II mit der Transactional Outbox, die dieses Fenster schließt.

TEIL II

Zuverlässig veröffentlichen

Kapitel 4 — Das Dual-Write-Problem und die Transactional Outbox

Stellen Sie sich Parcelly einige Wochen nach dem Start der Verkäufer-Aktionen vor. Eine Entwicklerin im Support öffnet ein Ticket, das vollständig lautet: „Käuferin sagt, sie habe den Checkout für Bestellung 81724 abgeschlossen, aber der Verkäufer habe nie ein Versandetikett erhalten. Die Bestellung wird als aufgegeben angezeigt.“ Die Entwicklerin prüft die Orders-Datenbank und findet die Zeile mit dem Status `PLACED`, angelegt um 14:02:11. Dann prüft sie Payments, und dort gibt es keine Autorisierung für diese Bestellung. Auch Inventory hat keine Reservierung. Aus Sicht des restlichen Parcelly existiert die Bestellung nicht.

In den Logs von Orders rund um 14:02 findet sie einen Deployment-Marker. Der Service wurde 200 Millisekunden nach dem Commit der Bestellung neu gestartet, noch bevor er `OrderPlaced` veröffentlicht hatte. Der Code tat genau das, wofür er geschrieben wurde: speichern, dann veröffentlichen. Nichts warf einen Fehler, denn der Prozess, der ihn geworfen hätte, war bereits verschwunden. Nehmen wir an, eine schnelle Abfrage zeigt 11 weitere Bestellungen derselben Art, alle innerhalb derselben Minute.

Das Team korrigiert die 11 von Hand und ergänzt das Runbook um eine Notiz. Gut fühlt sich das niemand. Der Code ist nicht auf eine Weise falsch, die ein Reviewer beanstanden würde, und doch kann er Arbeit verlieren, sobald ein Prozess im falschen Moment stirbt. Eine größere Maschine ändert daran nichts, und eine sorgfältigere Entwicklerin ebenso wenig. Die Frage dieses Kapitels lautet: Kann ein Service seine eigene Datenbank ändern und den Rest des Systems darüber informieren, und zwar so, dass beides nicht auseinanderfallen kann, ohne eine Transaktion, die Datenbank und Broker umspannt?

Das Problem und warum es schwierig ist

Kapitel 1 hat den Dual Write definiert: zwei Systeme werden ohne gemeinsame Transaktion aktualisiert. Orders muss eine Zeile in seine relationale Datenbank schreiben und eine Nachricht an den Broker veröffentlichen. Die Datenbank garantiert Atomarität für das Erste. Der Broker garantiert in einer bestimmten Konfiguration Dauerhaftigkeit für das Zweite. Nichts garantiert, dass beides geschieht oder keines von beiden.

Kapitel 3 hat die Absturzfenster des Producers in einem Diagramm skizziert. Hier möchte ich sie als Tabelle durchgehen, denn die Reihenfolge der beiden Schreibvorgänge ist eine Designentscheidung, die die meisten Teams unbemerkt treffen, und hinter jeder Wahl wartet ein Ausfall.

Jede Reihenfolge geht schief

Nehmen wir die beiden naheliegenden Abläufe und fragen für jede mögliche Ausfallstelle, wie die Welt danach aussieht. Das ist dieselbe Übung wie das Aufschreiben der Ausfallsemantik der Nahtstelle zwischen Datenbank und Broker: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall überlebt. Die folgende Tabelle ist diese Übung für den Producer.

Ablauf	Ausfallstelle	Zustand der Datenbank	Zustand des Brokers	Schaden
Erst Commit, dann veröffentlichen	Absturz nach dem Commit, vor dem Veröffentlichen	Bestellung gespeichert	Kein Ereignis	Verlorenes Ereignis. Nachgelagerte Systeme erfahren nie von der Bestellung
Erst Commit, dann veröffentlichen	Veröffentlichen schlägt fehl oder läuft in einen Timeout, Prozess überlebt	Bestellung gespeichert	Unbekannt	Der Aufrufer muss einen Retry durchführen oder aufgeben. Aufgeben verliert das Ereignis
Erst Commit, dann veröffentlichen	Broker nimmt an, Ack geht verloren	Bestellung gespeichert	Ereignis vorhanden	Ein Retry veröffentlicht ein Duplikat

Ablauf	Ausfallstelle	Zustand der Datenbank	Zustand des Brokers	Schaden
Erst veröffentlichen, dann Commit	Absturz nach dem Veröffentlichen, vor dem Commit	Keine Bestellung	Ereignis vorhanden	Phantomereignis. Nachgelagerte Systeme handeln auf Grundlage eines Faktums, das nie wahr wurde
Erst veröffentlichen, dann Commit	Commit schlägt fehl (Constraint, Deadlock, Timeout)	Keine Bestellung	Ereignis vorhanden	Dasselbe Phantomereignis, jetzt mit einem sichtbaren Fehler im Producer
Veröffentlichen innerhalb der Transaktion, Commit zuletzt	Commit schlägt nach dem Veröffentlichen fehl	Keine Bestellung	Ereignis vorhanden	Der Transaktionsblock vermittelt dem Code ein falsches Gefühl von Sicherheit, und der Broker ist nicht Teil davon

Die letzte Zeile verdient einen Kommentar, denn sie sieht am sichersten aus. Viele Frameworks erlauben es, aus einem Datenbank-Transaktionsblock heraus zu veröffentlichen. Es liest sich, als wäre das Veröffentlichen Teil der Transaktion. Das ist es nicht. Der Aufruf an den Broker geschieht sofort und lässt sich nicht zurückrollen, und der Commit folgt erst danach. Das ist die Reihenfolge „erst veröffentlichen, dann Commit“ mit schönerer Einrückung.

Das Spiegelbild ist der „After-Commit“-Callback, den viele Frameworks anbieten, ein Hook, der läuft, sobald die Transaktion festgeschrieben wurde. Teams halten ihn oft für sicher, weil nur für festgeschriebene Daten veröffentlicht wird. Es ist die erste Zeile der Tabelle: erst Commit, dann veröffentlichen, wobei die Absicht im Arbeitsspeicher des Prozesses liegt. Ein Absturz zwischen dem Commit und dem Hook verliert das Ereignis, und nichts hält fest, dass es je geschuldet war.

Es gibt auch eine vierte Form, die die Tabelle nicht erfasst, nämlich einen langsamen Broker. Wenn das Veröffentlichen innerhalb der Transaktion stattfindet, hält die Datenbank Sperren und eine Verbindung, während sie auf den Broker wartet. Ein Broker, der eine Minute lang langsam ist, wird zu einer Datenbank, die eine Minute lang langsam ist. Sie haben die Verfügbarkeit zweier Systeme gekoppelt, die eigentlich unabhängig sein sollten.

Warum ein Retry nicht hilft

Die naheliegende Reparatur ist eine Retry-Schleife um das Veröffentlichen. Retries helfen bei der zweiten Zeile der Tabelle, in der der Prozess überlebt und es erneut versuchen kann. Bei der ersten Zeile, in der der Prozess weg ist, helfen sie nicht, und sie erzeugen das Duplikat in der dritten Zeile. Kapitel 6 behandelt Retries ausführlich. Vorerst gilt: Eine Retry-Schleife lebt im Arbeitsspeicher des Prozesses, und der Ausfall, um den es uns geht, ist das Verschwinden des Prozesses.

Die zweite naheliegende Reparatur ist ein periodischer Job, der die Datenbank mit dem Broker vergleicht und erneut veröffentlicht, was fehlt. Manche Teams bauen das, und ich halte es für ein vernünftiges Auffangnetz, auf das Kapitel 13 als Abgleich zurückkommt. Als primärer Mechanismus hat es zwei Probleme. Der Broker lässt sich selten nach fachlichem Schlüssel abfragen, sodass „was fehlt“ zu einer teuren Schätzung wird. Und der Job führt selbst denselben Dual Write aus, kann also auf dieselbe Weise fehlschlagen, nur seltener und mit mehr Verwirrung.

Warum keine verteilte Transaktion

Wenn das Problem zwei Systeme ohne gemeinsame Transaktion sind, lautet die klassische Antwort, ihnen eine zu geben. Der **Zwei-Phasen-Commit (2PC)**, für Datenbanken und Transaktionsmanager als XA standardisiert, koordiniert mehrere Resource-Manager. In der ersten Phase bereitet sich jeder Teilnehmer vor, das heißt, er verspricht, festschreiben zu können, und hält seine Sperren. In der zweiten Phase weist ein Koordinator alle Teilnehmer an festzuschreiben oder zurückzurollen, falls einer sich nicht vorbereiten konnte.

Auf dem Papier schließt das das Fenster. In der Praxis ist es für einen Broker und eine Service-Datenbank selten die Antwort, und zwar aus überwiegend betrieblichen Gründen.

- **Die Unterstützung ist uneinheitlich.** Viele verbreitete Broker implementieren überhaupt keine Schnittstelle eines Transaktionsmanagers. Wo einer sie hat, unterscheiden sich Garantien und Fehlerbehandlung von denen der Datenbank, und Sie erben beide Regelwerke.
- **Teilnehmer halten Sperren, während der Koordinator entscheidet.** Stürzt der Koordinator zwischen den beiden Phasen ab, bleiben vorbe-

reitete Transaktionen in der Schwebe. Zeilen bleiben gesperrt, bis jemand oder etwas sie auflöst. Die Schwebephase ist genau die Art seltener, langer Ausfall, die niemand übt.

- **Der Koordinator wird zu einer Komponente, die Sie betreiben, absichern und wiederherstellen müssen.** Er braucht eigene dauerhafte Logs und ein Wiederherstellungsverfahren, das stimmen muss.
- **Verfügbarkeiten multiplizieren sich.** Eine Transaktion, die Datenbank, Broker und Koordinator gleichzeitig gesund braucht, ist nur verfügbar, wenn alle drei es sind. Die Unabhängigkeit, die den Broker attraktiv machte, ist dahin.
- **Durchsatz und Latenz leiden.** Zwei Round Trips, erzwungene Schreibvorgänge bei jedem Teilnehmer und Sperren, die über Netzwerkaufrufe hinweg gehalten werden, summieren sich.
- **Es reicht nicht bis in die Außenwelt.** Ein E-Mail-Anbieter oder ein Zahlungsanbieter wird Ihrer Transaktion nicht beitreten. Selbst ein Team, das 2PC für den Broker einführt, braucht für jeden externen Aufruf eine andere Antwort.

Ich würde 2PC nicht in jedem Fall ausschließen. Ein Team, das beide Ressourcen kontrolliert, einen ausgereiften Transaktionsmanager betreibt und die betrieblichen Fähigkeiten hat, ungeklärte Transaktionen aufzulösen, kann es zum Laufen bringen. Für die meisten Teams, die aus Services einen Marktplatz bauen, jeder mit eigener Datenbank und einem Broker ohne XA-Schnittstelle, sind die Kosten hoch und der Nutzen schmal. Was sie wollen, ist kleiner als eine verteilte Transaktion: Sie wollen, dass die Zustandsänderung und die Absicht zu veröffentlichen gemeinsam festgeschrieben werden. Das ist ein lokales Problem, und lokale Transaktionen sind gut verstanden.

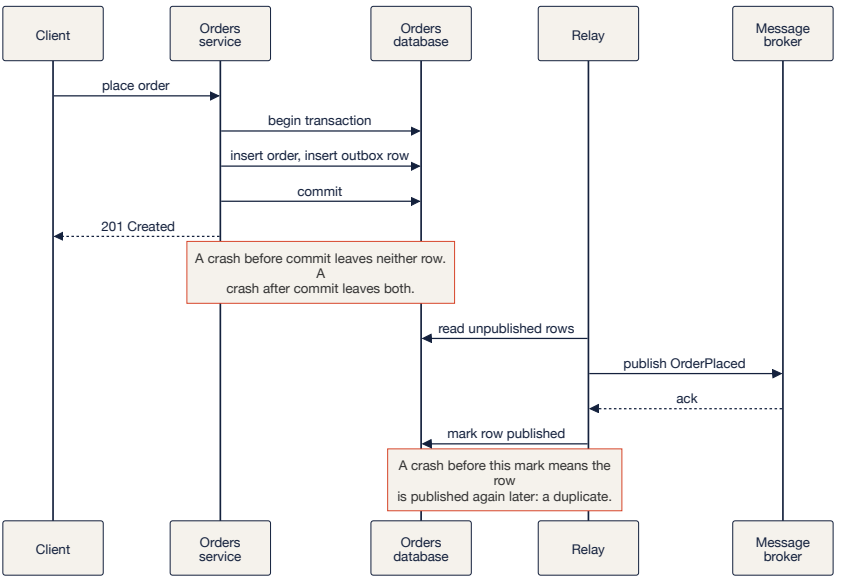
Die Transactional Outbox

Die Idee besteht darin, den Dual Write in einen einzigen Schreibvorgang plus eine spätere, wiederholbare Übertragung zu verwandeln. Statt innerhalb der Anfrage an den Broker zu veröffentlichen, schreibt der Service eine Zeile, die das Ereignis beschreibt, in eine Tabelle seiner eigenen Da-

tenbank, und zwar in derselben lokalen Transaktion wie die Zustandsänderung. Diese Tabelle ist die **Transactional Outbox**. Ab hier nennt das Buch sie die Outbox.

Ein separater Prozess, der **Relay**, liest Outbox-Zeilen und veröffentlicht sie an den Broker. Kapitel 5 handelt vom Relay. In diesem Kapitel behandeln wir ihn als eine Funktion, die jede Zeile schließlich mindestens einmal zustellt, und sehen uns an, was uns das bringt.

Die Überlegung ist kurz. Entweder werden die Bestellung und ihre Outbox-Zeile gemeinsam festgeschrieben, oder keines von beiden, weil sie in einer Datenbank und einer Transaktion liegen. Sobald die Zeile existiert, ist die Absicht zu veröffentlichen dauerhaft. Ein Absturz nach dem Commit verliert das Ereignis nicht mehr, denn die Zeile ist noch da, wenn der Service oder sein Ersatz wieder startet. Ein Absturz vor dem Commit hinterlässt weder Bestellung noch Zeile, also gibt es kein Phantomereignis.



Sehen Sie sich die beiden Notizen im Diagramm an. Die erste zeigt das Fenster, das wir geschlossen haben. Die zweite zeigt das Fenster, das wir behalten haben: Der Relay kann veröffentlichen und abstürzen, bevor er festhält, dass er das getan hat. Die Outbox beseitigt keine Duplikate, sie verlagert den Ausfall an eine Stelle, an der er begrenzt ist und behandelt wird. Darauf komme ich weiter unten zurück.

Die Tabelle

Die Outbox folgt der Form, die das übrige Buch verwendet. Sie trägt jedes Feld des Nachrichtenumschlags, sodass der Relay das Ereignis bauen kann, ohne den produzierenden Service etwas zu fragen. Dieses Kapitel fügt dem Umschlag vier Spalten hinzu, jede aus einem bestimmten Grund.

```
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
```

Einige dieser Entscheidungen brauchen eine Begründung.

- **`id` und `event_id` sind verschiedene Dinge.** Die `event_id` ist eine UUID, die die Consumer als Idempotenzschlüssel verwenden, und sie darf sich nicht ändern, wenn das Ereignis zweimal veröffentlicht wird. Die `id` ist eine von der Datenbank vergebene Sequenznummer, anhand derer der Relay Zeilen finden kann. Sie sagt nichts über die Commit-Reihenfolge aus, und der nächste Abschnitt zeigt, warum.
- **`aggregate_version` ist die maßgebliche Quelle der Reihenfolge.** Sie ist die Version des Aggregats, hochgezählt in derselben Transaktion, die die Zeile schreibt. Bei Parcelly-Bestellungen ist das die `order_version`. Consumer, die eine Reihenfolge pro Aggregat brauchen, vergleichen diese Zahl, nicht die `id` und nicht die Ankunftsreihenfolge. Kapitel 9 zeigt, wie ein Consumer sie verwendet. Der Standard-Nachrichtenumschlag hat dafür kein Feld; wenn der Relay das

Ereignis baut, kopiert er die Spalte deshalb als `order_version` in den Payload, und dort lesen die Consumer sie.

- **`aggregate_type` und `partition_key` sind getrennt.** Der `partition_key` ist das, was der Broker für die Reihenfolge verwendet, und er ist die Aggregat-ID, etwa `order-81724`. Der `aggregate_type` sagt dem Relay und jedem Menschen, der die Tabelle liest, auf welche Art von Ding sich der Schlüssel bezieht, und er eignet sich gut, um Zeilen auf verschiedene Topics zu verteilen.
- **`created_at` ist von `occurred_at` getrennt.** Das Erste ist der Zeitpunkt, an dem die Zeile geschrieben wurde, das Zweite der Zeitpunkt, an dem das Faktum eintrat. Meist liegen beide Millisekunden auseinander. Sie weichen voneinander ab, wenn ein Service ein Faktum festhält, das er spät erfahren hat, etwa einen Scan des Transporteurs, der nach dem Ereignis eintrifft. Eine Metrik für den Relay-Lag sollte `created_at` verwenden, weil der Lag eine Eigenschaft der Tabelle ist und nicht des Geschäfts. Lassen Sie die Datenbank die Spalte über einen Spaltenstandardwert füllen, statt im Anwendungscode `now()` aufzurufen, damit es nur eine Uhr gibt. Es bleibt der Zeitpunkt des Einfügens, nicht des Commits: Eine lange Transaktion lässt die Zeile älter aussehen, als der Relay sie sieht, und dieser Teil des Lags ist nicht die Schuld des Relays.
- **`headers` enthält Transportbelange.** Trace-Kontext und Routing-Hinweise gehören hierher, damit der Payload das fachliche Faktum bleibt. Ein Team, das nicht entscheiden kann, ob ein Feld Daten oder ein Transporthinweis ist, sollte Kapitel 11 lesen, bevor es das Feld in den Payload aufnimmt.
- **`published_at` ist nullable, und null hat eine Bedeutung.** Eine Zeile ohne `published_at` ist eine Zeile, die der Relay dem Broker noch schuldet. Manche Teams verwenden stattdessen eine Spalte `status` mit Werten wie `PENDING`, `PUBLISHED` und `FAILED`. Beide Entwürfe tragen dieselbe Information. Ich bevorzuge den Zeitstempel, weil er auch sagt, wann, was nützlich ist, wenn Sie fragen, wie spät der Relay dran war, und weil weniger schiefgehen kann. Wenn Sie eine Möglichkeit brauchen, Zeilen zu parken, die dauerhaft fehlschlagen, fügen Sie dann eine Spalte `failed_at` hinzu, nicht vorher.

Dieser Moment kommt früher, als es klingt. Eine Zeile, die der Broker nie annehmen wird, etwa ein zu großer Payload oder einer, den der Serializer ablehnt, blockiert ihren Schlüssel und im einfachsten Relay die ganze Tabelle, bis jemand sie repariert. Beim ersten Mal brauchen Sie eine Spalte `failed_at` (oder einen Status `FAILED`, falls Sie sich für Statusspalten entschieden haben) und einen Alarm auf geparkte Zeilen. Kapitel 5 zeigt, wie der Relay sie verwendet.

Ein Index auf den unveröffentlichten Zeilen in Einfügereihenfolge ist der einzige Index, den der Relay braucht. Ohne ihn durchsucht der Relay eine Tabelle, die ewig wächst. Kapitel 5 behandelt das Aufräumen.

Schreiben in derselben Transaktion

Der produzierende Code ist kurz, und die Kürze ist der Punkt. Die folgende Funktion ist die gesamte Änderung am Anfragepfad von 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
```

Drei Details sind wichtiger, als sie aussehen.

Erstens spricht nichts in der Funktion mit dem Broker. Ein Ausfall des Brokers lässt die Bestellaufgabe nicht fehlschlagen, und ein Broker, der eine Minute zum Antworten braucht, hält keine Datenbanktransaktion offen.

Das ist ein echter Gewinn an Verfügbarkeit: Orders kann Bestellungen annehmen, während der Broker ausgefallen ist, und die Outbox wächst, bis der Relay sie abarbeiten kann.

Zweitens wird die `event_id` vor dem Commit erzeugt und in der Zeile gespeichert. Jede spätere erneute Veröffentlichung dieses Ereignisses trägt dieselbe ID. Auf diese Eigenschaft verlassen sich die Consumer, und deshalb wäre es ein Fehler, die ID im Relay zum Zeitpunkt des Veröffentlichens zu erzeugen.

Drittens wird der Payload innerhalb der Transaktion gebaut, aus dem Zustand, den die Transaktion geschrieben hat. Würde der Payload später durch erneutes Lesen der Bestellung gebaut, könnte eine nebenläufige Änderung ihn verändern, und das Ereignis würde ein Faktum beschreiben, das zum damaligen Zeitpunkt nicht existierte. Ein Ereignis ist eine Momentaufnahme dessen, was geschah, und die Momentaufnahme wird an derselben Stelle gemacht wie die Änderung.

Der Pseudocode nimmt an, dass das Einfügen der Bestellung und das Einfügen in die Outbox dieselbe Datenbankverbindung und dieselbe Transaktion verwenden. Das wird häufiger versehentlich gebrochen, als man denkt. Eine Datenzugriffsschicht, die für die Outbox eine zweite Verbindung öffnet oder das Einfügen in die Outbox automatisch festschreibt, führt den Dual Write stillschweigend wieder ein, nun zwischen zwei Tabellen. Der Test dafür gehört in Ihre Suite: Erzwingen Sie einen Fehler nach dem Einfügen der Bestellung und prüfen Sie, dass keine Outbox-Zeile existiert.

Commit-Reihenfolge, Reihenfolge nach `id` und die Aggregatsperre

Eine neue Bestellung hat keinen Konkurrenten, also braucht `placeOrder` keine Sperre. Jede spätere Änderung an einer bestehenden Bestellung braucht eine, und die Sperre muss zuerst kommen. Hier ist `cancelOrder`, das `OrderCancelled` ausgibt.

```
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
```

Die Sperre macht die Version vertrauenswürdig, und sie ist auch das Einzige, was der Reihenfolge nach `id` für ein Aggregat eine Bedeutung gibt. Ohne sie nehmen wir zwei Transaktionen auf derselben Bestellung. Transaktion B fügt ihre Outbox-Zeile ein und erhält die `id` 101, dann blockiert sie auf der Bestellzeile, die Transaktion A hält. A fügt die `id` 102 ein und schreibt fest, und B schreibt danach fest. Die Commit-Reihenfolge ist A, dann B, und die Reihenfolge nach `id` ist B, dann A. Ein Relay, der beide Zeilen in einer Abfrage sieht, veröffentlicht sie vertauscht. Wenn die Sperre zuerst genommen wird, kann B nichts einfügen, bis A festgeschrieben hat, sodass die IDs eines Aggregats seiner Commit-Reihenfolge folgen.

Selbst dann würde ich Consumer nicht von der `id` abhängig machen. Zeilen verschiedener Aggregate verschränken sich, ein paralleler Relay ordnet über Batches hinweg um, und niemand außerhalb der Orders-Datenbank kann die `id` sehen. Die `aggregate_version` reist mit dem Ereignis, als `order_version` im Payload, und sagt, in welcher Reihenfolge die Fakten zu dieser Bestellung eingetreten sind. Kapitel 5 kommt darauf zurück, was ein Relay bewahren kann und was nicht, und Kapitel 9 zeigt die Seite des Consumers.

Was der Relay schuldet

Der Vertrag des Relays ist bewusst klein. Es wählt unveröffentlichte Zeilen aus, veröffentlicht jede an den Broker und markiert sie als veröffentlicht, sobald der Broker bestätigt. Die einfachste Version steht unten, und Kapi-

tel 5 behandelt die Betriebsfragen, die sie ignoriert, etwa Batching, Sperren, Reihenfolge und 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())
```

Der Kommentar in der Schleife ist die ganze Garantie. Die Zeile wird erst markiert, nachdem der Broker bestätigt hat, und deshalb wartet der Aufruf auf den Ack und deshalb stoppt ein Fehler den Batch. Ein Absturz zwischen dem Veröffentlichen und dem Markieren lässt die Zeile unveröffentlicht. Beim nächsten Durchlauf veröffentlicht der Relay sie erneut. Die Alternative, erst zu markieren und dann zu veröffentlichen, würde aus dem Ausfall ein verlorenes Ereignis statt eines Duplikats machen. Die Outbox wählt den sichereren Ausfall mit Absicht.

Was die Outbox garantiert

Die Outbox gibt Ihnen drei Dinge und nimmt eines weg.

- **Atomarität zwischen Zustand und Absicht.** Die Zustandsänderung und die Entscheidung zu veröffentlichen werden gemeinsam festgeschrieben. Es gibt keine festgeschriebene Bestellung ohne Outbox-Zeile und keine Outbox-Zeile ohne festgeschriebene Bestellung.
- **Dauerhaftigkeit der Absicht.** Sobald die Transaktion festgeschrieben ist, ist das Ereignis so dauerhaft wie die Datenbank. Der Relay kann ausfallen, der Broker kann ausfallen, der Producer kann ersetzt werden, und die Zeile wartet.
- **Veröffentlichung At-least-once.** Jede festgeschriebene Zeile wird schließlich veröffentlicht, vorausgesetzt, Relay und Broker erholen sich schließlich, der Relay findet Zeilen nach Zustand und nicht über

einen ID-Cursor, das Aufräumen löscht nur veröffentlichte Zeilen, und eine Zeile, die der Broker nie annehmen wird, wird geparkt und alarmiert, statt für immer zu blockieren (all das steht in Kapitel 5). Eine Zeile kann mehr als einmal veröffentlicht werden.

- **Nicht garantiert: Veröffentlichung Exactly-once.** Der Relay kann nicht atomar an den Broker veröffentlichen und eine Zeile in der Datenbank markieren, aus demselben Grund, aus dem es die Anwendung nicht konnte. Das Fenster ist vom Anfragepfad ins Relay gewandert, wo es kleiner, beobachtbar und reparierbar ist.

Bei dieser Zusammenfassung möchte ich sorgfältig sein. Die Outbox ist keine Behauptung, dass Ereignisse nie verloren gehen. Sie ist die Behauptung, dass das Verlustfenster des Dual Write auf der Seite des Producers geschlossen ist, sofern die Anwendung für jede Zustandsänderung, die ein Ereignis erzeugen muss, über die Outbox schreibt. Wer die Tabelle von Hand ändert oder über einen Pfad, der die Outbox umgeht, öffnet das Fenster wieder. Kapitel 3 hat das als menschlichen Fehler aufgeführt, und es ist der häufigste Weg, auf dem ein Team mit Outbox trotzdem Ereignisse verliert.

Das Duplikat wird zum Problem des Consumers

Weil die Veröffentlichung At-least-once erfolgt, muss jeder Consumer eines über die Outbox veröffentlichten Ereignisses Duplikate tolerieren. Das ist kein Makel, den man wegkonstruieren könnte, es ist der Preis der Garantie, und es lohnt sich, deutlich zu sagen, wer ihn zahlt.

Das Team des Producers übernimmt den leichteren Teil. Es fügt eine Tabelle hinzu, schreibt eine Zeile und betreibt einen Relay. Die Teams der Consumer übernehmen den schwereren Teil, denn sie müssen eine `event_id` wiedererkennen, die sie schon gesehen haben, und vermeiden, dass sich ihre Wirkung wiederholt. Payments darf nicht zweimal autorisieren. Notifications darf nicht dieselbe E-Mail zweimal senden. Ledger darf nicht denselben Eintrag zweimal buchen.

> **Warnung:** Wer eine Outbox ohne idempotente Consumer einführt, tauscht ein Problem verlorener Ereignisse gegen ein Problem doppelter Wirkungen. Bei manchen Ereignissen, etwa einer doppelten Belastung, ist das zweite Problem das teurere. Liefern Sie beides zusammen aus, und lesen Sie in Kapitel 7, wie ein Consumer wiederholte Arbeit erkennt.

Ich würde den Teams der Producer außerdem in klaren Worten sagen: Die Outbox ist ein Vertrag mit Ihren Consumern. Die `event_id` muss für jede erneute Veröffentlichung desselben Faktums stabil bleiben, und der Vertrag sollte in der Ereignisdokumentation stehen und nicht implizit bleiben.

Ein kurzer Durchgang durch die Ausfälle

Um den Entwurf an den Fenstern aus Kapitel 3 zu prüfen, gehen wir Orders noch einmal durch die Abstürze.

1. Der Prozess stirbt, bevor die Transaktion festgeschrieben wird. Keine Bestellung, keine Outbox-Zeile, kein Ereignis. Der Client sieht einen Fehler oder einen Timeout und wiederholt die Anfrage möglicherweise, was ein eigenes Problem der API-Schicht ist, das die Diskussion zur Idempotenz in Kapitel 7 behandelt.
2. Der Prozess stirbt direkt nach dem Commit. Bestellung und Zeile existieren beide. Der Relay veröffentlicht die Zeile beim nächsten Durchlauf. Der Client hat seine Antwort womöglich nicht erhalten, aber das System ist konsistent.
3. Der Relay stirbt nach dem Veröffentlichen und vor dem Markieren. Das Ereignis liegt im Broker, und die Zeile ist unveröffentlicht. Der nächste Durchlauf veröffentlicht es erneut. Consumer sehen ein Duplikat mit derselben `event_id`.
4. Der Broker ist nicht verfügbar. Der Relay kann nicht veröffentlichen, lässt die Zeilen unveröffentlicht und versucht es später erneut. Orders nimmt weiter Anfragen an. Der Lag wächst, und ein Alarm sollte es auch.
5. Der Service verliert seine Verbindung während des Commits und kennt das Ergebnis nicht. Entweder existieren Bestellung und Zeile, oder keines von beiden, und der Service kann nicht sagen, was zutrifft. Er darf nicht raten. Die sichere Antwort ist, den Retry des Clients einen Idem-

potenzschlüssel auf API-Ebene mitführen zu lassen, sodass der Retry, in welchem Zustand auch immer sich die Datenbank befindet, auf eine Bestellung und ein Ereignis konvergiert (Kapitel 7).

Jeder Fall endet entweder mit Konsistenz oder mit einem Duplikat, und keiner endet mit einem Phantomereignis oder einem stillen Verlust. Das ist alles, was die Outbox behauptet, und es reicht, um sie zu rechtfertigen.

Ein Duplikat sitzt stromaufwärts der Outbox, und keine nachgelagerte Abwehr sieht es. Wenn ein Client nach einer verlorenen Antwort einen Retry sendet und die API keinen eigenen Schlüssel hat, legt Orders eine zweite Bestellung und ein zweites `OrderPlaced` mit einer neuen `event_id` an. Die Inbox jedes Consumers sieht dann zwei verschiedene Ereignisse, weil es welche sind. Die Outbox hat ihre Aufgabe zweimal erfüllt. Die Lösung gehört an die API, mit dem Ansatz des `Idempotency-Key` aus Kapitel 7.

Alternativen und Nachbarn

Die Outbox ist nicht der einzige Weg, den Dual Write zu stoppen. Zwei weitere Entwürfe verdienen einen fairen Vergleich, denn ein Team könnte vernünftigerweise einen davon bevorzugen.

Auf sich selbst hören

Beim Pattern **Listen-to-yourself** schreibt der Service nicht zuerst in seine Datenbank. Er veröffentlicht das Ereignis an den Broker, konsumiert dann sein eigenes Ereignis und wendet die Zustandsänderung im Handler an. Der Broker wird zum einzigen Schreibvorgang, und die Datenbank wird zu einer Projektion dessen, was der Service gehört hat.

Das beseitigt den Dual Write, weil es nur einen Schreibvorgang gibt. Es verändert aber auch die Gestalt des Services. Die Anfrage des Clients kehrt zurück, bevor der Zustand existiert, sodass der Aufrufer seinen eigenen Schreibvorgang nicht lesen kann, wenn das System keine Möglichkeit zum Warten bietet. Die Validierung muss vor dem Veröffentlichen geschehen, denn ein Ereignis ist nach der Veröffentlichung ein Faktum, und ein ungültiges lässt sich nicht nachträglich ohne Kompensation ablehnen. Und der Handler, der das Ereignis anwendet, muss selbst idempotent sein, was dieselbe Last des Consumers zurückbringt.

Ich würde das für Services erwägen, die bereits in Ereignissen denken und in ihren eigenen Lesezugriffen Eventual Consistency vertragen. Für Orders, wo der Browser des Käufers erwartet, die Bestellung gleich nach dem Klick auf den Button zu sehen, würde ich es nicht tun. Die Lücke beim Lesen des eigenen Schreibvorgangs ist ein Produktproblem, das das Team am Ende mit Behelfslösungen löste.

Event Sourcing

Beim **Event Sourcing** sind die Ereignisse die maßgebliche Quelle. Der Service speichert die Folge der Ereignisse pro Aggregat, und der aktuelle Zustand wird daraus abgeleitet. Der Schreibvorgang in den Event-Store ist die Zustandsänderung, also gibt es nichts, was mit einem separaten Ereignis konsistent gehalten werden müsste. Das Veröffentlichen an den Broker liest dann aus dem Event-Store, was ein Relay unter anderem Namen ist.

Event Sourcing löst den Dual Write, indem es ändert, was die Datenbank ist. Es liefert Ihnen außerdem eine vollständige Historie und die Möglichkeit, Projektionen neu aufzubauen. Es kostet ein anderes Programmiermodell, die Schemaevolution gespeicherter Ereignisse, die Sie nie beiläufig umschreiben können, und eine Lernkurve für das ganze Team. Es ist eine erhebliche architektonische Festlegung, und ich würde sie nicht allein deshalb eingehen, um eine Outbox zu vermeiden. Wenn Sie es bereits einsetzen, brauchen Sie trotzdem einen Relay, und alles in diesem Kapitel über die Veröffentlichung At-least-once gilt weiter.

Die Optionen im Vergleich

Option	Was sie beseitigt	Was sie kostet	Passt am besten, wenn
Erst Commit, dann veröffentlichen	Nichts	Stiller Verlust bei einem Absturz	Ein verlorenes Ereignis akzeptabel ist, etwa bei Best-Effort-Metriken
2PC / XA	Das Fenster, in der Theorie	Sperren, Koordinator, uneinheitliche Unterstützung	Sie beide Ressourcen kontrollieren und die Wiederherstellung selbst betreiben
Transactional Outbox	Das Fenster auf der Seite des Producers	Eine zusätzliche Tabelle, ein Relay, Duplikate	Der Zustand in einer relationalen Datenbank liegt und Ereignisse nicht verloren gehen dürfen

Option	Was sie beseitigt	Was sie kostet	Passt am besten, wenn
Listen-to-yourself	Den zweiten Schreibvorgang	Eventual Consistency in den eigenen Lesezugriffen, frühe Validierung	Der Service ohnehin ereignisorientiert ist
Event Sourcing	Den separaten Zustandsschreibvorgang	Ein anderes Zustandsmodell	Die Historie selbst eine Anforderung ist

Für einen Service, der seinen Zustand bereits in einer relationalen Datenbank speichert und andere darüber informieren muss, ist die Outbox die kleinste Änderung. Deshalb geht der Großteil dieses Buches von ihr aus.

Abwägungen und Kosten

Die Outbox ist billig zu beschreiben und nicht kostenlos zu betreiben. Das sind die Kosten, die ich einem Team vorlegen würde, bevor es sich festlegt.

Mehr bewegliche Teile. Es gibt eine Tabelle, einen Relay-Prozess und einen Lag, den man überwachen muss. Wenn etwas kaputtgeht, gibt es einen neuen Ort, an dem man nachsehen muss. Der Relay ist eine Komponente, die jemand verantworten muss, und er braucht Deployments, Alarme und ein Runbook. Kapitel 5 liefert die Details.

Zusätzliche Last auf der Datenbank. Jedes Ereignis ist ein Einfügen in derselben Transaktion wie der fachliche Schreibvorgang, und jedes Veröffentlichen ist ein Lesen und ein Aktualisieren. Transaktionen werden etwas länger, und das Schreibvolumen steigt. Für die meisten Services ist der Anstieg im Vergleich zur übrigen Arbeit gering. Ein Service, der sehr hohe Ereignisvolumen schreibt, sollte das auf seiner eigenen Hardware messen, statt meinem „gering“ zu vertrauen.

Zusätzliche Latenz. Das Ereignis erreicht den Broker nach dem Commit der Transaktion und nachdem der Relay die Zeile bemerkt hat. Beim Polling hängt die Verzögerung vom Polling-Intervall ab. Bei einem Log-basierten Ansatz kann sie kürzer sein, und Kapitel 5 vergleicht beide. In jedem Fall ist die Outbox nicht die richtige Wahl für Abläufe, in denen das Ereignis den Broker innerhalb weniger Millisekunden nach der Anfrage erreichen muss, und Sie sollten Ihr Latenzbudget kennen, bevor Sie wählen.

Duplikate von Entwurf an. Siehe oben. Wenn ein Consumer nicht idempotent gemacht werden kann, verbessert die Outbox für ihn nichts, und das Team muss entscheiden, ob dieser Consumer in dieser Form existieren sollte.

Die Reihenfolge bleibt nur teilweise erhalten. Der Relay liest Zeilen nach `id`, und die Reihenfolge nach `id` ist nicht die Commit-Reihenfolge, es sei denn, jede Transaktion sperrt die Aggregatzeile, bevor sie die Outbox-Zeile einfügt. Diese Sperre serialisiert die Schreibenden eines Aggregats, was bei stark umkämpften Aggregaten eigene Kosten hat. Parallele Relays verschlechtern die Reihenfolge, und die Sperre hilft nicht über Aggregate hinweg. Die Spalte `aggregate_version` gibt den Consumern etwas zum Prüfen, und sie kostet bei jedem Schreibvorgang eine Versionserhöhung. Die Outbox gibt Ihnen einen Ort, um die Reihenfolge durchzusetzen, nicht die Reihenfolge geschenkt. Kapitel 5 und Kapitel 9 kommen darauf zurück.

Wachstum und Aufräumen. Veröffentlichte Zeilen sammeln sich an. Ohne Löschen oder Archivieren wächst die Tabelle, bis die Abfrage des Relays langsamer wird. Die Aufbewahrung ist eine Entscheidung, und sie wirkt auf den Abgleich, denn alte Zeilen sind auch Ihr Nachweis dessen, was veröffentlicht wurde.

Funktioniert nur für Zustand in einer Datenbank, die Transaktionen unterstützt. Wenn der Zustand in einem Store ohne Transaktionen über mehrere Zeilen liegt, brauchen Sie einen anderen Ansatz, etwa einen Store-spezifischen Änderungsstrom oder einen Event-Sourcing-Entwurf. Ein Store, der Atomarität nur innerhalb eines Dokuments bietet, hat eine eigene Variante: die ausstehenden Ereignisse im Aggregatdokument halten und den Relay sie von dort abarbeiten lassen.

Wann man sie nicht einsetzt

Ich würde keine Outbox hinzufügen, wenn:

- das Ereignis Best-Effort ist und sein Verlust akzeptabel ist, etwa eine Metrik oder ein Analytics-Hinweis von geringem Wert
- der Service selbst keinen dauerhaften Zustand hält, sodass nichts zusammen mit dem Ereignis festzuschreiben ist

- das System klein genug ist, dass ein einziger Prozess sowohl den Zustand als auch seine Consumer besitzt und ein direkter Aufruf einfacher ist
- das Team Consumer noch nicht idempotent machen kann; dann würde ich zuerst das beheben, denn die Outbox erhöht den Einsatz bei dieser Lücke

> **Hinweis:** „Wir verwenden eine Outbox“ ist eine Aussage über den Producer-Pfad eines einzelnen Services. Sie sagt nichts darüber, ob die Consumer sicher sind, ob die Reihenfolge hält oder ob die Ereignisse gut entworfen sind. Betrachten Sie sie als ein Pattern in einer Menge, nicht als Gütesiegel für Zuverlässigkeit.

Häufige Fehler

- **Die Outbox-Zeile in einer separaten Transaktion schreiben.** Es ist verlockend, weil ein Framework-Hook oder eine Event-Bus-Abstraktion die Verbindung verbirgt. Die beiden Schreibvorgänge werden dann zu einem Dual Write zwischen zwei Tabellen derselben Datenbank, und das Fenster ist zurück. Verwenden Sie eine Verbindung und eine Transaktion, und testen Sie den Rollback.
- **Aus einem „After-Commit“-Hook veröffentlichen.** Es ist verlockend, weil er nur für festgeschriebene Daten feuert und keine Tabelle braucht. Es ist Commit-dann-Veröffentlichen mit der Absicht im Arbeitsspeicher, sodass ein Absturz dazwischen das Ereignis verliert. Schreiben Sie die Zeile in der Transaktion und lassen Sie den Relay veröffentlichen.
- **Die Outbox-Zeile einfügen, bevor das Aggregat gesperrt ist.** Der Code funktioniert in jedem Test mit einem einzigen Schreibenden. Unter Nebenläufigkeit laufen die Reihenfolge nach `id` und die Commit-Reihenfolge für eine Bestellung auseinander, und der Relay kann ihre Ereignisse vertauscht veröffentlichen. Sperren Sie zuerst die Aggregatzeile, erhöhen Sie ihre Version und fügen Sie dann ein.
- **Aus dem Anfragepfad veröffentlichen und die Outbox als Backup behalten.** Es fühlt sich an wie Gürtel und Hosenträger. Es erzeugt zwei Publisher, Duplikate mit unterschiedlichem Timing und eine Rei-

henfolge, über die Sie nicht nachdenken können. Lassen Sie den Relay den einzigen Publisher sein.

- **Die `event_id` im Relay erzeugen.** Es sieht harmlos aus, denn der Relay baut die Nachricht. Eine erneute Veröffentlichung bekommt dann eine neue ID, und Consumer können sie nicht als Duplikat erkennen. Erzeugen Sie die ID, wenn die Zeile geschrieben wird.
- **Die Zeile als veröffentlicht markieren, bevor der Broker bestätigt.** Es verringert Duplikate und wirkt aufgeräumter. Es verwandelt ein Duplikat in ein verlorenes Ereignis, den schlimmeren Ausfall. Markieren Sie nach der Bestätigung und nehmen Sie das Duplikat in Kauf.
- **Den Payload nach dem Commit bauen.** Das erneute Lesen des Aggregats, um das Ereignis zu füllen, scheint den produzierenden Code einfach zu halten. Eine andere Transaktion kann die Daten dazwischen ändern, und das Ereignis passt nicht mehr zum Faktum. Bauen Sie den Payload aus dem, was die Transaktion geschrieben hat.
- **Manche Codepfade die Outbox umgehen lassen.** Ein Migrationsskript, ein Admin-Tool oder eine von Hand ausgeführte SQL-Korrektur ändert den Zustand und gibt nichts aus. Leiten Sie jede relevante Zustandsänderung durch dieselbe Funktion, und prüfen Sie manuelle Reparaturen auf die Ereignisse, die sie hätten erzeugen sollen.
- **Das Aufräumen vergessen.** Die Tabelle funktioniert monatelang, und dann wird die Abfrage des Relays zum schlechtesten Zeitpunkt langsamer. Legen Sie die Aufbewahrung fest, wenn Sie die Tabelle anlegen.
- **Die Outbox für das Ende der Zuverlässigkeitsgeschichte halten.** Sie schließt ein Fenster. Consumer brauchen weiterhin Idempotenz, Retries und einen Ort für Nachrichten, die nie gelingen.

Zentrale Erkenntnisse

- Ein Dual Write lässt sich nicht durch eine bessere Reihenfolge sicher machen: Erst Commit, dann veröffentlichen verliert Ereignisse, erst veröffentlichen, dann Commit erfindet sie, und ein Veröffentlichen innerhalb eines Transaktionsblocks ist immer noch Letzteres.
- Der Zwei-Phasen-Commit kann das Fenster im Prinzip schließen, doch für eine Service-Datenbank und einen Broker kostet er meist mehr an

Sperren, Koordinatorbetrieb und Verfügbarkeit, als das Problem wert ist.

- Die Transactional Outbox verwandelt den Dual Write in eine lokale Transaktion plus eine wiederholbare Übertragung, indem sie das Ereignis als Zeile in derselben Transaktion wie die Zustandsänderung schreibt.
- Die Outbox-Zeile trägt den vollständigen Nachrichtenumschlag des Ereignisses, eine stabile `event_id`, die beim Schreiben erzeugt wird, damit jede erneute Veröffentlichung erkennbar ist, und eine `aggregate_version`, die unter der Sperre des Aggregats hochgezählt wird, weil die Reihenfolge nach `id` nicht die Commit-Reihenfolge ist.
- Die Outbox garantiert Atomarität von Zustand und Absicht, dauerhafte Absicht und Veröffentlichung At-least-once, und sie garantiert nicht, dass eine Nachricht genau einmal veröffentlicht wird.
- Duplikate sind der Preis der Garantie, daher tauscht eine Outbox ohne idempotente Consumer einen Ausfall gegen einen anderen.
- Listen-to-yourself und Event Sourcing beseitigen den Dual Write ebenfalls, ändern aber, wie der Service seinen Zustand liest und speichert, deshalb würde ich sie wegen dessen wählen, was sie bieten, und nicht bloß, um einen Relay zu vermeiden.

Das Veröffentlichen ist nur die halbe Übertragung: Etwas muss diese Zeilen lesen und zum Broker tragen, ohne sie zu verlieren, umzuordnen oder ihn zu überfluten. Kapitel 5 behandelt den Relay, Polling im Vergleich zu Change Data Capture und den Betrieb der Outbox in der Produktion.

Kapitel 5 — Relays, Polling und Change Data Capture

Stellen Sie sich einen Mittwochmorgen bei Parcelly vor. Das Orders-Team hat die Transactional Outbox vor zwei Wochen ausgeliefert. Jedes Ereignis `OrderPlaced` wird nun in derselben lokalen Transaktion wie die Bestellung selbst in die Tabelle `outbox` geschrieben, und das Dual-Write-Fenster aus Kapitel 4 ist geschlossen. Das Team hält das Problem für gelöst und wendet sich dem nächsten Ticket zu.

Der Relay ist ein kleiner Prozess, der jede Sekunde aufwacht, Zeilen mit einer `id` liest, die größer ist als die zuletzt gesehene, sie veröffentlicht und die neue höchste `id` speichert. Er läuft als einzelne Instanz, und seit dem Tag des Deployments hat niemand mehr an ihn gedacht.

Dann ersetzt ein Routine-Deployment den Relay im falschen Moment. Die alte Instanz hat einen Batch von 200 Zeilen an den Broker veröffentlicht, aber ihre neue Position noch nicht gespeichert. Die neue Instanz startet an der zuletzt gespeicherten Position, liest dieselben 200 Zeilen und veröffentlicht sie noch einmal. Inventory reserviert Bestand ein zweites Mal für Bestellungen, die es bereits bearbeitet hat. Eine Woche später kommt ein Aufräumjob hinzu und löscht unbemerkt Zeilen, die der Relay noch nicht gelesen hat. Im Monat darauf wird eine lang laufende Transaktion verspätet festgeschrieben, und ihre Zeilen verlassen die Tabelle nie.

Keiner dieser Fehler liegt im Outbox-Pattern selbst. Alle liegen in dem Teil, der Zeilen zum Broker bewegt. Wie betreibt man diesen Teil so, dass er langsam ausfällt, schnell wieder anläuft und ehrlich bleibt, was die Duplikate angeht, die er noch erzeugen kann?

Das Problem und warum es schwierig ist

Die Transactional Outbox gibt Ihnen eine Garantie: Wenn die Bestellung festgeschrieben wurde, wurde die Absicht, ihr Ereignis zu veröffentlichen, mit ihr festgeschrieben. Über die zweite Hälfte der Reise sagt sie nichts. Jemand muss die Zeilen lesen, sie dem Broker übergeben und festhalten, dass er es getan hat. Dieser Jemand ist der **Relay**, der Prozess, der Outbox-Zeilen zum Broker bewegt.

Ein Relay tut drei Dinge: Er findet Zeilen, die veröffentlicht werden müssen, er veröffentlicht sie, und er hält fest, dass er das getan hat. Das sind drei getrennte Schritte, und Broker und Datenbank teilen keine Transaktion. Das Dual-Write-Problem, das wir aus dem Producer entfernt haben, taucht im Relay wieder auf, mit einem engeren Fenster.

Die Absturzfenster sind wenige genug, um sie aufzulisten. Die Tabelle ist die Ausfallsemantik des Relays: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall überlebt.

Absturz geschieht	Zustand der Zeile	Ergebnis
Vor dem Veröffentlichen	Unveröffentlicht	Die Zeile wird erneut aufgegriffen, das Ereignis einmal veröffentlicht. Sicher.
Nach dem Veröffentlichen, vor dem Markieren	Unveröffentlicht, aber das Ereignis liegt im Broker	Die Zeile wird erneut aufgegriffen, das Ereignis zweimal veröffentlicht. Ein Duplikat.
Nach dem Markieren	Veröffentlicht	Nichts zu tun. Sicher.

Keine Anordnung der drei Schritte schließt die mittlere Zeile. Markiert der Relay zuerst und veröffentlicht danach, verliert ein Absturz dazwischen das Ereignis, also genau den Ausfall, den die Outbox verhindern soll. Der Relay veröffentlicht daher zuerst und markiert danach und nimmt in Kauf, dass ein Absturz dazwischen ein Duplikat erzeugt. Das ist derselbe Kompromiss, den das Outbox-Kapitel eingegangen ist: Zustellung At-least-once, wobei die Behandlung der Duplikate den Consumern überlassen bleibt (Kapitel 7).

Die erste Entwurfsfrage lautet deshalb nicht, wie man Duplikate vermeidet. Sie lautet, wie man das Fenster klein hält, wie man sicherstellt, dass die Zeilen in sinnvoller Reihenfolge herauskommen, und wie man sieht,

dass der Relay zurückfällt, bevor es ein Kunde tut. Die zweite Entwurfsfrage ist die Wahl zwischen zwei Mechanismen zum Lesen der Zeilen: die Tabelle per Polling abfragen oder das Transaktionslog der Datenbank selbst lesen. Jeder verlagert die Schwierigkeit an eine andere Stelle.

Der Polling-Relay

Ein **Polling-Relay** fragt die Outbox-Tabelle in einem Intervall ab, veröffentlicht, was er findet, und markiert diese Zeilen als veröffentlicht. Er ist der einfachste Mechanismus, der funktioniert, und ich würde hier beginnen, es sei denn, die Zahlen sprechen dagegen.

Die Grundschleife

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

        // defined below: only broker-confirmed ids
        ackedIds = publishAcked(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())
```

Die Abfrage hinter `selectUnpublished` ist die Stelle, an der die meisten Polling-Relays falsch abbiegen, deshalb lohnt sich ein genauerer Blick. Die naheliegende Form lautet „gib mir Zeilen mit einer `id` größer als die letzte, die ich bearbeitet habe“. Sie sieht effizient aus und ist unsicher.

Warum ein Cursor auf der id nicht sicher ist

Zeilen-IDs werden meist vergeben, wenn das Einfügen läuft, nicht wenn die Transaktion festgeschrieben wird. Stellen Sie sich zwei Orders-Transaktionen vor. Transaktion A fügt eine Outbox-Zeile mit der id 101 ein und verbringt dann einige Zeit in einer langsamen Funktion, bevor sie festgeschrieben wird. Transaktion B fügt die id 102 ein und wird sofort festge-

schrieben. Der Relay fragt ab, sieht 102, veröffentlicht sie und setzt seinen Cursor auf 102. Dann wird A festgeschrieben. Zeile 101 liegt nun hinter dem Cursor und wird nie gelesen.

Der Fehler ist leise. Nichts wirft einen Fehler, keine Logzeile erscheint, und das Ereignis wird nie veröffentlicht. Es ist derselbe Verlust, den die Outbox beseitigen sollte, durch eine Optimierung wieder eingeführt.

Die Lösung, die ich bevorzuge, besteht darin, sich keine Position mehr zu merken und stattdessen nach Zustand abzufragen: Zeilen auswählen, bei denen `published_at` null ist, geordnet nach `id`. Eine Zeile, die verspätet festgeschrieben wird, ist bei der nächsten Abfrage noch unveröffentlicht und wird daher gefunden. Der Preis ist ein Index auf unveröffentlichte Zeilen und eine Tabelle, die klein genug bleiben muss, damit diese Abfrage günstig ist. Ein partieller Index, der nur unveröffentlichte Zeilen abdeckt, hält die Abfrage schnell, unabhängig davon, wie viel Historie die Tabelle enthält, sofern Ihre Datenbank so etwas unterstützt. Falls nicht, wiegt die Aufbewahrung weiter unten in diesem Kapitel schwerer. Ein partieller Index beseitigt nicht alle Kosten: In Datenbanken, die alte Zeilenversionen aufbewahren, wird jede Zeile eingefügt, aktualisiert und später gelöscht, und dieses ständige Einfügen, Aktualisieren und Löschen hinterlässt tote Zeilen, die die Engine aufräumen muss.

Das zustandsbasierte Lesen behebt den Verlust. Die Reihenfolge behebt es nicht, das ist eine eigene Frage, die der Abschnitt zum Bewahren der Reihenfolge beantwortet.

Batching und das Polling-Intervall

Zwei Einstellungen bestimmen das Verhalten des Relays: wie viele Zeilen er pro Abfrage nimmt und wie lange er schläft, wenn nichts zu tun ist.

Ein größerer Batch verteilt die Kosten der Abfrage und des Markierens auf mehr Zeilen und verbessert den Durchsatz. Er vergrößert aber auch das Absturzfenster: Ein Relay, der mitten in einem Batch von 500 Zeilen stirbt, hinterlässt bis zu 500 Ereignisse zur erneuten Veröffentlichung. Ein kleinerer Batch veröffentlicht nach einem Absturz weniger erneut und hält, mit dem unten beschriebenen Beanspruchen von Zeilen, Sperren kürzer. Sagen wir, 50 bis 200 Zeilen sind eine vernünftige erste Schätzung; messen Sie, bevor Sie justieren.

Das Polling-Intervall ist eine Untergrenze für die Latenz. Bei einem Intervall von einer Sekunde kann ein Ereignis fast eine Sekunde warten, bevor der Relay überhaupt hinsieht. Ein kürzeres Intervall senkt die Latenz und erhöht die Zahl leerer Abfragen, die die Datenbank bedienen muss. Für Parcelly ist eine Verzögerung von ein, zwei Sekunden zwischen dem Commit von `OrderPlaced` und seiner Veröffentlichung für Käufer unsichtbar. Bei einem Ablauf, in dem eine Person vor einem Bildschirm wartet, kann das anders sein.

Markieren nach dem Veröffentlichen

Der Markierungsschritt verdient eigene Sorgfalt. Veröffentlichen Sie jede Zeile, sammeln Sie die Bestätigung des Brokers ein und markieren Sie nur die Zeilen, die der Broker bestätigt hat. Markieren Sie keine Zeile, deren Veröffentlichung nicht bestätigt wurde, auch wenn der Aufruf ohne Fehler zurückkehrte, denn viele Client-Bibliotheken senden asynchron und melden einen Fehler erst später.

```
function publishAcked(rows):
    ackedIds = []
    for each row in rows:
        result = publishAndAwaitAck(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
```

Diese Funktion berührt die Tabelle nie. Sie gibt die IDs zurück, die der Broker bestätigt hat, und jeder Relay in diesem Kapitel markiert genau diese IDs und keine anderen. Eine Zeile, die fehlschlug, und jede Zeile danach im Batch bleibt unveröffentlicht und wird bei der nächsten Abfrage erneut gelesen. Veröffentlichen und Markieren werden mit Absicht getrennt gehalten, damit das Markieren an der einen Stelle, an der es geschieht, durch eine Sperre oder ein Lease geschützt werden kann. Jedes Ereignis geht an den Topic seines Aggregattyps, mit `partition_key` als Nachrichtenschlüssel, dasselbe Routing, das Kapitel 4 verwendet.

Das `break` ist eine bewusste Entscheidung, und es geht dabei um die Reihenfolge, auf die wir weiter unten zurückkommen. Eine fehlgeschlagene Zeile zu überspringen und die Zeilen danach zu veröffentlichen ist schneller und kann die Ereignisse eines Aggregats umordnen. Anzuhalten ist langsamer und sicher. Ich würde anhalten und mir vom Monitoring sagen lassen, dass der Relay feststeckt. Diese Skizze hält der Einfachheit halber den ganzen Batch an; nur den betroffenen Schlüssel zu blockieren ist die Verfeinerung, die unten unter „Die Reihenfolge bewahren“ beschrieben wird.

Ein aktiver Relay oder mehrere

Die einfachste Topologie ist eine Relay-Instanz, die die ganze Arbeit erledigt. Ihr Verhalten bei der Reihenfolge lässt sich leichter durchdenken, weil sie in der Reihenfolge nach `id` liest und in dieser Reihenfolge veröffentlicht, mit dem Vorbehalt zu `id` und Commit-Reihenfolge im Abschnitt „Die Reihenfolge bewahren“ weiter unten. Ihre Schwäche ist die Verfügbarkeit: Stirbt diese Instanz, wird nichts veröffentlicht, bis eine andere startet. Für viele Systeme ist ein Neustart innerhalb weniger Sekunden akzeptabel, und ein einzelner Relay mit einem Supervisor, der ihn neu startet, ist die richtige Antwort.

Wenn ein Relay nicht mehr mitkommt oder eine Lücke beim Veröffentlichen nicht hinnehmbar ist, betreiben Sie mehrere. Dann haben Sie ein neues Problem: Zwei Relays dürfen nicht dieselbe Zeile gleichzeitig veröffentlichen. Das Standardwerkzeug ist eine Zeilensperre mit Skip-Semantik. Viele relationale Datenbanken bieten eine Möglichkeit, Zeilen zum Aktualisieren auszuwählen und dabei Zeilen zu überspringen, die eine andere Transaktion bereits gesperrt hat, gemeinhin als Sperren im Stil von `SKIP LOCKED` beschrieben. Der Relay beansprucht einen Batch innerhalb einer Transaktion, und die anderen Relays übergehen diese Zeilen.

```
function claimAndPublish():
    transaction:
        rows = selectUnpublished(limit: 100, lockRows: true,
skipLocked: true)
        ackedIds = publishAcked(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
```

Das funktioniert und hat Kosten, die oft übersehen werden. Erstens bleibt die Transaktion offen, während der Relay mit dem Broker spricht, sodass ein langsamer Broker lange gehaltene Sperren und einen wachsenden Stapel offener Transaktionen bedeutet. In Datenbanken, die alte Zeilenversionen für nebenläufige Leser aufbewahren, hält eine lange offene Transaktion außerdem das Aufräumen toter Zeilen zurück, was genau die Tabelle trifft, die am stärksten umschlägt. Zweitens wird jede Zeile, die sie veröffentlicht hat, erneut veröffentlicht, wenn die Transaktion nach dem Veröffentlichen abbricht, etwa durch eine getrennte Verbindung oder ein fehlgeschlagenes Markieren. Eine kürzere Alternative ist eine kurze Claim-Transaktion, die die Zeilen mit einem Claim-Zeitpunkt versieht, ein Veröffentlichen außerhalb jeder Transaktion und eine separate Mark-Transaktion, mit einem Claim-Timeout, damit die Zeilen eines toten Relays wieder beanspruchbar werden. Das kostet einen zusätzlichen Zustand, über den man nachdenken muss. Drittens verlieren parallele Relays die globale Reihenfolge. Relay A beansprucht womöglich die Zeilen 1 bis 100, während Relay B die Zeilen 101 bis 200 beansprucht, und B wird möglicherweise zuerst fertig. Bei Ereignissen zu verschiedenen Bestellungen spielt das keine Rolle. Bei zwei Ereignissen zur selben Bestellung schon.

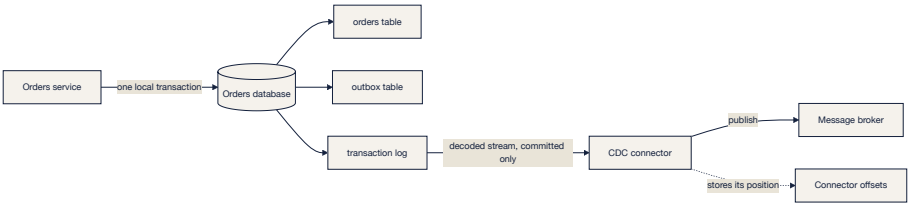
Die Abhilfe besteht darin, die Arbeit nach `partition_key` aufzuteilen statt nach Zeile. Weisen Sie jedem Relay eine Menge von Schlüsseln zu, zum Beispiel indem Sie `partition_key` per Hash auf eine feste Anzahl Buckets abbilden und jeden Bucket jeweils genau einem Relay geben. Innerhalb eines Buckets werden Zeilen von einem Worker der Reihe nach bearbeitet, über Buckets hinweg laufen sie parallel. Das ist dieselbe Idee wie Partitionen bei einem logbasierten Broker, nur einen Schritt früher angewandt. Das Wort „genau“ braucht das Lease aus dem Abschnitt zum Failover weiter unten, und die Übergabe eines Buckets bei einem Rebalance ist der Moment, in dem die Reihenfolge pro Schlüssel am stärksten gefährdet ist,

weil der alte und der neue Besitzer womöglich gleichzeitig für denselben Schlüssel veröffentlichen. Kapitel 9 erklärt, warum die Reihenfolge pro Schlüssel diejenige ist, die sich realistisch halten lässt.

Change Data Capture

Der zweite Mechanismus fragt die Outbox-Tabelle überhaupt nicht ab. **Change Data Capture (CDC)** liest das Transaktionslog der Datenbank selbst und macht aus festgeschriebenen Zeilenänderungen einen Strom von Änderungsdatensätzen. Ein CDC-Connector, als bekanntes Beispiel Debezium, hängt sich an dieses Log, bemerkt, dass eine Zeile in `outbox` eingefügt wurde, und veröffentlicht eine Nachricht an den Broker.

Die Tabelle behält ihre Rolle, denn die transaktionale Garantie entsteht weiterhin dadurch, dass sie in derselben Transaktion wie die Zustandsänderung geschrieben wird. Was sich ändert, ist, wer sie liest. Kein Anwendungsprozess führt Polling durch. Die Datenbank hält ohnehin jede festgeschriebene Änderung für ihre eigene Wiederherstellung und Replikation in einem geordneten Log fest, und der Connector liest dieses Log so, wie es ein Replikat täte.



Sehen Sie sich den Weg vom Transaktionslog zum Broker an. Der Änderungsstrom, den die Datenbank für Connectors bereitstellt (PostgreSQL nennt ihn Logical Decoding, MySQL stellt sein Binärlog bereit), enthält nur festgeschriebene Transaktionen, und sie erscheinen in Commit-Reihenfolge. Das beseitigt das Problem verspäteter Commits aus dem Polling-Abschnitt vollständig: Das Log ist bereits eine totale Ordnung der Commits, also erscheint eine Zeile, die verspätet festgeschrieben wird, an der richtigen Stelle verspätet.

Wie es sich verhält

Aus dem Entwurf ergeben sich einige Eigenschaften.

Die Reihenfolge folgt der Commit-Reihenfolge. Ereignisse erscheinen in der Reihenfolge, in der die Transaktionen festgeschrieben wurden. Ein Connector, der mit dem `partition_key` der Zeile als Nachrichtenschlüssel veröffentlicht, behält bei einem logbasierten Broker die Reihenfolge pro Schlüssel, vorausgesetzt, der Producer darin ist so konfiguriert, dass ein Retry Anfragen, die gemeinsam in Flug sind, nicht umordnen kann. Prüfen Sie die Einstellungen Ihres Clients; derselbe Vorbehalt gilt für einen Polling-Relay, der asynchron veröffentlicht.

Die Latenz ist niedrig und gleichmäßig. Es gibt kein Polling-Intervall. Der Connector sieht eine Änderung kurz nach dem Commit, die Verzögerung ist also meist gering und hängt nicht davon ab, wie oft irgendetwas aufwacht.

Der Connector verfolgt seine eigene Position. Er hält fest, wie weit er das Log gelesen hat, und setzt nach einem Neustart dort fort. Connectors speichern die Position üblicherweise periodisch und nicht nach jedem Ereignis, sodass ein Absturz jedes Ereignis seit der zuletzt gespeicherten Position erneut veröffentlichen kann. CDC ist At-least-once, genau wie Polling. Das Duplikatfenster hat sich verschoben, es ist nicht verschwunden.

Das Flag „veröffentlicht“ ist optional. Da der Connector keine unveröffentlichten Zeilen finden muss, braucht die Tabelle um des Relays willen keine Spalte `published_at`. Viele Teams, die CDC einsetzen, löschen jede Outbox-Zeile kurz nach dem Einfügen. Das Einfügen ist bereits im Log festgehalten, also sieht der Connector sie weiterhin. Die Tabelle bleibt winzig, und das Aufräumproblem verschwindet weitgehend.

Der Umschlag muss geformt werden. Ein roher Änderungsdatensatz beschreibt eine Zeile, kein Ereignis. Der Connector oder ein Transformationsschritt muss die Outbox-Spalten auf den Standardumschlag abbilden: `event_id`, `event_type`, `partition_key` als Nachrichtenschlüssel, `payload` als Rumpf. Die meisten Connectors unterstützen dafür Routing und Feldzuordnung. Halten Sie die Outbox-Spalten am Umschlag ausgerichtet, damit die Abbildung mechanisch bleibt.

Was es kostet

CDC ist nicht kostenlos, und die Kosten sind betrieblicher Natur.

Sie betreiben nun einen Connector, meist innerhalb eines Connector-Frameworks, samt der Broker-Infrastruktur, von der er abhängt. Jemand muss lernen, wie er ausfällt.

Die Datenbank muss ihr Log dem Connector zugänglich machen, was in der Regel bedeutet, eine bestimmte Logging-Stufe zu aktivieren und eine Replikationsposition anzulegen, die die Datenbank im Auftrag des Connectors aufbewahrt. Bleibt der Connector stehen und niemand bemerkt es, behält die Datenbank Log-Segmente, die sie nicht verwerfen kann, und die Platte läuft voll. Das ist eine bekannte Ausfallform in der CDC-Familie, und die Einzelheiten unterscheiden sich je nach Datenbank. Ich würde „der Connector ist ausgefallen“ als Alarmierung behandeln, denn der Schaden beschränkt sich nicht auf den Broker.

Schemaänderungen an der Outbox-Tabelle und an jeder Tabelle, die der Connector beobachtet, müssen geplant werden. Das Umbenennen einer Spalte oder eine Typänderung kann die Abbildung brechen.

Auch die Ersteinrichtung, Snapshots und Neustarts des Connectors nach langen Ausfällen verdienen eine Probe. Das Verhalten hängt vom Produkt und von der Datenbank ab, und ich würde es nicht aus einem Diagramm annehmen.

> **Hinweis:** CDC ist eine Familie von Techniken, und die Produkte unterscheiden sich darin, wie sie das Log lesen, wie sie Positionen speichern und was nach einem langen Ausfall geschieht. Betrachten Sie diesen Abschnitt als die allgemeine Form, und lesen Sie die Dokumentation Ihres Connectors und Ihrer Datenbank für das genaue Verhalten der Versionen, die Sie einsetzen.

Wenn das Log weg ist

„Die Platte läuft voll“ ist die eine Hälfte eines Zielkonflikts, und die andere Hälfte ist schlimmer. Die Datenbank behält entweder Log für die Position des Connectors ohne Grenze, was die Platte füllt, oder sie begrenzt, was sie behält, oder lässt Logs nach einem Zeitplan verfallen, was die Platte schützt. Viele Datenbanken lassen Ihnen die Wahl. Wenn Sie begrenzen

und der Connector länger ausfällt als die Grenze, zeigt seine gespeicherte Position auf Log, das nicht mehr existiert. Ereignisse in der Lücke gehen verloren, oder der Connector muss von einem Snapshot aus neu beginnen.

Was ein erneuter Snapshot mit einer Outbox macht, hängt von einer früheren Entscheidung ab. Hält die Tabelle noch Zeilen, gibt ein Snapshot alle erneut aus, sodass jede aufbewahrte Zeile für jeden Consumer zum Duplikat wird. Wurden die Zeilen, wie bei CDC üblich, direkt nach dem Einfügen gelöscht, findet der Snapshot nichts, und die Ereignisse in der Lücke lassen sich aus der Tabelle überhaupt nicht wiederherstellen. Das ist ein stiller Verlust: kein Fehler, kein Duplikat, eine Bestellung, die Inventory nie erreicht hat. Das Löschen, das die Tabelle winzig hielt, beseitigt auch die Beweise.

Ein Failover des Datenbank-Primärknotens fügt einen dritten Fall hinzu. Ob die Position des Connectors auf den neuen Primärknoten übergeht, hängt von Datenbank und Version ab, und ich würde das durch eine Probe des Failovers herausfinden und nicht durch das Lesen eines Diagramms.

Daraus folgen drei Gewohnheiten. Wählen Sie die Grenze der Aufbewahrung bewusst, als Abwägung von Platte gegen die Länge eines Ausfalls, den Sie überstehen können, und schreiben Sie die Zahl auf. Alarmieren Sie auf den Spielraum, also darauf, wie viel vom Aufbewahrungsfenster der Lag des Connectors schon verbraucht hat, bevor es aufgebraucht ist. Und entscheiden Sie vor dem Ausfall, wie Sie Ereignisse aus der Lücke wiederherstellen würden, zum Beispiel mit einem Abgleich-Job (Kapitel 13), der den Zustand der Bestellungen mit dem vergleicht, was veröffentlicht wurde. Kapitel 14 behandelt die Wiederherstellung, wenn der Schaden schon eingetreten ist.

Zwischen Polling und CDC wählen

Keiner der beiden Mechanismen ist im Abstrakten besser. Die Tabelle zeigt, wo sie sich unterscheiden.

Aspekt	Polling-Relay	CDC-Connector
Bewegliche Teile	Ein kleiner Prozess in Ihrer Hand	Connector, sein Framework, Log-Einstellungen der Datenbank

Aspekt	Polling-Relay	CDC-Connector
Latenz	Nach unten durch das Polling-Intervall begrenzt	Meist niedrig und gleichmäßig
Quelle der Reihenfolge	Abfragereihenfolge plus Ihre eigene Sorgfalt	Commit-Reihenfolge aus dem Log
Verspätet festgeschriebene Transaktionen	Sicher bei zustandsbasierter Abfrage	Vom Log erledigt
Last auf der Datenbank	Wiederholte Abfragen, Zeilensperren	Log-Lesezugriffe, aufbewahrte Log-Segmente
Duplikate nach Absturz	Möglich, das Fenster ist der unmarkierte Batch	Möglich, das Fenster ist die ungespeicherte Position
Wachstum der Tabelle	Braucht Aufbewahrungsarbeit	Zeilen können sofort gelöscht werden
Ausfall, der am meisten schmerzt	Stiller Lag, festsitzende Zeilen	Stehender Connector, aufbewahrtes Log
Nötige Fähigkeiten	Gewöhnliche Anwendungsentwicklung	Interna des Datenbank-Logs, Connector-Betrieb

Mein Standard für ein Team am Anfang ist ein Polling-Relay. Er besteht aus ein paar Dutzend Zeilen Code, das Team weiß bereits, wie man einen solchen Service betreibt und debuggt, und seine Ausfallarten sind in der Tabelle sichtbar, die er abfragt. Ich würde zu CDC wechseln, wenn eines von drei Dingen zutrifft: Die Polling-Latenz ist zu hoch und lässt sich nicht senken, ohne die Datenbank zu hämmern, das Outbox-Volumen macht die Polling-Abfragen zu einem messbaren Kostenfaktor, oder Sie betreiben bereits eine CDC-Plattform und jemand ist dafür verantwortlich.

Ich würde CDC nicht einführen, nur weil es ernsthafter klingt. Ein Team, das nie ein Replikationslog betrieben hat und seinen Bestellfluss darauf legt, hat ein Problem, das es verstand, gegen eines getauscht, das es nicht versteht. Umgekehrt gilt auch: Ein Team, das CDC schon für andere Zwecke betreibt, gewinnt wenig, wenn es nebenher einen Polling-Relay schreibt.

Den Relay betreiben

Den Mechanismus zu wählen ist die erste Hälfte der Arbeit. Die zweite Hälfte ist die Arbeit, die ihn gesund hält.

Aufbewahrung und Aufräumen

Ein Polling-Relay hinterlässt veröffentlichte Zeilen. Sich selbst überlassen, wächst die Tabelle ohne Grenze, die Abfrage nach unveröffentlichten Zeilen wird langsamer, und die Datenbank verwendet Mühe auf Zeilen, die niemand liest. Das Aufräumen muss entworfen und nicht später nachgerüstet werden.

Zwei Entscheidungen zählen: wie lange veröffentlichte Zeilen aufbewahrt werden und wie sie gelöscht werden. Sie eine Weile zu behalten hat Wert. Meldet ein Consumer ein fehlendes Ereignis, ist die Outbox die erste Stelle, an der Sie nachsehen, ob es je veröffentlicht wurde. Sagen wir, ein paar Tage sind ein vernünftiger Ausgangspunkt. Wie lange richtig ist, hängt davon ab, wie schnell Ihr Team fehlende Ereignisse bemerkt, und das ist eine Frage zu Menschen und Alarmierung, nicht zum Speicherplatz.

```
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)
```

Löschen Sie in kleinen Batches, damit der Job keine Sperren lange hält und die Replikate nicht überflutet. Ist die Tabelle groß, ist es günstiger, sie nach Zeit zu partitionieren und ganze Partitionen zu verwerfen, als Zeile für Zeile zu löschen, sofern Ihre Datenbank das unterstützt. Was auch immer Sie wählen, die Bedingung des Aufräumens muss `published_at is not null` enthalten. Ein Aufräumen, das allein nach Alter löscht, löscht an dem Tag, an dem der Relay zurückfällt, Ereignisse, die nie veröffentlicht wurden. Das ist der Fehler aus dem Eröffnungsszenario, und es ist ein Datenverlust, kein Duplikat.

> **Warnung:** Löschen Sie Outbox-Zeilen nie allein nach Alter. Steckt ein Relay länger fest als Ihre Aufbewahrungsdauer, werden seine unveröffentlichten Zeilen gelöscht, und diese Ereignisse sind für immer weg. Löschen Sie nur Zeilen, die als veröffentlicht markiert sind, und alarmieren Sie auf unveröffentlichte Zeilen, die alt werden.

Bei CDC ist das Aufräumen anders, aber nicht abwesend. Zeilen können direkt nach dem Einfügen gelöscht werden, die Tabelle bleibt also klein. Was jetzt wächst, ist das aufbewahrte Log, und was man beobachten muss, ist der Lag des Connectors dagegen. Sofortiges Löschen bedeutet auch, dass ein erneuter Snapshot nichts wiederherstellen kann, wie der frühere Abschnitt über ein verschwundenes Log erklärt, wägen Sie also das sofortige Löschen gegen eine kurze Aufbewahrung veröffentlichter Zeilen ab.

Den Relay überwachen

Ein Relay, der stehen bleibt, ist von außen unsichtbar. Orders gelingen weiter, die Outbox nimmt Zeilen an, und jeder andere Service kommt stillschweigend ohne Ereignisse aus. Niemand bekommt einen Fehler. Deshalb braucht der Relay eigene Signale, und ich würde drei davon vor dem Start einrichten.

1. **Alter der ältesten unveröffentlichten Zeile.** Das ist die beste Einzelzahl, weil sie misst, was Nutzer erleben: wie lange das langsamste Ereignis schon wartet. Alarmieren Sie, wenn sie einen Schwellenwert überschreitet, der am Geschäft hängt und nicht an der Technik. Soll die Bestätigungs-E-Mail für einen Käufer binnen einer Minute eintreffen, ist ein Alter der ältesten unveröffentlichten Zeile von fünf Minuten ein Incident.
2. **Anzahl unveröffentlichter Zeilen.** Ein wachsendes Backlog zeigt, dass der Relay langsamer ist als der Producer, auch wenn die älteste Zeile noch jung ist.
3. **Fehlerrate beim Veröffentlichen und Dauer der Batches.** Sie erklären die ersten beiden. Ein Relay, der bei jeder Abfrage dieselbe Zeile wiederholt, zeigt eine stabile Fehlerzahl und ein wachsendes Alter.

Bei einem CDC-Connector gelten dieselben Fragen in anderen Einheiten: wie weit die Position des Connectors hinter dem Head des Logs liegt, ob er läuft und wie viel Log die Datenbank für ihn aufbewahrt. Kapitel 13 kommt auf diese Signale zurück und auf den Abgleich, der erfasst, was Lag-Metriken übersehen.

Failover und doppeltes Veröffentlichen

Ein Relay muss seine eigenen Neustarts überstehen, und dazu gehört der Fall, dass zwei Instanzen kurz gleichzeitig laufen. Deployments, Netzwerkpartitionen, die die alte Instanz tot aussehen lassen, und Orchestratoren, die einen Prozess neu starten, bevor der alte beendet ist, erzeugen alle diese Überlappung.

Verwenden Sie für einen einzelnen aktiven Relay ein Lease: eine Zeile oder eine Sperre in der Datenbank, die der aktive Relay in kurzem Intervall erneuert und die verfällt, wenn der Halter aufhört. Ein Standby übernimmt nach dem Verfall. Das Detail, über das man stolpert: Der alte Halter weiß womöglich nicht, dass sein Lease abgelaufen ist. Er könnte durch eine lange Garbage Collection oder ein hängendes Netzwerk pausiert worden sein, aufwachen und einen Batch veröffentlichen, der ihm nicht mehr gehört.

```
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 = publishAckedWithin(rows, token)
        // marking rows the broker confirmed is true even if the lease
lapsed meanwhile
        markPublished(ackedIds, now())

function publishAckedWithin(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
```

Das ansteigende Token ist ein Fencing-Token, und die Idee funktioniert nur dort, wo die geschützte Ressource es prüft. Die Datenbank kann es prüfen: Ein Commit, der ein veraltetes Token trägt, lässt sich zum Scheitern bringen, und Kapitel 7 nutzt genau das für Consumer und APIs. Der Broker kann es in der Regel nicht, deshalb hindert das Token einen veralteten Relay nicht am Veröffentlichen. Das Markieren ist eine andere Sache. Der veraltete Halter markiert nur Zeilen, die der Broker bestätigt hat, seine Markierungen sind also wahr, und sie zurückzurollen würde nur weitere Duplikate erzeugen. Was ein Lease begrenzen muss, ist das gleichzeitige Veröffentlichen, denn zwei Halter, die zusammen veröffentlichen, ergeben Duplikate und können eine ältere Zeile eines Schlüssels hinter eine neuere setzen.

Vier Dinge begrenzen es. Halten Sie die TTL des Leases deutlich über der Worst-Case-Dauer eines Batches und halten Sie Batches klein, denn das Lease wird einmal pro Schleife erneuert, und ein langsamer Broker allein kann es überdauern. Prüfen Sie das verbleibende Lease vor jedem Veröf-

fentlichen, wie es die Skizze tut. Wo ein logbasierter Broker Producer-Fencing per Epoche unterstützt, nutzen Sie es, damit der Broker selbst den alten Halter abweist. Und wenn der alte Halter aufwacht, lassen Sie ihn es herausfinden, indem er die Datenbank fragt, in einer Transaktion, die die Lease-Zeile sperrt, bevor er irgendetwas weiter veröffentlicht. Eine Prüfung, die die Sperre nicht hält, ist ein Check-then-act und scheitert auf dieselbe Weise.

Das ist die ehrliche Grenze. Ein Failover kann ein doppeltes Veröffentlichen erzeugen, und kein Lease-Schema beseitigt es. Ein Lease erkauft Ihnen, dass das Fenster kurz und selten ist. Den Schaden beseitigt ein Consumer, der die zweite Kopie als dasselbe Ereignis behandelt, anhand der `event_id`, die über beide Veröffentlichungen hinweg stabil ist. Der Relay muss die `event_id` deshalb aus der Outbox-Zeile übernehmen und beim Veröffentlichen nie eine neue erzeugen. Ein Relay, der bei jedem Senden einen frischen Bezeichner prägt, macht aus einem harmlosen Duplikat zwei scheinbar verschiedene Ereignisse. Manche Broker akzeptieren eine vom Producer mitgegebene Deduplizierungs-ID und verwerfen Wiederholungen innerhalb eines Zeitfensters; wo Ihrer das tut, setzen Sie sie auf `event_id`. Das verengt das Fenster kostenlos und ersetzt nicht die Inbox des Consumers. Kapitel 7 behandelt die konsumierende Seite.

Die Reihenfolge bewahren

Die Outbox bewahrt die Reihenfolge der Ereignisse nur so weit, wie der Relay es tut, und der Relay kann nur die Reihenfolge bewahren, die die Tabelle enthält. Kapitel 4 hat gezeigt, dass die Reihenfolge nach `id` nicht die Commit-Reihenfolge ist: Eine Transaktion kann eine `id` nehmen, auf eine Sperre warten und nach einer Transaktion festgeschrieben werden, die eine spätere `id` genommen hat. Die `id` ist die Reihenfolge des Einfügens, nicht die des Commits.

Für einen Polling-Relay folgen daraus zwei Dinge. Wird eine niedrigere `id` für einen Schlüssel nach einer höheren für denselben Schlüssel, die bereits veröffentlicht wurde, festgeschrieben, gehen die beiden vertauscht hinaus, und die zustandsbasierte Abfrage, die die verspätete Zeile gerettet hat, veröffentlicht sie verspätet. Werden beide in einer Abfrage mit vertauschten IDs sichtbar, gehen sie in einem Batch vertauscht hinaus. Ein Relay, der

„in der Reihenfolge nach `id`“ veröffentlicht, liefert die Ereignisse eines Aggregats deshalb nicht in der Reihenfolge, in der sich das Aggregat geändert hat, es sei denn, eine von zwei Bedingungen gilt.

Die erste ist, dass Producer die Aggregatzeile sperren, bevor sie die Outbox-Zeile einfügen, sodass Transaktionen zum selben Schlüssel serialisiert werden und ihre IDs ihrer Commit-Reihenfolge folgen (Kapitel 4). Die zweite ist, dass die Reihenfolge in den Daten mitgeführt wird: die Spalte `aggregate_version`, die Consumer vergleichen und die Kapitel 9 verwendet. Ich würde beides tun. Die Sperre ist eine billige Versicherung für den Relay, und die Version ist das, was alles nach dem Relay übersteht. Ein CDC-Connector umgeht das erste Problem, weil das Log in Commit-Reihenfolge vorliegt, und hilft trotzdem nicht über Partitionen hinweg, die Version bleibt also nützlich.

Dennoch zählen die eigenen Entscheidungen des Relays. Jede der folgenden kann die Reihenfolge brechen, die er hat:

- Eine fehlgeschlagene Zeile überspringen und spätere Zeilen für denselben Schlüssel veröffentlichen.
- Parallele Relays ohne Partitionierung nach Schlüssel betreiben.
- Ein fehlgeschlagenes Veröffentlichen außerhalb des Ablaufs wiederholen, sodass eine spätere Zeile zuerst ankommt.
- Unterschiedliche Nachrichtenschlüssel für Ereignisse eines Aggregats verwenden.

Der sichere Standard ist Head-of-Line-Blocking pro Schlüssel: Kann eine Zeile für `order-81724` nicht veröffentlicht werden, hören Sie auf, spätere Zeilen für `order-81724` zu veröffentlichen, und lassen Zeilen für andere Schlüssel weiterlaufen. Das kostet Latenz für das eine blockierte Aggregat und schützt seine Abfolge. Handelt es sich um eine Zeile, die nie veröffentlicht werden kann, etwa einen Payload, den der Serializer ablehnt, muss jemand sie reparieren, und das Monitoring von oben ist der Weg, auf dem er davon erfährt.

Machen Sie sich klar, was die frühere Skizze tut. Mit einem `break` auf den ganzen Batch und einer Abfrage, die die älteste Zeile zuerst liefert, wird eine unveröffentlichbare Zeile bei jeder Abfrage als Erste zurückgegeben und hält jeden Schlüssel für immer an. Das Blockieren pro Schlüssel braucht zwei Ergänzungen, hier als Verfeinerung von `publishAcked` ge-

zeigt, die dessen `break` ersetzt: Der Relay merkt sich, welche Schlüssel in diesem Batch fehlgeschlagen sind, und eine Zeile, die wiederholt fehlschlägt, wird geparkt, sodass die Abfrage sie nicht mehr zurückgibt und auch keine späteren Zeilen ihres Schlüssels mehr.

```
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
```

Der Select schließt dann jeden Schlüssel aus, der eine geparkte Zeile hat, sodass spätere Zeilen dieses Schlüssels warten und Zeilen anderer Schlüssel fließen. Das Parken ist der Moment, in dem die Spalte `failed_at` aus Kapitel 4 gebraucht wird, zusammen mit einem Zähler `attempts`, damit ein vorübergehender Fehler einige Male wiederholt wird, bevor die Zeile geparkt wird. Eine geparkte Zeile ist eine, deren `failed_at` gesetzt ist. Alarmieren Sie auf jede geparkte Zeile, denn eine geparkte Zeile ist ein blockiertes Aggregat, und ein Mensch muss sie reparieren oder verwerfen. Die Kosten sind zwei Spalten, eine Abfrage, die sich schwerer indizieren lässt, und ein Reparaturverfahren, das jemand vor dem ersten Incident schreiben muss.

Die Reihenfolge geht auch nach dem Relay verloren, im Consumer und im Broker. Ein Relay, der seinen Teil tut, gibt Ihnen eine Abfolge, die zu bewahren sich lohnt, keine Garantie, dass sie den Rest des Weges übersteht. Kapitel 9 verfolgt die Reihenfolge vom Broker bis zum Handler.

Abwägungen und Kosten

Beide Mechanismen tauschen Einfachheit gegen betriebliche Reichweite, und beide hinterlassen ein Duplikatfenster, das sich nicht schließen lässt.

Ein Polling-Relay kostet Datenbanklast und Latenz und bleibt dafür in Code, den Ihr Team bereits versteht. Seine Schwachstellen sind die Abfrage nach unveröffentlichten Zeilen auf einer wachsenden Tabelle, das Sperrverhalten, wenn mehrere Relays laufen, und stiller Lag, wenn niemand die älteste Zeile beobachtet. Veröffentlicht Ihr System mit bescheidener Rate und ist eine Sekunde Verzögerung akzeptabel, ist der Polling-Relay bei den Betriebskosten schwer zu schlagen.

CDC kostet eine neue Komponente und neue Fähigkeiten und bietet dafür niedrigere Latenz, Lesen in Commit-Reihenfolge und eine Tabelle, die klein bleibt. Seine Schwachstellen sind ein stehender Connector, der die Datenbank zwingt, Log aufzubewahren, oder ein begrenztes Log, das unter einem stehenden Connector verfällt und Ereignisse verliert, Schemaänderungen, die Abbildungen brechen, und Wiederherstellungsverfahren, die selten geprobt werden. Wenn niemand den Connector verantworten wird, würde ich ihn nicht einführen, wie verlockend das Architekturdiagramm auch aussieht.

Wann man überhaupt kein Outbox-Relay einsetzt: Kann ein Ereignis ohne Folgen verloren gehen, etwa bei einer rein informativen Benachrichtigung, bei der eine verpasste Nachricht nichts kostet, kann ein direktes Veröffentlichen nach dem Commit akzeptabel und billiger sein. Müssen Ihre Ereignisse Zustand in einer Datenbank abbilden, die keine lokale Transaktion über Zustand und Outbox bietet, kann der Relay das nicht reparieren, und Sie brauchen einen anderen Entwurf.

Zwei Kosten sind beiden gemeinsam. Keiner gibt Ihnen Exactly-once-Veröffentlichung: Das Duplikat nach einem Absturz zwischen Veröffentlichen und Markieren oder zwischen Veröffentlichen und Speichern der Position gehört zum Entwurf. Und keiner hilft, wenn Relay und Producer sich über den Umschlag uneinig sind, halten Sie also die Outbox-Spalten und die Felder des Umschlags im Gleichschritt.

Häufige Fehler

- **Einen Cursor auf der id statt einer zustandsbasierten Abfrage verwenden.** Es ist verlockend, weil es das Durchsuchen nach unveröffentlichten Zeilen erspart. Verspätet festgeschriebene Transaktionen fallen dann hinter den Cursor und werden nie veröffentlicht. Fragen Sie nach `published_at is null` ab oder verwenden Sie einen logbasierten Mechanismus, der bereits nach Commit ordnet.
- **Zeilen markieren, bevor der Broker bestätigt.** Zuerst zu markieren sieht aus, als würde es Duplikate beseitigen. Es verwandelt einen Absturz in ein verlorenes Ereignis. Veröffentlichen, die Bestätigung abwarten, dann markieren.
- **Im Relay einen neuen Ereignisbezeichner erzeugen.** Es fühlt sich harmlos an, da der Relay ja der Sender ist. Die Inbox des Consumers kann ein erneut veröffentlichtes Ereignis dann nicht mehr als dasselbe erkennen. Kopieren Sie die `event_id` aus der Zeile.
- **Allein nach Alter löschen.** Es hält die Tabelle klein und besteht jeden Test, in dem der Relay gesund ist. Es löscht unveröffentlichte Ereignisse, wenn der Relay es nicht ist. Löschen Sie nur veröffentlichte Zeilen und alarmieren Sie auf alte unveröffentlichte.
- **Eine fehlgeschlagene Zeile überspringen, um den Durchsatz hochzuhalten.** Spätere Zeilen für dasselbe Aggregat überholen sie dann. Blockieren Sie stattdessen pro Schlüssel, parken Sie die Zeile, die immer wieder fehlschlägt, und beheben Sie die Ursache.
- **Glauben, die Reihenfolge nach `id` sei die Commit-Reihenfolge.** In jedem Test mit einem einzigen Schreibenden stimmt es. Bei nebenläufigen Schreibenden kann eine niedrigere `id` später festgeschrieben werden, und der Relay veröffentlicht das Paar vertauscht. Sperren Sie die Aggregatzeile vor dem Einfügen in die Outbox und führen Sie `aggregate_version` mit.
- **Darauf vertrauen, dass das Lease den Broker abschirmt.** Ein veralteter Relay, der nach Ablauf seines Leases aufwacht, veröffentlicht trotzdem, weil der Broker das Token nicht prüft. Halten Sie Batches innerhalb der TTL, prüfen Sie das Lease vor jedem Veröffentlichen und lassen Sie die Consumer den Rest auffangen.
- **CDC mit einer Aufbewahrungsgrenze betreiben, die niemand gewählt hat.** Der Standardwert fühlt sich wie ein Detail an, bis der Con-

nector ein Wochenende lang ausgefallen und die Position weg ist. Wählen Sie die Grenze, alarmieren Sie auf den Spielraum und wissen Sie, was ein erneuter Snapshot mit Ihrer Outbox macht.

- **Mehrere Relays ohne Partitionierung nach Schlüssel betreiben.** Zeilensperren verhindern doppelte Arbeit und sagen nichts über die Reihenfolge. Weisen Sie Schlüssel-Buckets einzelnen Workern zu.
- **Den Relay nicht überwachen.** Die Outbox macht den Producer zuverlässig, wodurch der Ausfall des Relays der einzige verbleibende ist, und ein Relay, der stehen geblieben ist, sieht genau aus wie ein ruhiges System. Alarmieren Sie auf das Alter der ältesten unveröffentlichten Zeile.
- **Den Ausfall eines Connectors für ein Broker-Problem halten.** Bei CDC bringt ein stehender Connector die Datenbank dazu, ihr Log aufzubewahren. Lösen Sie dafür eine Alarmierung aus und beobachten Sie die Platte, nicht nur den Lag.

Zentrale Erkenntnisse

- Ein Relay tut drei getrennte Dinge, finden, veröffentlichen und markieren, und Broker und Datenbank teilen keine Transaktion, sodass ein Absturz zwischen Veröffentlichen und Markieren immer ein mögliches Duplikat hinterlässt.
- Erst veröffentlichen, dann markieren: Die andere Reihenfolge macht aus einem Absturz ein verlorenes Ereignis, also genau den Ausfall, den die Outbox beseitigen soll.
- Ein Polling-Relay sollte unveröffentlichte Zeilen nach Zustand auswählen und nicht über einen ansteigenden id-Cursor, weil Transaktionen in anderer Reihenfolge festgeschrieben werden, als sie IDs vergeben, und selbst dann gilt die Reihenfolge pro Aggregat nur, wenn Producer das Aggregat zuerst sperren und eine `aggregate_version` mitführen.
- Um mehrere Polling-Relays zu betreiben, beanspruchen Sie Zeilen mit Skip-Locked-artigem Sperren und partitionieren Sie nach `partition_key`, damit Zeilen eines Aggregats immer der Reihe nach von einem Worker bearbeitet werden.
- Change Data Capture liest festgeschriebene Änderungen in Commit-Reihenfolge aus dem Log der Datenbank und beseitigt Polling und das

Problem verspäteter Commits, zum Preis eines Connectors, einer Abwägung der Log-Aufbewahrung zwischen voller Platte und verlorenen Ereignissen und Fähigkeiten, die Ihr Team verantworten muss.

- Beginnen Sie mit einem Polling-Relay, es sei denn, Latenz, Last oder eine bestehende CDC-Plattform sprechen dagegen, und behandeln Sie die Wahl als Abwägung von Einfachheit gegen betriebliche Reichweite.
- Überwachen Sie das Alter der ältesten unveröffentlichten Zeile, löschen Sie nur veröffentlichte Zeilen, parken Sie Zeilen, die nie veröffentlicht werden, übernehmen Sie die `event_id` aus der Outbox-Zeile und erwarten Sie Duplikate nach einem Failover, denn ein Lease begrenzt das gleichzeitige Veröffentlichen, und nur ein idempotenter Consumer macht den Rest harmlos.

Der Relay liefert nun jedes Ereignis mindestens einmal aus, in einer Reihenfolge, die zu bewahren sich lohnt, und manchmal zweimal. Das nächste Problem liegt auf der anderen Seite des Brokers: was ein Consumer tun soll, wenn seine Arbeit fehlschlägt, womit sich Kapitel 6 anhand von Retries befasst.

TEIL III

Zuverlässig konsumieren

Kapitel 6 — Retries richtig gemacht

Stellen Sie sich einen Dienstagmorgen bei Parcelly vor, zu Beginn einer Verkäuferaktion. Orders sendet ein Command `CapturePayment`, und Payments reicht es an den externen Zahlungsanbieter weiter. Der Anbieter hat Mühe und antwortet langsam. Die Aufrufe von Payments laufen in Timeouts, also lässt Payments die Nachricht fehlschlagen, und der Broker stellt sie erneut zu. Jede Instanz von Payments tut dasselbe, im selben Moment, mit derselben festen Wartezeit von einer Sekunde.

Innerhalb einer Minute erhält der Anbieter nicht mehr die normale Last der Aktion, sondern die normale Last plus jeden fehlgeschlagenen Aufruf, der immer wieder zurückkommt, in synchronisierten Wellen. Die Langsamkeit des Anbieters, die nur ein paar Sekunden hätte dauern können, hält nun so lange an, wie die Retries eintreffen. Währenddessen sind die Consumer von Payments damit beschäftigt, Nachrichten zu wiederholen, die nicht gelingen werden, sodass gesunde Nachrichten sich dahinter stauen. Irgendwo in diesem Haufen werden einige Einzüge, die beim Anbieter tatsächlich erfolgreich waren, erneut gesendet.

Niemand hat hier eine schlechte Codezeile geschrieben. Jede Instanz tat das Vernünftige: Sie schlug fehl, sie wartete, sie versuchte es erneut. Der Schaden entstand aus tausend vernünftigen Entscheidungen, die niemand aufeinander abgestimmt hatte.

In diesem Kapitel geht es darum, aus einem Retry eine Entscheidung statt eines Reflexes zu machen. Die Frage, die es beantwortet, lautet: Wie entscheiden Sie, wenn ein Aufruf fehlschlägt, ob Sie es erneut versuchen, wann, wie oft und auf welcher Ebene, sodass der Retry dem System bei der Erholung hilft, statt sie zu verhindern?

Das Problem und warum es schwierig ist

Ein Retry ist das Versprechen, es erneut zu versuchen, nicht das Versprechen, dass es gelingt. In diesem Kapitel ist ein **Versuch** jeder Anlauf, sodass die Zahl der Versuche eins plus die Zahl der Retries ergibt. Der naheliegende Ansatz, „den Aufruf in eine Schleife packen und ein paar Mal wiederholen“, scheitert aus vier getrennten Gründen, und sie verstärken sich gegenseitig.

Erstens **lohnt sich nicht jeder Ausfall für einen Retry**. Eine Anfrage, die abgelehnt wurde, weil die Karte abgewiesen wurde, wird auch beim zehnten Versuch abgelehnt. Ein Retry verschwendet Kapazität und verzögert den Moment, in dem jemand das eigentliche Problem bemerkt. Die Retry-Schleife muss wissen, welche Art von Ausfall sie vor sich hat, und die meisten Schleifen wissen es nicht.

Zweitens **erzeugen Retries zusätzliche Last im schlechtesten Moment**. Eine Abhängigkeit fällt aus, wenn sie überlastet, beeinträchtigt oder im Neustart ist. Retries sind zusätzliche Anfragen, die genau auf diese Abhängigkeit zielen. Wenn jeder Aufrufer dreimal wiederholt, was insgesamt vier Versuche ergibt, sieht ein angeschlagener Service bis zu viermal so viel Traffic wie normal, gerade wenn er es sich am wenigsten leisten kann.

Drittens **synchronisieren sich Retries**. Aufrufer, die gemeinsam fehlschlagen, wiederholen gemeinsam. Eine feste Wartezeit macht aus einem einzelnen Ausfall eine periodische Spitze, weil jeder Aufrufer im selben Augenblick aufwacht. Das ist der Thundering Herd, und er ist der Grund, warum die Wartezeit Zufall braucht und nicht nur Länge.

Viertens **wiederholt ein Retry eine Aktion, deren erstes Ergebnis unbekannt sein kann**. Kapitel 3 hat gezeigt, dass ein Timeout nicht sagt, ob die Anfrage verarbeitet wurde. Ein Retry nach einem Timeout riskiert daher, die Arbeit doppelt auszuführen. Ohne Idempotenz (Kapitel 7) ist ein Retry ein Duplikat, das nur darauf wartet zu entstehen.

Alles andere in diesem Kapitel ist eine Antwort auf eines dieser vier Probleme. Die Klassifizierung beantwortet das erste. Backoff mit Jitter und ein Retry-Budget beantworten das zweite und das dritte. Idempotenz, die Kapitel 7 aufbaut, beantwortet das vierte, und der Abschnitt zu ihrem Fehlen weiter unten erklärt, warum man sie nicht überspringen kann.

Ausfälle klassifizieren

Bevor Sie entscheiden, wie Sie wiederholen, entscheiden Sie, ob Sie es überhaupt tun. Kapitel 1 hat drei Fragen gegeben, die man an jede Nahtstelle stellen sollte: was ausfallen kann, wie Sie es beobachten würden und welche Garantie ihn übersteht. Die Antworten zusammen sind die Semantik eines Ausfalls, und eine Retry-Entscheidung ist diese Semantik in der Anwendung. Ich finde es nützlich, jeden Ausfall in eine von drei Klassen einzuordnen, danach, was er darüber aussagt, ob ein erneuter Versuch überhaupt helfen kann, und dann ein viertes Signal hinzuzufügen, das über die Nachricht gar nichts aussagt.

Transiente Ausfälle

Ein **transienter Ausfall** ist einer, dessen Ursache voraussichtlich von selbst verschwindet. Ein Verbindungsabbruch, ein Timeout, ein `503 Service Unavailable`, ein Deadlock-Opfer in der Datenbank, eine Throttling-Antwort, eine Leader Election im Broker. Die Anfrage war nicht falsch. Die Welt stand kurz im Weg.

Ein Retry ist die richtige Reaktion, mit Wartezeit, denn die Wartezeit lässt die Ursache abklingen.

Permanente Ausfälle

Ein **permanenter Ausfall** ist einer, der bei jedem Versuch wieder auftritt, weil die Ursache in der Anfrage oder in den Daten liegt. Ein Payload, der die Validierung nicht besteht, ein Verweis auf einen Datensatz, der lange nach dem erwarteten Eintreffen noch fehlt (ein fehlender Datensatz kann in einem ereignisgesteuerten System transient sein, weil das Ereignis, das ihn erzeugt, verspätet eintreffen kann; Kapitel 9 behandelt diesen Fall), ein `400 Bad Request`, eine Karte, die wegen unzureichender Deckung abgewiesen wurde, ein Ereignis, dessen Schemaversion der Consumer nicht versteht.

Ein Retry ist vergebliche Mühe und manchmal schädlich. Die richtige Reaktion ist, aufzuhören, den Grund festzuhalten und die Nachricht dorthin zu schicken, wo ein Mensch oder ein Reparaturjob sich ihrer annehmen kann. Kapitel 8 behandelt, wohin permanente Ausfälle gehen. Beachten

Sie besonders die Kartenabweisung: Für Payments ist sie gar kein Fehler. Sie ist ein erwartetes geschäftliches Ergebnis, das zu einem Ereignis `PaymentFailed` wird, und es ist kein Retry beteiligt.

Unbekannte Ausfälle

Ein **unbekannter Ausfall** ist einer, dessen Klasse Sie nicht bestimmen können. Ein Timeout ist der klassische Fall, ebenso eine unerwartete Exception mit einer Meldung, die niemand vorhergesehen hat. Der Timeout könnte transient gewesen sein. Die Exception könnte ein Bug im Consumer sein, der bei jedem Versuch fehlschlägt.

Unbekannt ist eine echte Klasse, keine Lücke in der Taxonomie, und wie Sie sie behandeln, ist eine Richtlinienentscheidung. Mein Standard ist, unbekannte Ausfälle wie transiente zu behandeln, aber mit einer strengeren Grenze als bei den bekannten transienten, denn die Kosten eines fälschlichen Retrys bei einem unbekannten Ausfall sind ein paar vergeudete Versuche, während die Kosten, ihn fälschlich fallen zu lassen, verlorene Arbeit sind. Wenn die Versuche aller Ebenen aufgebraucht sind, wird ein unbekannter Ausfall wie ein permanenter behandelt: Er geht mit seinem vollständigen Verlauf in die Dead Letters.

Wenn das Ergebnis eines Versuchs unbekannt ist, fragen Sie nach, bevor Sie erneut handeln. Fragen Sie vor einem Retry, und erst recht vor einer Kompensation, beim Anbieter den Status der Operation über ihren Idempotenzschlüssel ab. Wenn der Einzug stattgefunden hat, ist die richtige Folgeaktion eine Erstattung, keine Annullierung. Wenn nicht, sind ein Retry oder eine Annullierung sicher. In beide Richtungen zu raten ist der Weg, auf dem ein Käufer für eine Bestellung belastet wird, die Parcelly für storniert hält.

Zurückgestellt: Die Abhängigkeit ist bekanntermaßen ausgefallen

Das vierte Signal ist kein Urteil über die Nachricht. Wenn ein Circuit Breaker (weiter unten beschrieben) offen ist, sagt das Fehlschlagen eines Aufrufs nichts über die Anfrage aus: Der Aufruf hat die Abhängigkeit nie erreicht. Diese Klasse nenne ich **zurückgestellt**. Eine zurückgestellte Nachricht verbraucht keinen Versuch, zählt zu keiner Grenze und wird deswe-

gen nie in die Dead Letters verschoben. Der Consumer hört auf zu lesen oder pausiert die Partition, und die Nachricht wartet dort, wo sie ist, bis es sich wieder lohnt, die Abhängigkeit aufzurufen.

Die Unterscheidung ist wichtig, weil die Alternative Arithmetik ist. Ein Breaker, der 30 Sekunden offen bleibt, vor einem Handler, der drei Versuche im Handler im Abstand von einer Sekunde erlaubt, lässt eine gesunde Nachricht ihr gesamtes Budget in etwa den ersten drei Sekunden verbrennen und wegen eines Problems in der DLQ landen, das sie nie hatte.

Die Klassifizierung explizit machen

Die Klassifizierung muss an einer Stelle leben, an der man sie lesen und prüfen kann. Eine Tabelle, die ein Ausfallsignal einer Klasse und einer Behandlung zuordnet, gehört zu den wertvollsten kleinen Artefakten in einem Consumer. Für die Aufrufe von Payments beim Anbieter sieht eine beispielhafte Version so aus:

Signal	Klasse	Behandlung
Verbindungsabbruch, DNS-Fehler	Transient	Retry mit Backoff
Timeout bei einem Einzugsaufruf	Unbekannt	Retry mit demselben Idempotenzschlüssel, strenge Grenze
`429`, Throttling-Antwort	Transient	Retry, den Wartehinweis des Anbieters beachten, falls vorhanden
`5xx` vom Anbieter	Transient	Retry mit Backoff
`400`, Validierungsfehler	Permanent	Kein Retry, in die Dead Letters verschieben
Karte abgewiesen	Permanent, geschäftliches Ergebnis	`PaymentFailed` ausgeben, kein Retry
Nicht erkannter Fehler	Unbekannt	Retry mit strenger Grenze, danach in die Dead Letters
Circuit Breaker offen	Zurückgestellt	Kein Versuch verbraucht; Konsum pausieren, fortsetzen, wenn der Breaker es erlaubt

Zwei Vorbehalte. Erstens sind Statuscodes Hinweise, keine Garantien; ein Anbieter, der für eine fehlerhafte Anfrage `500` zurückgibt, hat sie für Sie falsch klassifiziert, und das werden Sie aus der Erfahrung mit diesem An-

bieter lernen müssen. Zweitens sollte die Tabelle für jedes Signal, das sie noch nicht gesehen hat, standardmäßig auf unbekannt fallen, nie auf permanent oder transient. Ein neuer Fehler, den niemand klassifiziert hat, ist kein Beleg für irgendetwas.

Der Klassifizierer selbst ist klein:

```
// 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 mit Jitter

Sobald Sie sich für einen Retry entschieden haben, lautet die nächste Frage, wann. Sofort zu wiederholen gibt einer transienten Ursache keine Zeit abzuklingen. Nach einer festen Wartezeit zu wiederholen gibt ihr Zeit, hält aber alle Aufrufer im Gleichschritt.

Exponential Backoff lässt die Wartezeit mit jedem Versuch wachsen: sagen wir eine Sekunde, dann zwei, dann vier, bis zu einer Obergrenze. Frühe Retries sind schnell, was zu Ausfällen passt, die im Nu abklingen. Spätere Retries sind langsam, was eine Abhängigkeit entlastet, die wirklich ausgefallen ist. Die Obergrenze verhindert, dass die Wartezeit über das hinauswächst, was das Geschäft tolerieren kann.

Jitter fügt jeder Wartezeit Zufall hinzu. Ohne ihn berechnen Aufrufer, die im selben Moment fehlgeschlagen sind, dieselben Wartezeiten und wiederholen im Gleichschritt, und die Herde übersteht den Backoff unversehrt. Mit ihm verteilen sich die Retries über ein Intervall, und die Abhängigkeit sieht einen Schleier von Anfragen statt einer Wand.

Es gibt mehrere Jitter-Verfahren. Zwei sollte man mit Namen kennen. Beim **Full Jitter** ist die Wartezeit ein Zufallswert zwischen null und der exponentiellen Obergrenze für diesen Versuch. Beim **Equal Jitter** ist die Hälfte der Wartezeit fest und die Hälfte zufällig, sodass die Mindestwartezeit nie nahe null liegt. Full Jitter verteilt die Last am besten, während Equal Jitter eine Untergrenze für die Wartezeit garantiert. Ich würde mit Full Jitter beginnen und nur dann zu Equal Jitter wechseln, wenn die Abhängigkeit unter sehr schnellen Retries leidet.

```
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
```

Diese Schleife ist unter den Ausfällen, die sie behandelt, korrekt, aber sie ist in zwei Punkten unvollständig, die die nächsten Abschnitte aufgreifen. Sie weiß nicht, wie viel Zeit dem Aufrufer noch bleibt, und sie weiß nicht, wie viel Retry-Aufwand das Gesamtsystem bereits betreibt.

Wenn der Anbieter einen Hinweis darauf gibt, wann ein erneuter Versuch sinnvoll ist, etwa einen Wartewert in einer Throttling-Antwort, beachten Sie ihn. Der Anbieter kennt seine eigene Erholung besser als Ihre Formel. Behandeln Sie den Hinweis als Untergrenze unter Ihrer berechneten Wartezeit, nicht als Ersatz dafür, und behalten Sie den Jitter bei, damit Aufrufer mit demselben Hinweis nicht gemeinsam zurückkehren. Eine Untergrenze ist trotzdem keine Erlaubnis, über die Deadline hinaus zu schlafen: Wenn der Hinweis länger ist als die verbleibende Zeit, hören Sie auf zu wiederholen und überlassen die Entscheidung der nächsten Ebene.

Deadlines und Retry-Budgets

Backoff mit Jitter bestimmt den Abstand der Retries. Zwei weitere Grenzen bestimmen, wie viele es geben darf.

Deadlines

Eine **Deadline** ist der Moment, nach dem das Ergebnis nicht mehr nützlich ist. Ein anfragegetriebener Ablauf hat eine natürliche: Der Käufer starrt auf eine Checkout-Seite, und in 10 Sekunden wird er gehen. Ein Hintergrund-Consumer hat eine weichere: Ein Versandetikett, das nach dem Annahmeschluss des Paketdienstes eintrifft, ist so verspätet wie eines, das nie eintrifft.

Retry-Logik, die Versuche zählt, aber die Zeit ignoriert, kann weit über den Nutzen hinauslaufen. Drei Versuche mit je 30 Sekunden Timeout und wachsenden Wartezeiten dazwischen sind Minuten des Wartens. Ich bevorzuge es, jeder wiederholten Operation ein Gesamtzeitbudget zu geben und das Wiederholen zu beenden, wenn die nächste Wartezeit es überschreiten würde, was auch immer die Anzahl der Versuche sagt.

```
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)
```

Die Deadline sollte außerdem mitwandern. Wenn Orders einem Ablauf 10 Sekunden gibt und Payments aufruft, sollte Payments weitergeben, was übrig ist, statt eigene frische 10 Sekunden zu beginnen. Sonst wiederholt jede Ebene innerhalb der Geduld der vorherigen, und die Summe überschreitet unbemerkt, was der Nutzer abzuwarten bereit ist.

Retry-Budgets

Eine Versuchsgrenze pro Aufruf schützt eine Abhängigkeit nicht, wenn viele Aufrufe gleichzeitig fehlschlagen. Jeder Aufruf bleibt innerhalb seiner Grenze von drei Retries, und die Abhängigkeit erhält trotzdem das Vierfache ihres normalen Traffics.

Ein **Retry-Budget** begrenzt Retries als Anteil am gesamten Traffic, nicht pro Aufruf. Eine verbreitete Form ist ein Token Bucket: Jeder erfolgreiche Aufruf fügt einen Bruchteil eines Tokens hinzu, jeder Retry verbraucht eines, und wenn der Bucket leer ist, schlagen Aufrufe schnell fehl, ohne zu wiederholen. Im stationären Zustand sind Retries ein kleiner Bruchteil der

Anfragen. Bei einem weitreichenden Ausfall ist das Budget binnen Augenblicken aufgebraucht, und das System hört auf, den eigenen Ausfall zu verstärken.

Die Zahlen können Sie selbst wählen, und sie hängen davon ab, wie viel zusätzliche Last die Abhängigkeit aufnehmen kann. Ich nenne keinen Wert, denn jeder Wert, den ich nennte, wäre eine Vermutung über Ihre Abhängigkeit. Entscheidend ist, dass das Budget existiert, dass es von den Aufrufern in einem Prozess gemeinsam genutzt wird und dass sein Aufbrauchen auf einem Dashboard sichtbar ist. Ein verhältnisbasiertes Budget pro Prozess genügt meist, denn wenn jeder Prozess sein eigenes Verhältnis deckelt, ist damit bereits die Summe gedeckelt. Ein über Instanzen geteiltes Budget ist eine Option, keine strikte Verbesserung: Es braucht einen verteilten Zähler, und dieser Zähler ist eine neue Abhängigkeit auf dem Fehlerpfad, mit einem eigenen Ausfallmodus.

Der Preis ist, dass einige Aufrufe, die bei einem Retry erfolgreich gewesen wären, stattdessen fehlschlagen. Das ist der Sinn eines Budgets: Es tauscht den Erfolg Einzelner gegen das Überleben des Systems, und es sollte das mit Absicht tun.

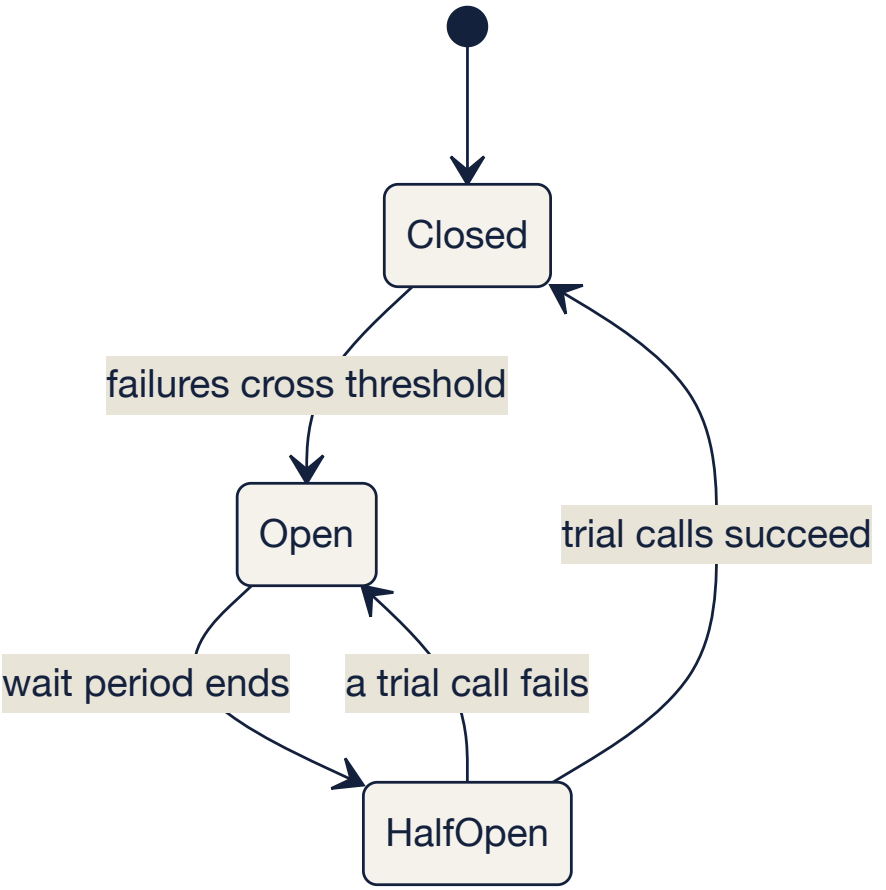
Retry-Storms und der Circuit Breaker

Ein **Retry-Storm** entsteht, wenn das Wiederholen zur dominierenden Last auf einer ausfallenden Abhängigkeit wird. Parcellys Eingangsszenario war einer. Er braucht drei Zutaten: eine Abhängigkeit, die langsam ist oder ausfällt, Aufrufer, die wiederholen, und keinen Mechanismus, der den Aufrufern sagt, sie sollen aufhören. Ebenen machen es schlimmer. Wenn Orders drei Versuche bei Payments macht und Payments drei Versuche beim Anbieter, kann ein einziger fehlgeschlagener Einzug zu neun Aufrufen beim Anbieter werden. Mit einer weiteren Ebene von drei Versuchen werden es 27.

Zwei Abwehrmaßnahmen greifen den Storm direkt an. Das oben beschriebene Retry-Budget begrenzt, wie viel das System wiederholt. Der Circuit Breaker begrenzt, ob es überhaupt aufrufen sollte.

Ein **Circuit Breaker** beobachtet die Ergebnisse der Aufrufe an eine Abhängigkeit und hat drei Zustände. Im Normalbetrieb ist er geschlossen, und Aufrufe gehen durch. Wenn Ausfälle eine Schwelle überschreiten, öffnet

er, und Aufrufe schlagen sofort fehl, ohne die Abhängigkeit zu berühren, was der Abhängigkeit Raum zur Erholung gibt. Nach einer Wartezeit geht er in den halboffenen Zustand über und lässt einige Probeaufrufe durch. Gelingen sie, schließt er. Wenn nicht, öffnet er erneut.



Der Breaker und die Retry-Richtlinie ergänzen sich und beantworten verschiedene Fragen. Der Backoff entscheidet, wie lange bei einem Aufruf bis zum nächsten Versuch gewartet wird. Der Breaker entscheidet anhand des Musters über die Aufrufe hinweg, ob Versuche überhaupt sinnvoll sind. Ein offener Breaker macht aus dem nächsten Versuch der Retry-Schleife einen sofortigen Ausfall. Dieser Ausfall ist nicht transient und er ist nicht unbekannt: Er ist das Signal **zurückgestellt** aus der obigen Klassifizierung. Die Retry-Schleife darf es nicht zählen, nicht mit Backoff darauf reagieren und nicht darauf kreisen. Sie gibt die Kontrolle an den Consumer zurück, der entscheidet, was er nicht mehr tut.

Ein Consumer hat eine zusätzliche Wahl, die einem anfragegetriebenen Aufrufer fehlt. Wenn der Breaker offen ist, sollte ein Consumer aufhören, Nachrichten vom Broker zu lesen, oder die Partition pausieren, statt Nachrichten zu konsumieren und jede davon fehlschlagen zu lassen. Nachrichten, die im Broker bleiben, sind so lange sicher, wie seine Aufbewahrung es erlaubt, sie kosten Speicher und erscheinen als Consumer-Lag. Nachrichten, die konsumiert, wiederholt und fehlgeschlagen werden, verbrennen ihre Versuchszähler und könnten wegen eines Problems in die Dead Letters verschoben werden, das nie ihres war. Das Pausieren ist die Form des Breakers auf der Consumer-Seite, und ich würde sie vor den meisten anderen Dingen in diesem Kapitel bauen.

Das Pausieren wirft zwei Anschlussfragen auf. Erstens testet bei fehlendem Datenfluss niemand den halboffenen Zustand. Setzen Sie mit einer Nebenläufigkeit von eins fort oder senden Sie einen synthetischen Probeaufruf, damit die Probeaufrufe von etwas gemacht werden, das weiß, dass es Proben sind. Zweitens muss ein pausierter Consumer auf einem logbasierten Broker seine Poll-Schleife am Leben halten, die Partitionen pausieren, aber nicht das Polling, sonst hält die Gruppe den Consumer für tot und weist seine Partitionen neu zu (das unten beschriebene Liveness-Fenster).

Der Preis eines Breakers ist, dass er fälschlich fehlschlagen kann. Zu empfindliche Schwellen öffnen bei einem kurzen Aussetzer und machen aus einem transienten Ausfall einen Ausfall des Dienstes. Zu nachsichtige Schwellen öffnen zu spät. Ein Breaker fügt außerdem Zustand hinzu, den jemand verstehen muss, wenn um drei Uhr nachts ein Ablauf nicht mehr funktioniert: Die erste Frage ist, ob die Abhängigkeit ausgefallen ist oder nur der Breaker offen.

Retries bei Überlast: Backpressure

Ein Breaker reagiert auf eine Abhängigkeit, die ausgefallen ist. **Backpressure** ist die Disziplin für die Phase davor, wenn eine Abhängigkeit langsam, gesättigt oder in der Erholung ist und das System entscheiden muss, wie viel Arbeit es ihr übergibt. Kapitel 3 verortet sie im Ausfallmodell. Dieser Abschnitt ist der Mechanismus.

Kehren Sie zum Eingangsszenario zurück. Die Langsamkeit des Anbieters kam nicht von einem Bug. Sie kam von mehr Arbeit, als er aufnehmen konnte, und jede Instanz von Payments fütterte ihn weiter. Ein Consumer ohne Grenze für In-Flight-Arbeit verhält sich genau so: Er liest weiter, startet weiter Aufrufe und legt weiter auf den Haufen, der schon zu schwer ist. Die folgenden Grenzen würde ich einziehen, ungefähr in dieser Reihenfolge.

- **Die In-Flight-Arbeit begrenzen.** Deckeln Sie die Zahl der Nachrichten, die ein Consumer gleichzeitig hält. Bei einem warteschlangenbasierten Broker ist das der Prefetch oder die In-Flight-Zahl. Bei einem logbasierten Broker ist es die Zahl der Records pro Poll plus die Zahl der laufenden Handler. Ohne Deckel werden der Speicher des Consumers und die Redelivery-Uhr des Brokers zur Grenze, und beide sind schlechtere Grenzen als eine, die Sie selbst gewählt haben.
- **Die Nebenläufigkeit senken, wenn die Abhängigkeit langsamer wird.** Eine Nebenläufigkeit, die bei 50 Millisekunden pro Aufruf richtig ist, ist bei fünf Sekunden pro Aufruf falsch, denn dieselbe Zahl von Slots hält jetzt ein Vielfaches an wartender Arbeit. Senken Sie die Zahl der gleichzeitigen Aufrufe, wenn die Latenz steigt, und erhöhen Sie sie langsam wieder, wenn die Latenz sich erholt.
- **Pausieren statt fehlschlagen.** Wenn die Abhängigkeit gesättigt ist, hören Sie auf zu lesen, wie bei einem offenen Breaker. Ein pausierter Consumer zeigt sich als Lag, was sichtbar und sicher ist. Ein Consumer, der liest und fehlschlägt, verbrennt Versuche.
- **Live-Traffic vor Retries bedienen.** Retries und Redrive konkurrieren mit ersten Versuchen um dieselbe Abhängigkeit. Wenn die Kapazität knapp ist, sollten erste Versuche gewinnen, denn ein Retry ist Arbeit, von der das System bereits weiß, dass sie Mühe macht. Betreiben Sie Retry-Topic- und Redrive-Consumer mit geringerer Nebenläufigkeit, oder lassen Sie sie warten, solange der Hauptfluss Backlog hat.
- **Am Eingang Last abwerfen.** Irgendwo stromaufwärts muss es eine Grenze für angenommene Arbeit geben, sonst wächst die Warteschlange unbegrenzt, während die Abhängigkeit sich erholt. Weisen Sie neue Arbeit ab oder stellen Sie sie zurück, mit einem klaren Signal an den Aufrufer, statt sie unbegrenzt hinter einer Abhängigkeit einzureihen, die sie nicht bedienen kann.

Diese Grenzen und das Retry-Budget sind zwei Sichten auf eine Idee. Das Budget deckelt, wie viel zusätzliche Arbeit das Wiederholen hinzufügt, und Backpressure deckelt, wie viel Arbeit, zusätzliche oder nicht, in Bearbeitung ist. Keines genügt allein: Ein Budget ohne In-Flight-Deckel lässt eine langsame Abhängigkeit jeden Slot mit ersten Versuchen füllen, und ein In-Flight-Deckel ohne Budget lässt die Slots sich mit Retries füllen.

Der Preis sind Lag und Abweisung. Ein System, das Last abwirft, sagt einigen Aufrufern mit Absicht Nein, und ein System, das pausiert, stellt manche Nachrichten verspätet zu. Beides ist ein Tausch von Durchsatz gegen Überleben, und wer die Antwort auf „welche Arbeit wartet“ verantwortet, ist der fachlich Verantwortliche des Ablaufs, nicht der Entwickler, der die Zahlen einstellt. Kapitel 8 wendet dieselbe Rate-Limit-Idee auf den Redrive an, und Kapitel 9 wendet die In-Flight-Grenze auf Lanes mit Schlüsseln an.

Wo Retries stattfinden

Eine Nachricht kann auf mehreren Ebenen wiederholt werden, und die Ebenen verhalten sich unterschiedlich. Sie ohne Kenntnis zu vermischen, ist der Weg, auf dem eine Nachricht weit öfter wiederholt wird, als jemand beabsichtigt hat, oder gar nicht.

Retry auf Anwendungsebene

Der Consumer wiederholt innerhalb des Handlers: der Code von oben, um einen Aufruf des Anbieters gelegt. Die Nachricht bleibt unbestätigt, während die Retries laufen. Diese Ebene ist billig, schnell und gut für Ausfälle, die in Millisekunden oder Sekunden abklingen. Ihre Grenzen sind, dass sie die Kapazität des Consumers während des Wartens bindet, dass ein Absturz des Consumers den Retry-Zustand verliert (der Broker stellt erneut zu, und der Versuchszähler beginnt von vorn, weshalb eine Nachricht, die den Prozess zum Absturz bringt, die in Kapitel 8 beschriebene Erkennung braucht), und dass lange Wartezeiten die eigenen Timeouts des Brokers überschreiten können.

Bei einem warteschlangenbasierten Broker ist die unbestätigte Nachricht für das **Visibility-Timeout** verborgen, nach dessen Ablauf sie erneut zugestellt wird. Wenn die Retries im Handler diesen Timeout überdauern, stellt

der Broker erneut zu, während die erste Kopie noch wiederholt wird, und nun laufen zwei Kopien. Bei einem logbasierten Broker kann ein Consumer, der zu lange zwischen zwei Polls verbringt, von seiner Consumer-Group für tot gehalten werden und seine Partitionen verlieren, mit demselben Ergebnis. Die Regel, der ich folgen würde: Die längste Gesamtzeit der Retries im Handler muss bequem innerhalb des Timeouts des Brokers liegen, und alles Längere gehört auf eine andere Ebene.

Erneute Zustellung durch den Broker

Der Consumer lässt die Nachricht fehlschlagen, und der Broker stellt sie erneut zu. Die beiden Brokerfamilien unterscheiden sich hier, und der Unterschied ist wichtig.

Bei einem warteschlangenbasierten Broker ist die erneute Zustellung eingebaut. Der Consumer sendet einen Nack für die Nachricht oder lässt den Visibility-Timeout ablaufen, und die Nachricht kehrt in die Warteschlange zurück, sofort oder nach einer konfigurierten Verzögerung, wo jeder konkurrierende Consumer sie erhalten kann.

Ein logbasierter Broker stellt nicht von selbst erneut zu. Die Leseposition des Consumers rückt beim Polling vor, und der per Commit festgehaltene Offset hält nur fest, wo ein neu gestarteter Consumer fortsetzen würde (Kapitel 2 behandelt das). Wenn die Verarbeitung eines Records fehlschlägt und sein Offset nicht per Commit festgehalten wird, geschieht daher von sich aus nichts: Der Consumer liest den nächsten Record, und wenn er dann einen Commit ausführt, deckt dieser Commit auch den fehlgeschlagenen Record ab, der stillschweigend übersprungen wird. Der fehlgeschlagene Record wird erst nach einem Neustart, einem Rebalance oder einem ausdrücklichen Schritt des Consumers erneut gelesen.

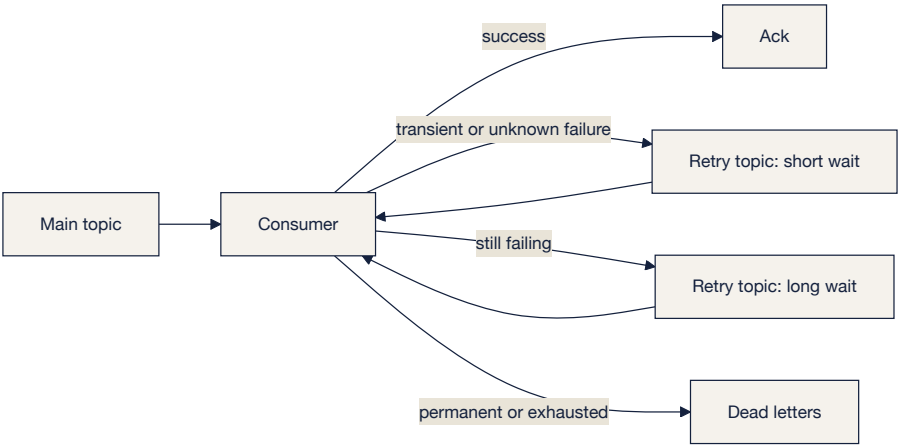
Auf einem Log muss der Consumer also eine von drei Aktionen wählen. Er kann auf den fehlgeschlagenen Offset zurückspringen oder die Partition pausieren und an Ort und Stelle wiederholen, was die Reihenfolge bewahrt und alles dahinter anhält. Er kann den Record in einen Retry-Topic veröffentlichen und weitermachen. Oder er kann ihn in einen Dead-Letter-Topic schreiben und weitermachen. Jede davon ist eine bewusste Entscheidung im Code des Consumers, nichts, was der Broker für Sie tut.

Die erneute Zustellung durch den Broker ist dauerhaft: Sie übersteht einen Absturz des Consumers. Sie ist aber auch grob. Viele Broker stellen sofort oder nach festem Zeitplan erneut zu, ohne Backoff, und der Versuchszähler ist unter Umständen nur als Zustellzähler verfügbar, den der Consumer lesen und auswerten muss. Behandeln Sie sie als Auffangnetz unter den anderen Ebenen, nicht als die eigentliche Retry-Richtlinie.

Retry-Topics und Delay-Queues

Wenn ein Ausfall eine längere Wartezeit braucht, als der Retry im Handler halten kann, besteht eine verbreitete Struktur darin, die Nachricht aus dem Hauptfluss herauszunehmen und später zurückkehren zu lassen. Ein Consumer, der eine Nachricht fehlschlagen lässt, veröffentlicht sie mit erhöhtem Versuchszähler und einem Not-before-Zeitpunkt in einen **Retry-Topic** (oder eine Retry-Queue) und bestätigt das Original. Ein separater Consumer, oder derselbe auf einem anderen Topic, nimmt sie auf, wenn die Zeit gekommen ist. Broker, die verzögerte Zustellung oder Verzögerungen pro Nachricht unterstützen, bieten das unmittelbar. Wo sie es nicht tun, ist eine verbreitete Anordnung eine Leiter von Retry-Topics, jedes mit einer festen Wartezeit, eines für jede Stufe des Backoffs.

Die Wartezeiten sind aus einem Grund pro Topic fest. Auf einem Log kann ein Consumer einen Record, der noch nicht fällig ist, nicht überspringen: Er muss am Head warten, bis der Record bereit ist. Wenn jeder Record in einem Topic dieselbe Wartezeit hat, ist der Topic bereits in der Reihenfolge der Fälligkeit, und das Warten am Head kostet nichts extra. Ein Topic, der kurze und lange Verzögerungen mischte, würde die kurzen hinter den langen festhalten. Not-before-Zeitpunkte pro Nachricht funktionieren nur bei Brokern mit nativer verzögerter Zustellung.



Diese Ebene entlastet den Hauptfluss, sodass eine fehlschlagende Nachricht die Nachrichten dahinter nicht mehr aufhält. Sie übersteht Neustarts, weil der Zustand im Broker liegt. Sie gibt Ihnen außerdem einen Ort zum Nachsehen: Die Tiefe des Retry-Topics ist ein direktes Maß dafür, wie viel Arbeit auf eine sich erholende Abhängigkeit wartet.

Sie kostet mehr Maschinerie. Es gibt mehr Topics, die man anlegen, benennen, überwachen und absichern muss. Das erneute Veröffentlichen ist selbst eine Produce-Operation, die fehlschlagen kann, was den Dual Write aus Kapitel 1 zurückbringt, daher kommt es auf die Reihenfolge an: zuerst in den Retry-Topic veröffentlichen, dann das Original bestätigen. Wenn der Prozess zwischen beiden stirbt, existiert die Nachricht doppelt. Das ist ein Duplikat, das der Consumer ohnehin tolerieren muss.

Drei Regeln verhindern, dass die Leiter beschädigt, was sie schützen soll:

- **Den Topic pro Consumer besitzen.** Ein Retry-Topic gehört einer Consumer-Group oder einem Abonnement. Wenn eine andere Gruppe es lesen kann, verarbeitet diese Gruppe eine Nachricht erneut, bei der sie nie fehlgeschlagen ist.
- **Die Identität bewahren.** Die erneut veröffentlichte Nachricht behält den ursprünglichen Schlüssel, die `event_id` und die fachlichen Header. Ein neuer Schlüssel zerstört die Reihenfolge, und eine neue `event_id` zerstört die Inbox.
- **Die Aufbewahrung an die Wartezeit anpassen.** Die Aufbewahrung eines Retry-Topics muss die längste Wartezeit plus den erwarteten Lag darauf übersteigen, sonst läuft die Nachricht während des Wartens ab.

Der wichtigste Preis ist die **Reihenfolge**, die der nächste Abschnitt behandelt.

Die Ebenen wählen

In der Praxis nutzt ein Consumer alle drei. Ein kurzer Retry auf Anwendungsebene fängt die Aussetzer ab, etwa einen Verbindungsabbruch oder einen Deadlock, die innerhalb einer oder zweier Sekunden abklingen. Ein Retry-Topic oder eine Delay-Queue deckt Ausfälle ab, die Minuten brauchen. Die erneute Zustellung des Brokers deckt Abstürze ab. Eine Dead-Letter-Queue (DLQ) beschließt die Kette und nimmt auf, was seine Versuche aufgebraucht hat oder nie gelingen kann; Kapitel 8 behandelt ihr Design.

Die Ebenen sollten gemeinsam entworfen werden, mit einem Gesamtversuchsbudget statt dreien unabhängigen. Wenn jede Ebene drei Versuche erlaubt, kann eine Nachricht 27-mal versucht werden, und das hat niemand geplant. Die Richtlinie am Ende dieses Kapitels macht die Multiplikation explizit und deckelt sie.

Retries und Nachrichtenreihenfolge

Das Wiederholen einer Nachricht ändert die Reihenfolge, in der Nachrichten verarbeitet werden. Das folgt aus der obigen Schichtung, und es bleibt unbemerkt, bis es Schaden anrichtet.

Betrachten Sie den Inventory-Consumer von Parcelly, der für dieselbe Bestellung zuerst `OrderPlaced` und dann `OrderCancelled` erhält. Angenommen, die Verarbeitung von `OrderPlaced` schlägt transient fehl, und der Consumer verschiebt sie in einen Retry-Topic und verarbeitet dann `OrderCancelled`. Die Stornierung findet keine Reservierung zum Freigeben, weil nach dem Kenntnisstand von Inventory noch keine existiert. Dann trifft das wiederholte `OrderPlaced` ein und erzeugt eine Reservierung, die nie jemand freigeben wird. Der Endzustand ist falsch, und jeder einzelne Schritt sah vernünftig aus.

Es gibt zwei ehrliche Optionen, und jede hat ihren Preis.

- **Blockieren und an Ort und Stelle wiederholen.** Bei einem logbasierten Broker springt der Consumer auf den fehlschlagenden Offset zu-

rück oder pausiert die Partition und wiederholt ihn mit Backoff, bevor er irgendetwas dahinter anfasst. Die Reihenfolge bleibt für diese Partition erhalten. Der Preis ist, dass die ganze Partition hinter einer Nachricht wartet, sodass eine einzige fehlschlagende Nachricht jedes Aggregat auf dieser Partition aufhalten kann. Das ist die richtige Wahl, wenn die Reihenfolge wichtiger ist als der Durchsatz, und es ist der Grund, warum es so wichtig ist, einen permanenten Ausfall schnell zu klassifizieren: Eine Poison Message am Head einer Partition blockiert alles dahinter.

- **Beiseiteschieben und Umordnung hinnehmen.** Die fehlschlagende Nachricht geht in einen Retry-Topic, und der Consumer macht weiter. Der Durchsatz bleibt erhalten. Der Preis ist, dass der Consumer Ankunft in falscher Reihenfolge tolerieren muss, durch Techniken wie Versionsprüfungen oder dadurch, dass auch jede spätere Nachricht für dasselbe Aggregat geparkt wird, bis die frühere gelingt. Kapitel 9 behandelt diese Techniken.

Ich würde pro Stream wählen. Bei einem Ereignis-Stream, in dem Nachrichten für ein Aggregat der Reihe nach angewendet werden müssen, würde ich an Ort und Stelle wiederholen und in schnelle Klassifizierung und einen Ausgang in die Dead Letters investieren. Bei einem Stream unabhängiger Fakten, bei dem die Reihenfolge über Nachrichten hinweg keine Rolle spielt, ist das Beiseiteschieben billig und hält den Fluss in Bewegung.

Bei warteschlangenbasierten Brokern ohne Gruppierungsfunktion ist die Reihenfolge über konkurrierende Consumer hinweg ohnehin schwach, sodass Retries eine bestehende Eigenschaft etwas verschlechtern, statt eine Garantie zu brechen. Dennoch sollte sich ein Consumer, der von einer Reihenfolge ausgeht, nicht darauf verlassen, dass eine Warteschlange sie liefert. Wenn Sie eine Gruppierungsfunktion nutzen (FIFO-Queues, Message Groups, Sessions), ist die Abwägung dieselbe wie auf einem Log: Eine fehlschlagende Nachricht blockiert ihre Gruppe, bis sie gelingt oder in die Dead Letters verschoben wird, so wie ein Retry an Ort und Stelle eine Partition blockiert.

Retries ohne Idempotenz

Jeder Retry ist ein wiederholter Versuch an Arbeit, deren früheres Ergebnis unbekannt sein kann. Wenn der frühere Versuch fehlschlug, bevor er etwas getan hat, ist der Retry harmlos. Wenn er fehlschlug, nachdem er die Arbeit getan hat, tut der Retry die Arbeit doppelt.

Das ist kein seltener Randfall. Betrachten Sie das Eingangsszenario. Payments ruft den Anbieter auf, um 4.990 kleinste Währungseinheiten in EUR zu erfassen. Der Anbieter bucht das Geld ab und sendet seine Antwort, und die Antwort geht verloren. Payments sieht einen Timeout. Der Timeout wird als unbekannt klassifiziert, und die Richtlinie sagt, es sei zu wiederholen. Die zweite Einzugsanfrage geht hinaus. Ob der Käufer einmal oder zweimal belastet wird, hängt jetzt ganz davon ab, ob etwas die zweite Anfrage als Wiederholung erkennbar macht.

Dieses Etwas ist ein **Idempotenzschlüssel**: ein stabiler Bezeichner, der mit der Anfrage gesendet wird und dem Empfänger erlaubt zu erkennen, dass er diese Arbeit schon gesehen hat. Beim Aufruf des Anbieters ist es bei jedem Versuch für denselben Einzug derselbe Schlüssel, den ein Anbieter, der solche Schlüssel beachtet, als einen Einzug behandelt. Beim Konsumieren von Ereignissen ist es normalerweise die `event_id`, in derselben Transaktion wie die Arbeit gegen die Inbox geprüft. Kapitel 7 baut beides auf.

> **Warnung:** Eine Retry-Richtlinie ist nur auf einer idempotenten Operation sicher. Bevor Sie Retries für einen Aufruf mit Wirkung aktivieren, etwa das Belasten einer Karte, das Senden einer E-Mail oder das Reservieren von Beständen, klären Sie, was den zweiten Versuch harmlos macht. Wenn nichts ihn harmlos macht, ist die richtige Zahl automatischer Retries null, und der Ausfall sollte an einen Menschen gehen. Da ein At-least-once-Broker nach einem Absturz oder Timeout erneut zustellt, was auch immer der Handler will, muss „null“ vom Handler durchgesetzt werden: Er muss erkennen, dass er diese Nachricht schon gesehen hat, oder vor der Wirkung bestätigen und At-most-once hinnehmen, und dann den Ausfall an einen Menschen weiterleiten.

Die Abhängigkeit gilt in beide Richtungen. Idempotenz macht Retries sicher, und Retries machen Idempotenz notwendig. Ein System, das aggressiv wiederholt, ohne idempotente Consumer zu haben, erzeugt doppelte Wirkungen mit einer Rate, die proportional zu seiner Ausfallrate ist, also genau in den Momenten, in denen das System ohnehin unter Stress steht und niemand Zeit zum Aufräumen hat.

Es gibt auch einen subtileren Fall. Eine Operation kann für denselben Schlüssel idempotent und trotzdem unsicher zu wiederholen sein, weil sich der Schlüssel ändert. Wenn ein Handler bei jedem Versuch einen neuen Bezeichner erzeugt, etwa eine neue UUID bei jedem Aufruf, sieht der Anbieter jedes Mal eine andere Anfrage. Der Schlüssel muss einmal aus stabilen Eingaben abgeleitet werden, etwa aus der `event_id` oder der Bestellung, und bei jedem Versuch wiederverwendet werden.

Eine Retry-Richtlinie für Parcelly

Um die Teile zusammenzuführen, hier das, was ich für den Consumer von `CapturePayment` bei Payments aufschreiben würde. Die Zahlen sind Illustrationen, keine Empfehlungen, und Ihre Abhängigkeit wird eigene brauchen.

```
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
```

Lesen Sie sie als eine Reihe von Entscheidungen, jede mit einem Grund. Die Ebene im Handler fängt kurze Aussetzer ab. Die Retry-Topics decken einen Anbieter ab, der Minuten ausgefallen ist. Die Deadline sagt, wann sich die Arbeit nicht mehr lohnt, und die Entscheidung darüber verantwortet das Geschäft, nicht der Entwickler. Ein veralteter Einzug muss möglicherweise zu einem `OrderCancelled` werden, aber die Kompensation hängt davon ab, was beim Anbieter tatsächlich geschehen ist. Wenn der letzte Versuch in einen Timeout lief, kann der Einzug erfolgreich gewesen sein, und dann ist die richtige Folgeaktion ein `PaymentRefunded`, kein `PaymentVoided`. Deshalb verlangt die Richtlinie zuerst den Status. Der einmal abgeleitete Schlüssel ist das, was die ganze Struktur davon abhält, einen Käufer zweimal zu belasten.

Zwei Zeilen der Richtlinie verdienen einen zweiten Blick. `maxTotalAttempts` existiert, weil die Ebenen sich multiplizieren: drei Versuche im Handler, mal ein erster Durchgang plus drei Stufen der Retry-Topics, sind bis zu 12 Aufrufe beim Anbieter, noch bevor irgendeine erneute Zustellung nach einem Absturz mitgezählt ist. Eine einzige Gesamtzahl von 8 (illustrativ) ist die Zahl, die tatsächlich jemand verantwortet. Und

der gesamte Retry-Horizont, hier zwei Stunden, muss kürzer sein als die Zeit, die der Anbieter sich Idempotenzschlüssel merkt. Anbieter bewahren Schlüssel nur für ein begrenztes Fenster auf, und ein Retry, der nach dem Fenster eintrifft, ist eine neue Anfrage mit einem alten Schlüssel.

Wenn der Redrive aus Kapitel 8 einen toten Einzug zurückbringt, beginnt der Versuchszähler von vorn, aber ein separater Redrive-Zähler bleibt bestehen, sodass der Verlauf nicht verloren geht und eine Nachricht, die immer wiederkehrt, als solche sichtbar ist.

Beachten Sie, was fehlt: eine einzelne Einstellung „Retry-Anzahl“. In einem System mit Ebenen verbirgt diese eine Zahl mindestens vier Entscheidungen.

Abwägungen und Kosten

Retries sind eines der billigsten Mittel, die scheinbare Zuverlässigkeit eines Systems zu verbessern, und eines der leichtesten, die man übernutzt. Ihre Kosten sind konkret.

Latenz. Jeder Retry fügt Wartezeit hinzu. Backoff und Jitter verteilen die Last, dehnen aber auch die Zeit bis zum Erfolg. Ein Ablauf, der minutenlang wiederholt, kann korrekt abschließen und trotzdem für den Käufer nutzlos sein, der nach 10 Sekunden gegangen ist. Wo ein Mensch wartet, würde ich einen schnellen Ausfall und eine klare Meldung einem langen stillen Retry vorziehen.

Verstärkung. Retries vervielfachen die Last, und Retries auf mehreren Ebenen vervielfachen sie erneut. Eine Richtlinie, die in jedem Service vernünftig ist, kann in der Kette gefährlich sein. Jemand muss das Ende-zu-Ende-Versuchsbudget verantworten, denn kein einzelner Service kann es sehen.

Kapazität. Ein Consumer, der an Ort und Stelle wiederholt, belegt während des Wartens seinen Slot. Bei einem langen Ausfall können alle Consumer am Ende warten, und das System kommt auch bei Arbeit, die gelingen würde, nicht mehr voran. Den Konsum über einen Breaker zu pausieren, die In-Flight-Arbeit mit Backpressure zu begrenzen und lange Wartezeiten in Retry-Topics zu verlagern, sind die üblichen Abwehrmittel.

Verborgene Ausfälle. Ein Retry, der schließlich gelingt, sieht von außen wie ein Erfolg aus. Wenn Sie Retries und Versuche nicht zählen, kann eine Abhängigkeit wochenlang schlechter werden hinter einer Richtlinie, die das stillschweigend auffängt. Kapitel 13 behandelt, was zu messen ist: Versuchszahlen, Retry-Rate und das Alter der ältesten wiederholten Nachricht gehören auf ein Dashboard.

Komplexität. Retry-Topics, Delay-Queues und Budgets sind Infrastruktur, die jemand bauen, benennen, überwachen und erklären muss. Für ein kleines System mit einem Consumer und einer Abhängigkeit kann ein Retry im Handler mit Backoff und einem Ausgang in die Dead Letters alles sein, was Sie brauchen, und eine Leiter von Retry-Topics kann mehr Maschinerie sein, als das Problem verdient.

Wann nicht wiederholen. Wiederholen Sie keine permanenten Ausfälle. Wiederholen Sie keine Operationen mit Wirkung, die nicht idempotent sind. Wiederholen Sie nicht über die Deadline hinaus. Wiederholen Sie nicht, wenn ein Circuit Breaker sagt, dass die Abhängigkeit ausgefallen ist, und belasten Sie diese Nachricht dafür nicht mit einem Versuch. Wiederholen Sie nicht innerhalb einer Ebene, die bereits innerhalb des Retrys einer anderen Ebene sitzt, es sei denn, Sie haben die Multiplikation durchgerechnet. Und wiederholen Sie nicht, um eine Entscheidung zu vermeiden: Ein Ausfall, der immer wiederkehrt, ist eine Information, und die richtige Reaktion ist oft, innezuhalten und hinzusehen.

Häufige Fehler

Alles gleich wiederholen

Es ist verlockend, weil eine einzige Schleife der kürzeste Code ist. Sie behandelt eine Kartenabweisung wie einen Verbindungsabbruch und verschwendet Kapazität auf Ausfälle, die nicht gelingen können. Klassifizieren Sie zuerst, und geben Sie jeder Klasse ihre eigene Behandlung.

Feste Wartezeiten ohne Jitter

Eine konstante Wartezeit wirkt vorhersehbar und einfach. Sie synchronisiert alle Aufrufer, die gemeinsam fehlgeschlagen sind, zu einer sich wiederholenden Spitze. Verwenden Sie Exponential Backoff mit Jitter, und meiner Ansicht nach Full Jitter als Ausgangspunkt.

Versuche zählen, aber nicht die Zeit

Eine Versuchsgrenze sieht aus wie eine Sicherheitsschranke. Mit langen Timeouts und wachsenden Wartezeiten kann sie dennoch minutenlang über den Nutzen hinauslaufen. Setzen Sie eine Deadline für die gesamte Operation, und hören Sie auf, wenn die nächste Wartezeit sie überschreiten würde.

Auf jeder Ebene wiederholen

Der Verantwortliche jeder Ebene fügt einen vernünftigen Retry hinzu, und die Multiplikation ist für keinen von ihnen sichtbar. Legen Sie ein Gesamtversuchsbudget für die Kette fest, und setzen Sie Retries dorthin, wo sie am wenigsten kosten.

Wiederholen, während der Breaker offen ist

Es ist verlockend, die Schleife weiterlaufen zu lassen, weil die Schleife funktionierte. Ein Consumer, der Nachrichten liest und gegen eine ausgefallene Abhängigkeit fehlschlagen lässt, verbraucht deren Versuche und kann sie fälschlich in die Dead Letters verschieben. Behandeln Sie den offenen Breaker als zurückgestelltes Signal, das keinen Versuch kostet, pausieren Sie den Konsum und lassen Sie die Nachrichten im Broker warten.

Einen Consumer ohne Grenze lesen lassen

Ein unbegrenzter Consumer ist der Standard, und er fühlt sich nach maximalem Durchsatz an. Gegen eine langsame Abhängigkeit füllt er jeden Slot mit wartender Arbeit und füttert den Retry-Storm. Begrenzen Sie die In-Flight-Nachrichten, senken Sie die Nebenläufigkeit, wenn die Latenz steigt, und lassen Sie erste Versuche gegenüber Retries gewinnen.

Die Timeouts des Brokers ignorieren

Retries im Handler wirken lokal für den Handler. Wenn sie den Visibility-Timeout oder das Liveness-Fenster der Gruppe überdauern, stellt der Broker erneut zu oder weist neu zu, und zwei Kopien der Arbeit laufen gleichzeitig. Halten Sie die Gesamtzeit im Handler deutlich innerhalb der Grenzen des Brokers.

Über eine Nachricht hinwegwiederholen, die in Reihenfolge bleiben muss

Eine fehlschlagende Nachricht in einen Retry-Topic zu verschieben, ist eine schnelle Lösung für eine festgefahrene Partition. Sie ordnet die Ereignisse dieses Aggregats um. Entscheiden Sie zuerst die Anforderung an die Reihenfolge, und wählen Sie dann zwischen Blockieren und Beiseiteschieben.

Nicht idempotente Wirkungen wiederholen

Die Retry-Schleife ist an einem Nachmittag hinzugefügt, und die Idempotenz wird verschoben. Der erste Timeout nach einem erfolgreichen Aufruf erzeugt eine doppelte Wirkung. Klären Sie, was den zweiten Versuch harmlos macht, bevor der erste Retry aktiviert wird.

Bei jedem Versuch einen neuen Schlüssel erzeugen

Es sieht nach Sorgfalt aus, da jede Anfrage eindeutig identifiziert ist. Es vereitelt den Zweck, weil der Empfänger eine Wiederholung nicht erkennen kann. Leiten Sie den Schlüssel einmal aus stabilen Eingaben ab, und verwenden Sie ihn wieder.

Retries nicht zählen

Wenn nur endgültige Ausfälle ins Log geschrieben werden, bleibt eine sich verschlechternde Abhängigkeit unsichtbar, bis die Retries nicht mehr genügen. Halten Sie jeden Versuch fest, mit seiner Klasse und Wartezeit.

Zentrale Erkenntnisse

- Ein Retry ist das Versprechen, es erneut zu versuchen, nicht das Versprechen, dass es gelingt, daher lautet die erste Frage bei jedem Ausfall, ob ein erneuter Versuch überhaupt helfen kann.
- Klassifizieren Sie Ausfälle als transient, permanent oder unbekannt, halten Sie die Klassifizierung als Tabelle fest und behandeln Sie nicht erkannte Ausfälle als unbekannt mit einer strengeren Grenze.
- Verwenden Sie Exponential Backoff mit Jitter, begrenzen Sie die gesamte Operation mit einer Deadline und teilen Sie ein Retry-Budget, damit Retries nicht zur dominierenden Last auf einer ausfallenden Abhängigkeit werden können.

- Ein Circuit Breaker entscheidet, ob überhaupt aufgerufen wird; ein offener Breaker ist ein zurückgestelltes Signal, das keinen Versuch kostet, und für einen Consumer ist die stärkste Form, den Konsum zu pausieren. Backpressure leistet dieselbe Arbeit früher, indem sie die In-Flight-Arbeit begrenzt und den Live-Traffic gegenüber Retries und Redrive gewinnen lässt.
- Retries können im Handler, in der erneuten Zustellung des Brokers und in Retry-Topics leben, und sie sollten gemeinsam unter einem Versuchsbudget entworfen werden, wobei die Zeit im Handler innerhalb der Timeouts des Brokers bleibt.
- Eine fehlschlagende Nachricht beiseitezuschieben ordnet die Ereignisse ihres Aggregats um; wählen Sie zwischen Blockieren an Ort und Stelle und Tolerieren der Umordnung entsprechend der Anforderung des Streams an die Reihenfolge.
- Eine Retry-Richtlinie ist nur auf idempotenten Operationen sicher, wobei der Idempotenzschlüssel einmal abgeleitet und bei jedem Versuch wiederverwendet wird.

Retries entscheiden, wie oft Sie es erneut versuchen, aber sie können nicht entscheiden, ob zweimaliges Versuchen harmlos ist. Kapitel 7 baut die andere Hälfte auf: den idempotenten Consumer, der aus einem Duplikat von einer Katastrophe ein Nicht-Ereignis macht.

Kapitel 7 — Idempotenz

Stellen Sie sich den Ledger-Service von Parcelly mitten in einem schrittweisen Deployment vor, während seine Consumer nacheinander neu gestartet werden. Ein Consumer hat ein Ereignis `PaymentCaptured` gelesen, einen Ledger-Eintrag über 4.990 kleinste Währungseinheiten in EUR geschrieben und will die Nachricht gerade bestätigen, als der Prozess sein Shutdown-Signal erhält. Die Bestätigung verlässt die Maschine nie. Der Broker tut genau das, was At-least-once-Zustellung verspricht, und übergibt dieselbe Nachricht dem nächsten Consumer, der danach fragt.

Der zweite Consumer liest das Ereignis, findet nichts Ungewöhnliches und schreibt einen zweiten Ledger-Eintrag. Das Ledger ist append-only und per Konstruktion korrekt, also hält es nun getreu und dauerhaft fest, dass der Käufer zweimal bezahlt hat. Nichts ist abgestürzt. Kein Alarm wurde ausgelöst. Jede Komponente hat sich wie vorgesehen verhalten, und die Bücher sind falsch.

Notifications erlebt denselben Nachmittag in seiner eigenen Variante. Ein Käufer erhält die E-Mail „Ihr Paket ist eingeplant“ dreimal, weil die Nachricht erneut zugestellt wurde, nachdem ein langsamer Handler seinen Visibility-Timeout überschritten hatte. Das ist peinlich. Die Sache im Ledger ist teuer.

Frühere Kapitel haben festgestellt, dass Duplikate normal sind: Der Broker stellt erneut zu, der Relay veröffentlicht nach einem Failover doppelt, und die Retries aus Kapitel 6 sind Duplikate per Konstruktion. Dieses Kapitel beantwortet die Frage, die jene Kapitel offenlassen. Wenn dieselbe Nachricht zweimal eintrifft, was genau soll ein Consumer tun, damit das zweite Eintreffen nichts verändert?

Das Problem und warum es schwierig ist

Das Ziel klingt bescheiden: jede Nachricht einmal verarbeiten, soweit die Welt es erkennen kann. Die Schwierigkeit besteht darin, dass der Consumer nicht wissen kann, ob er eine Nachricht schon gesehen hat, ohne sich zu erinnern, und das Erinnern ist selbst ein Schreibvorgang, der im falschen Moment fehlschlagen kann.

Beginnen Sie damit, woher Duplikate kommen, denn die Liste ist länger als der Broker.

Quelle	Wie das Duplikat entsteht
Erneute Zustellung durch den Broker	Die Bestätigung ging verloren, der Visibility-Timeout ist abgelaufen, oder ein Rebalance hat die Partition verschoben, bevor der Offset per Commit festgehalten wurde
Relay	Ein Relay hat eine Zeile veröffentlicht, ist abgestürzt, bevor er sie als veröffentlicht markiert hat, und hat sie nach dem Neustart erneut veröffentlicht (Kapitel 5)
Retry der Anwendung	Ein Consumer hat einen Aufruf wiederholt, oder ein Producer hat eine Veröffentlichung wiederholt, deren erster Versuch tatsächlich gelungen war (Kapitel 6)
Redrive und Replay	Jemand hat Dead Letters zurückverschoben oder ein Ereignislog absichtlich ab einem früheren Offset erneut gelesen
Menschen und Clients	Ein Käufer hat doppelt auf „Bezahlen“ geklickt, eine mobile App hat in einem instabilen Netz wiederholt, ein Webhook-Sender hat zweimal zugestellt

Die meisten davon erzeugen *dieselbe* Nachricht zweimal, mit derselben `event_id`. Einige erzeugen zwei verschiedene Nachrichten, die dieselbe Absicht ausdrücken, und diese Unterscheidung wird wichtig, wenn wir einen Schlüssel wählen.

Die verlockende Lösung ist, vor dem Handeln nachzusehen. Fragen Sie, ob diese Nachricht schon verarbeitet wurde, und verarbeiten Sie sie, wenn nicht. Ich möchte das als **Check-then-act**-Antipattern benennen, denn es fühlt sich richtig an und scheitert lautlos.

```
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)
```

Lesen Sie es unter Ausfall. Stirbt der Prozess nach `postLedgerEntry` und vor `inboxInsert`, existiert der Eintrag, und die Inbox weiß es nicht, sodass die erneute Zustellung ihn noch einmal bucht. Vertauschen Sie die beiden Schreibvorgänge, und der Ausfall kehrt sich um: Ein Absturz nach `inboxInsert` und vor dem Schreiben ins Ledger markiert die Nachricht als erledigt, obwohl sie es nicht war, und die Wirkung ist für immer verloren. Keine Reihenfolge ist sicher, weil die beiden Schreibvorgänge nicht ein atomarer Akt sind.

Es gibt einen zweiten Ausfall, der mit Abstürzen nichts zu tun hat. Zwei Consumer können dieselbe Nachricht fast im selben Moment erhalten, etwa wenn ein langsamer erster Versuch seinen Visibility-Timeout überschreitet. Beide führen die Prüfung aus, beide finden nichts, und beide machen weiter. Das Fenster zwischen „Check“ und „Act“ ist klein, und bei ausreichendem Volumen wird es routinemäßig getroffen.

Das eigentliche Problem hat also zwei Hälften. Das Gedächtnis darüber, was verarbeitet wurde, muss atomar mit der Wirkung geschrieben werden, und das Gedächtnis muss zwischen gleichzeitigen Versuchen entscheiden können. Der Rest des Kapitels baut beides auf, beginnend mit dem Fall, in dem Sie keines von beiden brauchen.

Natürliche und erzwungene Idempotenz

Eine Operation ist **idempotent**, wenn ihre zweimalige Anwendung das System im selben Zustand hinterlässt wie ihre einmalige. Manche Operationen sind durch ihre Form idempotent, und nach diesen würde ich suchen, bevor ich irgendeine Maschinerie baue.

Das Setzen eines Werts ist idempotent: `set order.status = "PAID"` kann 10-mal laufen, und die Bestellung ist immer noch `PAID`. Das Löschen einer Zeile über ihre ID ist idempotent. Das Einfügen einer Zeile mit einem Primärschlüssel, den Sie aus dem Ereignis ableiten, und das Ignorieren des Konflikts ist idempotent. Das Schreiben des vollständigen neuen Zustands eines Aggregats („die Sendung ist jetzt `DISPATCHED`, mit dieser Tracking-Nummer“) ist idempotent. Ich nenne das **natürliche Idempotenz**: Das Duplikat hat keine zusätzliche Wirkung wegen dessen, was die Operation ist, nicht wegen irgendetwas, das Sie hinzugefügt haben.

Andere Operationen sind es nicht. `balance = balance + 4990` wird zweimal angewendet zu zwei Einzahlungen. `send email` sendet zwei E-Mails. `append a ledger entry` hängt zwei an. `increment the counter of units sold` zählt die Einheit doppelt. Der Aufruf des Einzugs-Endpoints eines Zahlungsanbieters ohne Schlüssel kann doppelt belasten. Bei diesen muss Idempotenz hinzugefügt werden, was ich **erzwungene Idempotenz** nenne: Sie halten fest, dass die Arbeit getan wurde, und schlagen darin nach, bevor Sie sie erneut tun.

Natürliche Idempotenz hat zwei ehrliche Grenzen. Die erste ist, dass sie stillschweigend von der Reihenfolge abhängt. Das Setzen von `status = "PAID"` ist idempotent, aber ein verspätetes Duplikat eines älteren Ereignisses kann einen neueren Status überschreiben: ein erneut zugestelltes `OrderPlaced`, das nach `OrderConfirmed` eintrifft und den Status auf `PLACED` zurücksetzt. Die Operation ist idempotent, und das System ist trotzdem falsch. Kapitel 9 behandelt Reihenfolge und Versionsprüfungen; merken Sie sich vorerst, dass „auf einen Wert setzen“ nur dann sicher gegen Duplikate ist, wenn der Wert nicht veralten kann.

Die zweite Grenze ist, dass man leicht zur Überschätzung neigt. Ein Handler, der „nur eine Zeile aktualisiert“, schreibt oft auch einen Audit-Eintrag, gibt ein Folgeereignis aus oder erhöht einen Zähler, und jedes davon ist ein verkleidetes Inkrement. Wenn ich einen Consumer prüfe, liste ich jede seiner Wirkungen auf und frage bei jeder einzeln, ob sie einen zweiten Lauf übersteht. Eine einzige nicht idempotente Wirkung genügt, um die Inbox zu brauchen.

Meine Faustregel lautet, natürliche Idempotenz zu bevorzugen, wo die Wirkung es zulässt, und im Handler festzuhalten, warum sie gilt. Ist die Wirkung ein Zähler, ein Anhängen, eine E-Mail oder ein externer Aufruf,

verwenden Sie die erzwungene Art. Und im Zweifel verwenden Sie trotzdem die Inbox: Ihre Kosten sind gering und vorhersehbar, und die Kosten eines Irrtums sind es nicht.

Den Idempotenzschlüssel wählen

Ein **Idempotenzschlüssel** ist das, womit der Consumer wiederholte Arbeit erkennt. Bevor Sie die Tabelle bauen, müssen Sie entscheiden, was darin steht, und die Wahl bestimmt, welche Duplikate Sie abfangen können.

Die event_id

Für den Konsum von Ereignissen ist der Standard die `event_id` aus dem Envelope. Sie ist pro Ereignis eindeutig und, entscheidend, **stabil über die erneute Zustellung hinweg**: Jede Kopie der Nachricht, die der Broker Ihnen übergibt, trägt denselben Wert. Der Producer vergibt sie einmal, wenn das Ereignis in die Outbox geschrieben wird (Kapitel 4), sodass ein Relay, der doppelt veröffentlicht, dieselbe `event_id` zweimal veröffentlicht. Das ist genau die Eigenschaft, die eine Deduplizierungsprüfung braucht.

Die `event_id` identifiziert eine *Nachricht*. Sie beantwortet „habe ich diese konkrete Tatsache schon behandelt?“ und nichts Umfassenderes.

Der fachliche Schlüssel

Manchmal beschreiben zwei verschiedene Nachrichten dieselbe Absicht. Angenommen, Orders wiederholt ein Command `CapturePayment` nach einem Timeout, und wegen eines Bugs oder einer Designentscheidung wird jeder Retry als neues Command mit einer frischen `command_id` geschrieben. Payments erhält nun zwei Commands mit verschiedenen IDs, die beide „den Zahlungseinzug für Bestellung 81724 durchführen“ bedeuten. Die Deduplizierung über die Nachrichten-ID fängt keinen von beiden.

Das Mittel dagegen ist ein **fachlicher Schlüssel**: ein Bezeichner, der aus der Bedeutung der Arbeit abgeleitet ist und nicht aus dem Envelope. Für den Einzug könnte es die `payment_id` plus die Operation sein, etwa `capture:payment-5521`. Für einen Ledger-Eintrag könnte es die Zahlungs-ID plus die Art des Eintrags sein. Ein fachlicher Schlüssel sagt „diese Arbeitseinheit darf einmal stattfinden“, gleichgültig, wie viele Nachrichten sie anfordern.

Aspekt	`event_id` / `command_id`	Fachlicher Schlüssel
Fängt die erneute Zustellung derselben Nachricht	Ja	Ja
Fängt zwei Nachrichten für dieselbe Absicht	Nein	Ja
Wer ihn vergibt	Der Producer, automatisch	Eine menschliche Entscheidung pro Operation
Risiko	Übersieht logische Duplikate	Ein zu grober Schlüssel weist legitime Arbeit ab
Wo gespeichert	Generische Inbox-Tabelle	Oft ein Unique Constraint auf der Domänentabelle

Das Risiko in der vorletzten Zeile verdient einen konkreten Fall. Wenn der fachliche Schlüssel für Erstattungen nur die `payment_id` ist, wird eine zweite, legitime Teilerstattung derselben Zahlung als Duplikat abgewiesen. Der Schlüssel muss alles enthalten, was zwei Anfragen *tatsächlich verschieden* macht: hier vielleicht einen Idempotenzschlüssel für die Erstattung, den der Aufrufer liefert. Den Schlüssel zu wählen bedeutet zu entscheiden, was in Ihrer Domäne „dieselbe Arbeit“ heißt, und das ist eine Designentscheidung, keine Framework-Einstellung.

Meiner Ansicht nach arbeiten die Schichten gut zusammen. Verwenden Sie die `event_id` in einer generischen Inbox als Grundlage für jeden Consumer, und fügen Sie überall dort einen Unique Constraint auf den fachlichen Schlüssel in der Domänentabelle hinzu, wo zwei verschiedene Nachrichten plausibel dasselbe bedeuten könnten. Die Inbox schützt vor dem Broker und dem Relay. Der Constraint schützt vor dem Rest des Systems, einschließlich seiner Bugs.

Das Inbox-Pattern

Die **Inbox** ist die Tabelle der verarbeiteten Nachrichten-IDs auf der Consumer-Seite. Kapitel 4 hat dem Producer eine transaktionale Outbox gegeben, damit eine Zustandsänderung und ihr Ereignis gemeinsam per Commit festgehalten werden. Die Inbox ist das Spiegelbild: Der Consumer hält

seine Zustandsänderung und den Vermerk, die Nachricht behandelt zu haben, gemeinsam per Commit fest, in einer lokalen Transaktion in seiner eigenen Datenbank.

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

Die Spalte `consumer` erlaubt es einem Service, mehrere unabhängige Consumer gegen eine Tabelle zu betreiben, sodass der Posting-Consumer des Ledgers und sein Reporting-Consumer jeweils ihre eigene Historie behalten. Deshalb ist der Primärschlüssel das Paar `(consumer, event_id)`. Mit `event_id` allein würde der erste Consumer, der eine Nachricht behandelt, den Insert des zweiten Consumers scheitern lassen, und die Wirkung des zweiten Consumers würde übersprungen, als wäre sie ein Duplikat. Ein Service mit einem einzigen Consumer pro Tabelle kann die Spalte weglassen und über `event_id` schlüsseln, was die einfachere Form ist, die an anderen Stellen im Buch verwendet wird.

Der Handler schreibt die Inbox-Zeile und die fachliche Änderung in derselben Transaktion.

```
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)
```

Gehen Sie die Ausfälle durch. Stirbt der Prozess vor dem Commit, existieren weder der Ledger-Eintrag noch die Inbox-Zeile, sodass die erneute Zustellung die Nachricht von vorn verarbeitet. Stirbt er nach dem Commit und vor der Bestätigung, existieren beide, und die erneute Zustellung

nimmt den schnellen Pfad: Die Prüfung findet die Zeile, die Nachricht wird bestätigt, und nichts wird geschrieben. Die Wirkung und ihr Gedächtnis können sich nie widersprechen, weil sie als eines per Commit festgehalten werden.

Die Prüfung am Anfang ist eine Optimierung. Sie spart im häufigen Duplikatfall eine Transaktion und lässt sich entfernen, ohne die Korrektheit zu verlieren. Das Sicherheitsnetz ist der Unique Constraint innerhalb der Transaktion, und dort findet die eigentliche Entscheidung statt.

Innerhalb einer Transaktion ändert die Reihenfolge der Schreibvorgänge nichts an der Atomarität, sodass die Warnung vor dem Vertauschen der beiden Schreibvorgänge nur für die unsichere Version oben gilt. Sie ändert aber die Kosten. In dieser Version leistet ein gleichzeitiges Duplikat die ganze Arbeit, bevor es beim Insert scheitert, und in manchen Datenbanken bricht eine Verletzung die gesamte Transaktion ab, sodass der Handler sich darin nicht erholen kann. Ich würde zuerst beanspruchen: Fügen Sie die Inbox-Zeile mit der Semantik „on conflict do nothing“ ein, sehen Sie nach, ob eine Zeile eingefügt wurde, und überspringen Sie die Wirkung, wenn nicht. Das Duplikat wartet dann auf den Schlüssel, leistet keine Arbeit und löst nie einen Fehler aus. Das Beispiel im nächsten Abschnitt verwendet diese Form.

Ein Wort zu `ack`, denn der Begriff verbirgt einen Unterschied zwischen den beiden Brokerfamilien. Bei einem warteschlangenbasierten Broker bestätigt er eine einzelne Nachricht. Bei einem logbasierten Broker ist er meist ein Commit eines Offsets, was „alles bis hierher“ bedeutet, und ein Consumer, der eine Partition parallel verarbeitet, kann einen Commit über eine Nachricht hinaus ausführen, die er noch nicht abgeschlossen hat, und sie verlieren. Die Inbox schützt vor Duplikaten, nicht davor, und Kapitel 9 betrachtet es dort, wo Reihenfolge und Parallelität sich treffen.

Folgeereignisse eines Consumers

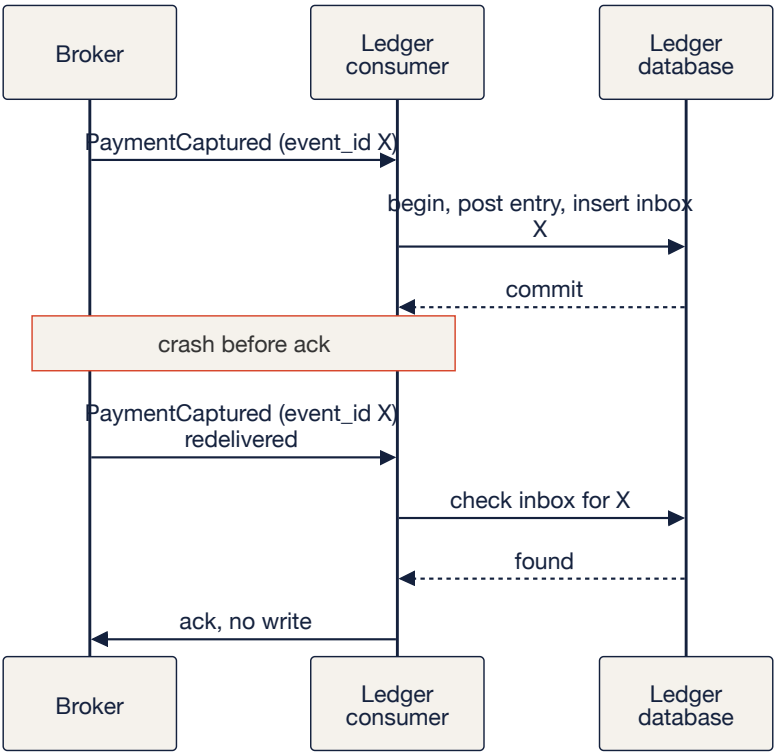
Viele Consumer tun mehr, als ihren eigenen Zustand zu ändern: Sie kündigen an, was sie getan haben. Das Ledger bucht einen Eintrag und gibt `LedgerEntryPosted` aus. Dieses Folgeereignis ist eine Wirkung wie jede andere, und es muss einen Absturz genau einmal überstehen, nicht nullmal und nicht zweimal.

Ein Handler, der das Folgeereignis direkt nach seinem Commit veröffentlicht, verliert es bei einem Absturz dauerhaft. Die erneute Zustellung nimmt den schnellen Pfad der Inbox, findet die Zeile, bestätigt und überspringt die Veröffentlichung. Der Zustand stimmt, und das Ereignis hat nie existiert. Die Lösung ist die Outbox aus Kapitel 4, auf der konsumierenden Seite angewendet: Die Inbox-Zeile, die Zustandsänderung und der Outbox-Insert werden in einer lokalen Transaktion per Commit festgehalten.

```
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)
```

Stirbt der Prozess vor dem Commit, existiert nichts, und die erneute Zustellung erledigt alles. Stirbt er nach dem Commit, liegt das Folgeereignis in der Outbox und der Relay veröffentlicht es, während die erneute Zustellung die Inbox-Zeile findet und nichts tut. Die Outbox des Consumers hat dieselbe At-least-once-Veröffentlichung wie die des Producers, sodass jeder, der `LedgerEntryPosted` konsumiert, eine eigene Inbox braucht. Kapitel 10 baut auf dieser Regel auf, wenn ein Saga-Schritt auf ein Ereignis reagiert und das nächste ausgibt.

Das ist auch der Grund, warum die Inbox in *derselben Datenbank* liegt wie der Zustand, den sie schützt. Läge der Vermerk über verarbeitete Nachrichten in einem Cache oder in einer anderen Datenbank, wären wir wieder bei zwei Schreibvorgängen ohne gemeinsame Transaktion, dem Dual Write aus Kapitel 4 auf der Consumer-Seite. Die Inbox ist keine Technik, die man von außen an den Datenspeicher eines Services anflanschen kann.



Sehen Sie sich die Notiz in der Mitte an: Das Absturzfenster liegt zwischen dem Commit und der Bestätigung, und die Inbox-Zeile macht aus der erneuten Zustellung ein No-op.

Gleichzeitige Duplikate

Die obige Sequenz setzt voraus, dass das Duplikat eintrifft, nachdem der erste Versuch abgeschlossen ist. Nebenläufigkeit ist schwieriger, und der Unique Constraint ist es, der Sie rettet.

Stellen Sie sich zwei Consumer vor, die dieselbe Nachricht gleichzeitig verarbeiten. Beide beginnen eine Transaktion. Beide erledigen die Arbeit. Beide versuchen, die Inbox-Zeile einzufügen. Die Datenbank lässt genau einen von ihnen festschreiben; der andere erhält eine Verletzung des Unique Constraints, seine Transaktion wird zurückgerollt, und sein Ledger-Eintrag verschwindet mit ihr. Der Verlierer lässt die Nachricht dann unbestätigt oder bestätigt sie nach erneuter Prüfung, und in beiden Fällen enthält das Ledger einen Eintrag.

Das ist der ganze Mechanismus, und es lohnt sich zu sagen, was er nicht verlangt: keine verteilte Sperre, keine Koordination zwischen den beiden Consumern, keine Annahme über das Timing. Der Constraint der Datenbank ist eine einzige Entscheidungsstelle, die beide Versuche passieren müssen.

```
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
```

Ob eine Unique-Verletzung bestätigt oder wiederholt werden soll, ist eine kleine Entscheidung, die man bewusst treffen sollte. Liegt die Verletzung auf dem Inbox-Schlüssel, wurde die Nachricht von einer festgeschriebenen Transaktion behandelt, und die Bestätigung ist richtig. Liegt sie auf einem anderen Constraint innerhalb von `applyChange`, bedeutet der Fehler etwas ganz anderes, und die Bestätigung würde einen echten Bug verbergen. Meiner Ansicht nach sollte der Handler sie unterscheiden, was er tun kann, indem er den Namen des verletzten Constraints prüft, wie der obige Code es tut.

Längere Handler und Arbeit in Bearbeitung

Das obige Pattern hält den ganzen Handler in einer Transaktion. Das ist in Ordnung, wenn die Arbeit aus ein paar Schreibvorgängen besteht. Unangenehm wird es, wenn der Handler etwas Langsames aufruft, denn dann bleibt die Transaktion offen, Sperren bleiben gehalten, und der Visibility-Timeout kann während des Wartens ablaufen. Die abgelaufene Nachricht wird einem zweiten Consumer erneut zugestellt, der dieselbe langsame Arbeit parallel beginnt.

Es gibt zwei tragfähige Antworten. Die erste ist, die Transaktion kurz zu halten: Erledigen Sie die langsame externe Arbeit *außerhalb* von ihr, mit dem Ansatz für externe Wirkungen im nächsten Abschnitt, und legen Sie nur die lokale Zustandsänderung und die Inbox-Zeile in die Transaktion. Die zweite, wenn die Arbeit wirklich beansprucht werden muss, ist, das Beanspruchen mit einem Status explizit zu machen.

```
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)
```

Ein Consumer fügt die Zeile als `processing` mit einem Lease ein, und dieser Anspruch wird in seiner eigenen kurzen Transaktion per Commit festgehalten, bevor die langsame Arbeit beginnt. Ein Duplikat, das ein gültiges Lease findet, darf die Nachricht nicht bestätigen, denn der erste Versuch kann noch fehlschlagen, und die Bestätigung würde den einzigen Retry auslöschen. Es stellt die Nachricht mit einer Verzögerung wieder in die Warteschlange, länger als das verbleibende Lease, und sieht bei der nächsten Zustellung erneut nach. Eines, das ein abgelaufenes Lease findet, darf mit einem neueren Lease-Token übernehmen, den der ursprüngliche Halter prüfen muss, bevor er seine lokalen Änderungen per Commit festhält; eines, das `done` findet, bestätigt.

Das Token schützt nur das, was es prüfen kann. Die lokale Transaktion kann es: Sie sperrt die Inbox-Zeile, vergleicht das Token und rollt zurück, wenn ein neuerer existiert. Der langsame externe Aufruf lässt sich nicht zurückrollen, also erhält die Wirkung den deterministischen Schlüssel aus dem nächsten Abschnitt, und ein veralteter Halter, der aufwacht und den Aufruf wiederholt, wird vom Downstream erkannt. Das ist die Fencing-Idee aus dem Relay-Lease von Kapitel 5, mit derselben Grenze: Ein Fence funktioniert nur dort, wo die geschützte Ressource das Token prüft.

Das Schema funktioniert, und es fügt eine Zustandsmaschine hinzu, über die jemand nachdenken muss, einschließlich der Frage, was geschieht, wenn ein Worker stirbt, während er ein Lease hält. Ich würde danach nur greifen, wenn der erste Ansatz nicht verfügbar ist, und ich würde das Lease kurz und erneuerbar halten.

> **Warnung:** Verlassen Sie sich nicht auf ein vorheriges `SELECT` oder eine Zeilensperre, um Duplikate fernzuhalten. Ob ein zweiter Consumer die nicht festgeschriebene Arbeit des ersten sieht, hängt vom Isolationsgrad der Datenbank ab, und eine Prüfung, die unter einem Grad korrekt ist, versagt unter einem anderen. Auch das Anheben des Grads auf Serializable macht den Constraint nicht überflüssig: Es ersetzt ein stilles Duplikat durch einen Serialisierungsfehler, den der Handler als wiederholbar behandeln und von vorn neu ausführen muss. Sperren verringern Konkurrenz; der Unique Constraint innerhalb der Transaktion ist die Garantie.

Deduplizierungsfenster und Aufbewahrung

Eine Inbox, die jede ID für immer behält, wächst für immer. Irgendwann werden Sie alte Zeilen löschen wollen, und das Löschen verändert die Garantie auf eine Weise, die einen Namen verdient: das **Deduplizierungsfenster**. Eine Nachricht, deren Inbox-Zeile gelöscht wurde, kann erneut verarbeitet werden, wenn sie jemals wieder auftaucht.

Wie lang muss das Fenster also sein? Es muss die längste Zeitspanne abdecken, nach der ein Duplikat plausibel nach dem Original eintreffen könnte. Ich kenne diese Zahl für Ihr System nicht, und sie hängt von einer kurzen Liste von Dingen ab.

- **Aufbewahrung des Brokers und Horizont der erneuten Zustellung.** Eine Warteschlange, die vier Tage lang erneut zustellt, oder ein Log, das eine Woche aufbewahrt, bedeutet, dass Duplikate so spät eintreffen können.
- **Praxis bei Dead Letters und Redrive.** Wenn ein Team Dead Letters eine Woche nach ihrem Fehlschlag per Redrive zurückbringen darf, muss das Fenster länger sein (Kapitel 8).
- **Replay.** Das erneute Lesen eines Ereignislogs ab einem alten Offset erzeugt Duplikate von Ereignissen, die so alt sind wie der Replay. Entweder ist das Fenster lang genug, oder ein Replay läuft gegen einen Consumer, der absichtlich zurückgesetzt wird, was eine Prozedur ist, die Sie vorab entwerfen.

- **Relay-Failover.** Doppelte Veröffentlichungen nach einem Failover liegen meist zeitlich nahe am Original und damit weit innerhalb jedes Fensters, das Sie wählen würden.

Eine verbreitete Form ist eine Aufbewahrungsdauer, die deutlich länger ist als die eigene Aufbewahrung des Brokers, sagen wir als Illustration einige Wochen, mit einem geplanten Job, der in Batches löscht. Die Inbox-Tabelle nach Zeit zu partitionieren macht das Löschen billig, denn eine ganze Partition zu verwerfen kostet weit weniger, als Millionen von Zeilen zu löschen.

Bedenken Sie, dass die Inbox nicht die einzige Verteidigungslinie ist. Ein Constraint auf den fachlichen Schlüssel in der Domänentabelle lebt so lange wie die Daten und schützt daher vor Duplikaten, die eintreffen, nachdem die Inbox-Zeile verschwunden ist. Das ist ein weiterer Grund, beide Schichten zu nutzen: Die Inbox gibt Ihnen billigen Schutz mit begrenztem Speicher gegen den häufigen Fall, und der Constraint deckt den Rest ab.

Wenn das System es akzeptieren kann, gibt es eine Alternative zu einem Zeitfenster: den Schlüssel aus etwas Monotonem abzuleiten. Die Regel ist enger, als sie zunächst aussieht. Sie gilt für zustandstragende Ereignisse, die Art, die Kapitel 9 von Deltas unterscheidet, bei denen jedes Ereignis den vollständigen neuen Zustand eines Aggregats enthält. Für diese speichert der Consumer die höchste angewendete Version des Aggregats und weist alles ab, was nicht neuer ist. Der Offset ist eine Position innerhalb einer Partition und die Version ist eine Eigenschaft eines Aggregats, die beiden sind also nicht austauschbar: Ein Offset sagt nichts über Aggregate, die auf anderen Partitionen leben.

Bei Delta-Ereignissen ist „abweisen, wenn nicht neuer“ falsch, denn eine übersprungene Version bedeutet eine verlorene Änderung. Ein solcher Consumer wendet Version `last + 1` an, puffert eine spätere, bis die Lücke sich füllt, und behandelt alles bei oder unter `last` als Duplikat; Kapitel 9 beschreibt das Puffern und seine Grenzen. In beiden Fällen ist das Gedächtnis ohne TTL begrenzt, und es hängt von Reihenfolgegarantien ab, die Kapitel 9 untersucht. Es passt zu Aggregaten mit einem Versionszähler und nicht zu Consumern, die aus mehreren Partitionen gespeist werden.

Externe Seiteneffekte

Die Inbox schützt Wirkungen, die in Ihrer eigenen Datenbank liegen. Viele Wirkungen tun das nicht. Eine E-Mail verlässt Ihr System in dem Moment, in dem Sie sie senden, und ein Aufruf beim Zahlungsanbieter belastet eine Karte, ob Ihre Transaktion später festgeschrieben wird oder nicht. Keine lokale Transaktion kann diese zurückrollen, also greift der Atomaritätstrick nicht.

Betrachten Sie Payments bei der Behandlung eines Commands `CapturePayment`. Es muss den externen Anbieter aufrufen und dann das Ergebnis festhalten.

```
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)
```

Bei einem Absturz zwischen dem Aufruf des Anbieters und der Transaktion ist das Geld geflossen, und Payments hat keinen Vermerk. Bei der erneuten Zustellung ruft der Handler den Anbieter erneut auf. Ob der Käufer nun doppelt belastet wird, hängt ganz vom Anbieter ab.

Den Schlüssel an den Downstream weitergeben

Die Lösung besteht darin, den Idempotenzschlüssel an das System weiterzugeben, das die Wirkung ausführt, damit es die Wiederholung erkennen kann. Zahlungsanbieter unterstützen das häufig: Sie senden mit der Anfrage einen Schlüssel, und eine zweite Anfrage mit demselben Schlüssel liefert das erste Ergebnis zurück, statt erneut zu handeln. Die Einzelheiten des Headers oder Felds unterscheiden sich je nach Anbieter, und ich würde die Dokumentation Ihres Anbieters für sein genaues Verhalten und dafür lesen, wie lange er Schlüssel behält.

Das wichtigste Detail ist, *wie Sie den Schlüssel wählen*. Er muss **deterministisch aus der Arbeit abgeleitet sein, nicht pro Versuch erzeugt**. Ein zufälliger Schlüssel, der im Handler erzeugt wird, sieht bei jedem Retry aus wie eine frische Anfrage und schützt nichts.

```
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
```

Verfolgen Sie nun die Ausfälle erneut. Ein Absturz nach dem Aufruf des Anbieters und vor dem Commit lässt das Geld bewegt und Payments ahnungslos zurück. Das erneut zugestellte Command führt denselben Code mit demselben Schlüssel aus, der Anbieter liefert das ursprüngliche Ergebnis, und die Transaktion hält es fest. Ein Absturz während des Aufrufs beim Anbieter, bei dem die Antwort verloren geht, ist dieselbe Situation: Der Retry trägt denselben Schlüssel, und der Anbieter führt den Einzug entweder einmal aus oder liefert das frühere Ergebnis. Zwei Consumer, die sich beim selben Command ein Rennen liefern, senden gleichzeitig denselben Schlüssel, und ein Anbieter, der den zweiten als „in Bearbeitung“ abweist, hat die Arbeit nicht scheitern lassen: Der Handler behandelt diese Antwort als wiederholbar und sieht bei der nächsten Zustellung erneut nach.

Das Gedächtnis des Anbieters und die Inbox sind nicht derselbe Schutz. Ein Anbieter merkt sich Schlüssel nur für begrenzte Zeit, und ich würde seine Dokumentation lesen, wie lange. Eine erneute Zustellung, die nach dieser Frist eintrifft, würde einen zweiten Einzug auslösen. Die lokale In-

box-Zeile, die mindestens so lange behalten wird wie das oben beschriebene Deduplizierungsfenster, hält den Handler davon ab, den Anbieter überhaupt aufzurufen, sodass die Schlüsselaufbewahrung des Anbieters nie das Fenster ist, auf das Sie sich verlassen.

Wenn die Arbeit fachlich statt nach der Nachricht geschlüsselt ist, verwenden Sie auch hier einen fachlich abgeleiteten Schlüssel, etwa die Zahlungs-ID plus die Operation. Dann erreichen selbst zwei Commands mit verschiedenen IDs den Anbieter als eine Anfrage.

Wenn der Downstream keinen Schlüssel kennt

Nicht jeder Downstream akzeptiert einen. Einfaches SMTP hat keinen Idempotenzschlüssel, und viele E-Mail- und SMS-Anbieter bieten keinen verlässlichen an. Dann stehen Sie vor der Wahl zwischen zwei unvollkommenen Ergebnissen, und ich würde sie ausdrücklich treffen, pro Wirkung.

Senden Sie vor dem Festhalten, kann ein Absturz zu einer doppelten E-Mail führen. Halten Sie vor dem Senden fest, kann ein Absturz zu einer fehlenden führen. Das sind At-least-once- und At-most-once-Verhalten für die Wirkung. Bei einer Benachrichtigung „Ihr Paket ist eingeplant“ ist ein seltenes Duplikat billig und eine verlorene Nachricht ein Support-Ticket, also würde ich zuerst senden und die gelegentliche Wiederholung hinnehmen. Bei einem einmaligen Gutscheincode, der Wert gewährt, würde ich in die andere Richtung neigen und eine Möglichkeit hinzufügen, einen fehlenden Versand zu erkennen und zu beheben.

Manche Wirkungen lassen sich sicherer machen, indem man die *Absicht* in der lokalen Transaktion festhält und die Wirkung von einem separaten Worker ausführen lässt, was die Outbox-Idee auf Commands angewendet ist. Das beseitigt das Problem an der Grenze zum Anbieter nicht, weil der Worker zwischen dem Aufruf und der Markierung immer noch abstürzen kann. Es verlagert das Problem aber an einen Ort, an dem Sie es sehen, gezielt wiederholen und abgleichen können. Kapitel 13 behandelt den Abgleich: Ihre Aufzeichnungen mit denen des Anbieters zu vergleichen und die Unterschiede zu beheben, was die letzte Verteidigung für genau die Wirkungen ist, die sich nicht idempotent machen lassen.

Idempotenzschlüssel und HTTP-APIs

Dasselbe Problem tritt an Ihrer eigenen Haustür auf. Das Telefon eines Käufers sendet „Bestellung aufgeben“, das Netz verliert die Antwort, und die App wiederholt. Orders kann einen Retry nicht von einer zweiten Bestellung unterscheiden, es sei denn, der Client sagt es.

Die weit verbreitete Konvention ist ein Request-Header `Idempotency-Key`: Der **Client** erzeugt für jede logische Operation einen eindeutigen Wert und sendet bei jedem Retry davon denselben Wert. Das ist die andere Hälfte der weiter oben gegebenen Definition: Beim Konsum von Ereignissen ist der Schlüssel normalerweise die `event_id`, die der Producer vergeben hat, und bei Client-Anfragen wird er vom Aufrufer geliefert. Aufgabe des Servers ist es, einen wiederholten Schlüssel wie die erste Anfrage wirken zu lassen.

Ein Server braucht mehr als den Schlüssel. Er speichert den Schlüssel zusammen mit einem Fingerprint der Anfrage und der Antwort, die er erzeugt hat.

```
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
```

Die Fingerprint-Prüfung ist wichtig. Ein Client, der einen Schlüssel für einen anderen Body wiederverwendet, hat einen Bug, und die stillschweigende Rückgabe der ersten Antwort würde ihn verbergen. Der Zustand „in Bearbeitung“ ist wichtig für den Fall der Nebenläufigkeit: Zwei gleichzeitige Anfragen mit einem Schlüssel dürfen nicht beide ausgeführt werden. Die Zeilensperre ordnet nur eine Übernahme einer bestehenden Zeile; der erste Anspruch wird vom Unique Constraint entschieden, weshalb der unterlegene Insert abgefangen und mit 409 beantwortet wird.

Das Lease ist wichtig für den Absturzfall, und sein Token für den langsamen Fall. Ein Halter, der über sein Lease hinaus pausiert wurde, kann aufwachen und versuchen, fertigzustellen, aber `requireLeaseToken` sieht das neuere Token, und seine gesamte Transaktion, einschließlich Bestellung und Outbox-Zeilen, wird zurückgerollt. Hier wirkt das Token als Fence, weil die geschützte Ressource, die lokale Datenbank, es innerhalb der Transaktion prüft, dieselbe Idee wie das Relay-Lease aus Kapitel 5. Die Länge des Leases verkürzt nur die Überlappung; das Token ist es, der den Commit des veralteten Halters scheitern lässt. Der veraltete Aufrufer erhält einen Fehler, und sein Retry erhält die gespeicherte Antwort. Beide Prüfungen verwenden die Uhr der Datenbank, `dbNow()`, nicht die des Anwendungsservers, aus den Gründen, die Kapitel 3 zu Uhren nennt. Den Schlüssel auf den Aufrufer zu beschränken verhindert, dass ein Client mit einem anderen kollidiert.

Der Pseudocode zeigt den Erfolgspfad, und auch der Fehlerpfad braucht eine Regel. Scheitert die Anfrage aus einem deterministischen Grund, etwa einem Validierungsfehler, der ein 4xx zurückgibt, speichert der Server diese Antwort als die abgeschlossene. Ein Retry mit demselben Schlüssel erhält dann dasselbe 4xx und führt die Validierung nicht erneut gegen Daten aus, die sich geändert haben können. Scheitert sie aus einem transienten Grund, etwa einem Timeout oder einer nicht verfügbaren Abhängigkeit, wird der Anspruch freigegeben oder läuft ab, damit der Retry des Clients die Arbeit ausführen kann. Einen transienten Fehler als endgültige Antwort zu cachen, würde aus einem Aussetzer einen dauerhaften Fehler für diesen Schlüssel machen.

Der Unique Constraint auf `(caller, key)` spielt dieselbe entscheidende Rolle wie der Primärschlüssel der Inbox. Es ist dasselbe Pattern: ein atomarer Anspruch, ein gespeichertes Ergebnis und eine deterministische Antwort für eine Wiederholung. Wo die API Ereignisse erzeugt, können die Bestellzeile, die Outbox-Zeilen und der Abschluss des Idempotenz-Eintrags alle in einer lokalen Transaktion geschrieben werden, sodass eine wiederholte Anfrage weder eine zweite Bestellung anlegt noch ein zweites `OrderPlaced` ausgibt. Behalten Sie den gespeicherten Schlüssel mindestens so lange, wie Clients wiederholen dürfen, was eine Entscheidung über das Verhalten der Clients ist und nicht über Broker.

Eine erwähnenswerte Verbindung: Der vom Client gelieferte Schlüssel kann in das Ereignis einfließen. Orders kann ihn in die Header des Ereignisses legen oder die `event_id` von `OrderPlaced` aus ihm ableiten, sodass nachgelagerte Consumer vom Tippen des Käufers bis zum Ledger-Eintrag eine Identität sehen. Wird die `event_id` abgeleitet, muss sie deterministisch aus Aufrufer und Schlüssel zusammen entstehen, zum Beispiel als namensbasierte UUID über beide, damit zwei Aufrufer, die zufällig denselben Schlüssel wählen, nie eine Identität teilen. Kapitel 11 diskutiert, was in ein Ereignis gehört; ob der Schlüssel dorthin gehört, hängt davon ab, wie weit Idempotenz der Anfrage folgen soll.

Abwägungen und Kosten

Die Inbox fügt jeder Nachricht einen Schreibvorgang hinzu. Jede konsumierte Nachricht kostet einen zusätzlichen Insert, innerhalb derselben Transaktion, auf dem Hot Path. Für einen Consumer mit einer Handvoll Schreibvorgängen ist der Mehraufwand gering. Bei einem sehr hochvolumigen, sehr billigen Consumer, etwa einem, der einen Cache für ein Lesemodell aktualisiert, kann er die dominierenden Kosten sein, und natürliche Idempotenz ist dort oft die bessere Wahl.

Speicher und Bereinigung der Inbox sind eine laufende Aufgabe. Die Tabelle wächst mit dem Traffic, und die Aufbewahrungsrichtlinie ist eine Entscheidung, die Sie verantworten und erneut prüfen. Ein zu kurzes Fenster lässt alte Duplikate wieder zu; ein zu langes kostet Speicher und Indexgröße. Jemand muss den Bereinigungsjob beobachten, denn wenn er stehen bleibt, wächst die Tabelle unbemerkt, bis sie zum Performanceproblem wird.

Ein Unique Constraint auf dem Hot Path kann zum Engpass durch Konkurrenz werden. Mit einem gut verteilten Schlüssel ist er billig. Bei einem Schlüssel, der sich häuft, oder bei vielen Consumern, die auf ein heißes Aggregat zugreifen, kann es zu Warteschlangen kommen. Die Tabelle nach Consumer und Event-ID zu schlüsseln und die Transaktion des Handlers kurz zu halten, hält das beherrschbar.

Fachliche Schlüssel sind eine Modellierungsentscheidung, und eine falsche ist schwer zu erkennen. Zu fein, und logische Duplikate passieren; zu grob, und legitime Arbeit wird abgewiesen. Der Fehler ist in beide

Richtungen lautlos. Ich würde den Schlüssel mit absichtlich unbequemen Fällen testen, etwa zwei legitimen Operationen auf einem Aggregat, bevor ich mich auf ihn verlasse.

Idempotenz deckt nicht alles ab. Sie macht eine Wiederholung harmlos; sie repariert keinen Handler, der schon beim ersten Mal falsch ist, sie ordnet keine Nachrichten, und sie macht keine Wirkung rückgängig, für die der Anbieter keinen Schlüssel kennt. Eine Inbox schützt auch nicht vor einem Duplikat, das sich *verändert* hat: Wenn ein Producer „dieselbe" Tatsache mit einer anderen `event_id` erneut veröffentlicht und kein fachlicher Schlüssel existiert, erkennt nichts sie.

Wann man die Inbox nicht verwendet. Wenn jede Wirkung des Consumers natürlich idempotent ist und nicht veralten kann, etwa das Überschreiben eines denormalisierten Felds mit dem neuesten vollständigen Zustand, ist die Inbox Kosten ohne Nutzen, und ich würde sie auslassen und die Begründung im Code festhalten. Ungeeignet ist sie außerdem bei einem Consumer, der nicht in eine eigene transaktionale Datenbank schreiben kann, etwa einem zustandslosen Forwarder. Ein solcher Consumer braucht einen Downstream, der dedupliziert, mit weitergegebenem Schlüssel, oder er muss so umgestaltet werden, dass er einen eigenen Speicher besitzt.

Häufige Fehler

- **Prüfen und Handeln in getrennten Schritten.** Es ist verlockend, weil der Code sich natürlich liest: „habe ich das schon gesehen? nein, dann tue ich es." Es scheitert bei einem Absturz zwischen beiden und bei Nebenläufigkeit. Legen Sie den Inbox-Schreibvorgang in dieselbe Transaktion wie die Wirkung, und lassen Sie den Unique Constraint entscheiden.
- **Die Inbox-Zeile nach der Bestätigung schreiben oder bevor die Wirkung festgeschrieben ist.** Zuerst zu bestätigen verliert die Nachricht, wenn die Arbeit dann scheitert. Vor dem Commit der Wirkung festzuhalten markiert Arbeit als erledigt, die nie stattgefunden hat. Die Wirkung, die Inbox-Zeile und der Commit sind eine Einheit; die Bestätigung folgt dem Commit.

- **Die Inbox in einem anderen Speicher halten als die Wirkung.** Ein Cache oder eine separate Datenbank fühlt sich billiger an und erzeugt den Dual Write neu. Halten Sie sie in derselben Datenbank wie den Zustand, den sie schützt.
- **Pro Versuch einen frischen Schlüssel erzeugen.** Ein zufälliger Wert, der im Handler oder in der Retry-Schleife des Clients erzeugt wird, lässt jeden Retry neu aussehen. Leiten Sie den Schlüssel aus dem Command, dem Ereignis oder der fachlichen Operation ab, damit er bei jedem Versuch identisch ist.
- **Nur `event_id` verwenden und annehmen, das decke alle Duplikate ab.** Sie fängt zwei verschiedene Nachrichten für dieselbe Absicht nicht. Fügen Sie einen Constraint auf einen fachlichen Schlüssel hinzu, wo die Domäne ein solches Risiko hat.
- **Ein Deduplizierungsfenster durch Raten wählen.** Ein Fenster, das kürzer ist als die Aufbewahrung des Brokers oder Ihre Redrive-Praxis, lässt alte Duplikate stillschweigend wieder zu. Leiten Sie das Fenster aus diesen Zahlen ab, und schreiben Sie die Herleitung auf.
- **Eine Unique-Verletzung als generischen Fehler behandeln.** Jede Verletzung zu bestätigen verbirgt echte Bugs; jede zu wiederholen läuft bei einer bereits behandelten Nachricht endlos im Kreis. Fangen Sie die Inbox-Verletzung für sich ab.
- **Die externe Wirkung vergessen.** Die Inbox kann eine E-Mail oder eine Kartenbelastung nicht zurückrollen. Geben Sie dem Anbieter einen deterministischen Schlüssel mit, oder entscheiden Sie bewusst zwischen einem möglichen Duplikat und einem möglichen Verlust.
- **Folgeereignisse direkt aus dem Handler veröffentlichen.** Ein Absturz zwischen dem Commit und der Veröffentlichung verliert das Ereignis, und die erneute Zustellung nimmt den schnellen Pfad der Inbox und wiederholt es nie. Schreiben Sie das Folgeereignis in derselben Transaktion wie die Inbox-Zeile und die Zustandsänderung in die Outbox.
- **Eine Nachricht bestätigen, deren Lease jemand anderem gehört.** Ein Duplikat, das ein gültiges Lease sieht und bestätigt, hat den einzigen Retry entfernt, falls der erste Versuch dann scheitert. Stellen Sie sie stattdessen mit einer Verzögerung wieder in die Warteschlange.

Zentrale Erkenntnisse

- Duplikate kommen vom Broker, vom Relay, von Retries, von Redrive und Replay sowie von Clients, daher sollte jeder Consumer so geschrieben sein, als könne jede Nachricht zweimal eintreffen.
- Check-then-act ist unsicher, weil die Prüfung und die Wirkung getrennte Schritte sind und ein Absturz oder ein gleichzeitiger Versuch dazwischen schlüpfen kann.
- Die Inbox-Zeile und die fachliche Änderung müssen in einer lokalen Transaktion in der eigenen Datenbank des Consumers per Commit festgehalten werden, und der Unique Constraint auf den Schlüssel entscheidet tatsächlich über gleichzeitige Duplikate. Ein Consumer, der ankündigt, was er getan hat, fügt dieser Transaktion eine Outbox-Zeile hinzu.
- Verwenden Sie die `event_id` als grundlegenden Idempotenzschlüssel für den Konsum von Ereignissen, und fügen Sie einen fachlichen Schlüssel hinzu, wo zwei verschiedene Nachrichten dieselbe Absicht ausdrücken können.
- Das Deduplizierungsfenster muss die Aufbewahrung des Brokers, Ihre Redrive-Praxis und jeden geplanten Replay überdauern, und ein Constraint auf den fachlichen Schlüssel in der Domänentabelle schützt Sie, nachdem Inbox-Zeilen gelöscht wurden.
- Senden Sie bei externen Seiteneffekten einen deterministischen Idempotenzschlüssel, der aus der Arbeit und nicht aus dem Versuch abgeleitet ist, behalten Sie die lokale Inbox-Zeile als langfristigen Schutz, weil der Anbieter Schlüssel vergisst, und wählen Sie dort, wo der Downstream keinen Schlüssel kennt, bewusst zwischen einem möglichen Duplikat und einem möglichen Verlust.
- An der API-Grenze liefert der Aufrufer den Idempotenzschlüssel, und der Server speichert den Schlüssel, einen Fingerprint der Anfrage und die Antwort, damit eine Wiederholung das erste Ergebnis zurückgibt.

Idempotenz macht eine wiederholte Nachricht harmlos, setzt aber voraus, dass die Verarbeitung irgendwann gelingt. Manche Nachrichten werden das nie: Sie scheitern bei jedem Versuch, und die Retries und erneuten Zustellungen türmen sich hinter ihnen auf. Kapitel 8 befasst sich mit diesen Poison Messages und mit den Dead Letters, die sie hinterlassen.

Kapitel 8 — Poison Messages und Dead Letters

Stellen Sie sich Parcelly am Morgen vor, nachdem ein neues Feld, `gift_message`, mit `OrderPlaced` ausgerollt wurde. Notifications liest es, um die Bestätigungs-E-Mail zu bauen, und die Testbestellung eines Verkäufers enthält eine Geschenknachricht mit einem Zeichen, das die E-Mail-Vorlage nicht kodieren kann. Der Consumer wirft bei genau dieser einen Nachricht eine Exception. Die Nachricht kommt zurück. Der Consumer wirft erneut.

Bis zum Mittag ist die Nachricht mehrere tausend Mal zugestellt worden. Jeder Versuch scheitert an derselben Stelle, auf dieselbe Weise, innerhalb von Millisekunden. Der Consumer ist beschäftigt, nach allen Infrastrukturmetriken gesund und hat seit Stunden nichts Sinnvolles bestätigt. Auf einem warteschlangenbasierten Broker bedienen die übrigen Worker weiterhin andere Nachrichten, sodass der Schaden klein aussieht. Auf einem logbasierten Topic steht eine Partition still, weil der Consumer immer wieder auf den fehlgeschlagenen Record zurückspringt, statt über ihn hinwegzugehen, und jede Bestellung auf dieser Partition hinter ihm wartet.

Irgendwann fällt es jemandem auf, und jemand schlägt vor, die Nachricht woandershin zu verschieben, damit die Arbeit weitergehen kann. Das ist die kurze Hälfte der Antwort. Die schwere Hälfte ist, was danach geschieht: wer sich diese Nachricht ansieht, wie er entscheidet, was sie bedeutet, und wie sie wieder ins System gelangt, ohne einen zweiten Incident auszulösen. Dieses Kapitel behandelt beide Hälften. Was sollte ein System mit einer Nachricht tun, die es nicht verarbeiten kann, und wie verhindert ein Team, dass diese Antwort zur Deponie wird?

Das Problem und warum es schwierig ist

Eine **Poison Message** ist eine Nachricht, die bei jeder Verarbeitung fehlschlägt. Das Wort „jeder“ trägt die Definition. Eine Nachricht, die fehlschlägt, weil eine Datenbank kurz nicht erreichbar war, ist kein Gift; sie gelingt, sobald die Datenbank zurück ist. Eine Poison Message schlägt fehl, weil etwas an der Nachricht oder am lesenden Code den Erfolg unmöglich macht, bis ein Mensch oder ein Deployment etwas ändert. In der Terminologie von Kapitel 3 besagt die Semantik des Ausfalls, dass ein weiterer Versuch nicht helfen kann, und genau deshalb ist ein Retry vergeudeter Aufwand.

Die üblichen Ursachen fallen in wenige Gruppen:

- **Fehlerhafter oder unerwarteter Inhalt.** Ein Pflichtfeld fehlt, ein Wert liegt außerhalb des Bereichs oder ein Payload lässt sich nicht parsen. Schemaänderungen, die mit einem alten Consumer nicht kompatibel waren, führen dazu (Kapitel 12 behandelt, wie man das vermeidet).
- **Ein Bug im Consumer.** Die Nachricht ist gültig, aber der Code behandelt einen Fall falsch, den niemand getestet hat, etwa ein null, wo nie eines erwartet wurde.
- **Eine verletzte fachliche Annahme.** Ein Ereignis `PaymentCaptured` erreicht einen Consumer, der keinen Eintrag zur zugehörigen Autorisierung hat, vielleicht weil Daten gelöscht wurden oder weil sich Ereignisse unterwegs überholt haben und die Retries inzwischen jedes vernünftige Fenster für eine Umordnung überdauert haben. Innerhalb dieses Fensters ist dasselbe Symptom vorübergehend, wie Kapitel 6 anmerkt und Kapitel 9 ausführlich behandelt, und gehört zu den Retries, nicht zur DLQ. Danach kann der Consumer nicht fortfahren und den fehlenden Eintrag nicht erfinden.
- **Eine dauerhafte Ablehnung durch ein nachgelagertes System.** Ein externer Paketdienst weist eine Adresse als ungültig zurück. Ein erneuter Versuch führt zur selben Ablehnung.

Kapitel 6 hat den Unterschied zwischen vorübergehenden und dauerhaften Ausfällen eingeführt, und dieses Kapitel setzt dort an, wo jenes endet. Eine korrekt arbeitende Retry-Richtlinie gibt bei einer Nachricht schließ-

lich auf. Retries sind durch Versuche, Zeit oder Budget begrenzt. Was an diesem Punkt geschieht, ist eine Designentscheidung, und wenn niemand sie trifft, trifft der Broker sie für Sie.

Die Endlosschleife der erneuten Zustellung

Tut der Consumer nichts Besonderes, greift das Standardverhalten des Brokers. Ein warteschlangenbasierter Broker stellt eine unbestätigte Nachricht erneut zu, entweder sofort nach einem Nack oder nach Ablauf des Visibility-Timeouts, und führt meist einen Zähler der erneuten Zustellungen, der mit jedem Versuch wächst.

Ein logbasierter Broker arbeitet anders, und der Unterschied zählt. Es gibt keine erneute Zustellung pro Nachricht: Der Consumer hält einen Offset und liest vorwärts. Wer einen Offset nicht per Commit festschreibt, bringt den Broker nicht dazu, den Record erneut zu liefern; die Position des Consumers im Speicher wandert weiter, und der fehlgeschlagene Record wird stillschweigend übersprungen, bis der nächste Neustart oder Rebalance den Consumer auf den letzten festgeschriebenen Offset zurücksetzt. Eine Schleife auf einem Log gibt es nur, weil der Consumer sie baut: Er springt auf den fehlgeschlagenen Offset zurück und liest ihn erneut, oder er hält die Partition an, während er an Ort und Stelle wiederholt. Ein Consumer, der keins von beiden tut und trotz des Fehlers über ihn hinweg festschreibt, verliert den Record ohne jedes Signal.

In beiden Familien kann das Ergebnis dasselbe sein: Die Nachricht kehrt zurück, scheitert und kehrt wieder zurück, oder, auf einem Log mit nachlässigem Offset-Handling, sie verschwindet. Die richtigen Werkzeuge der Log-Familie sind ein ausdrückliches Seek oder Pause, ein Retry-Topic, der den fehlgeschlagenen Record aufnimmt, damit die Hauptpartition weiterkommt, und ein Dead-Letter-Topic für Records, die man aufgegeben hat.

Die Schleife hat drei Kosten. Die erste ist vergeudete Kapazität: Worker verbringen ihre Zeit mit Arbeit, die nicht gelingen kann. Die zweite ist Rauschen: Logs füllen sich mit identischen Stack-Traces, und echte Probleme verstecken sich darunter. Die dritte ist die Kostenart, die am meisten zählt, **Head-of-Line-Blocking**. Auf einem Log verarbeitet ein Consumer eine Partition der Reihe nach, sodass ein Record, den er nicht überspringt, jeden Record dahinter blockiert. Auf einer einfachen Warteschlange blockiert eine einzelne Poison Message andere nicht von selbst, aber eine

Handvoll davon kann jeden Worker belegen. Gruppiert die Warteschlange Nachrichten für die Reihenfolge (FIFO-Warteschlangen, Message-Groups, Sessions), blockiert eine Poison Message ihre Gruppe, was denselben Auswirkungsbereich hat wie eine Log-Partition, nur schmaler.

Die zwei schlechten Auswege

Teams unter Druck greifen meist zu einem von zwei Auswegen, und beide sind aus demselben Grund falsch: Sie verstecken den Ausfall, statt ihn zu behandeln.

Der erste ist, **die Nachricht zu verwerfen**: die Exception abfangen, ins Log schreiben, bestätigen und weitermachen. Die Pipeline läuft wieder, aber das fachliche Ereignis ist weg. Bei `OrderPlaced` bedeutet das eine Bestellung, die in Orders existiert und in keinem anderen Service. Bei `PaymentCaptured` bedeutet es ein Hauptbuch, das nicht mit der Wirklichkeit übereinstimmt. Verwerfen ist nur dann die richtige Wahl, wenn die Nachricht keinen Wert trägt, und das kann ein Consumer im Moment des Fehlers nicht wissen.

Der zweite ist, **ewig zu wiederholen** und darauf zu vertrauen, dass jemand die Ursache behebt. Das ist die Endlosschleife von oben, und sie macht aus einer schlechten Nachricht einen Ausfall für alles, was dahinter liegt.

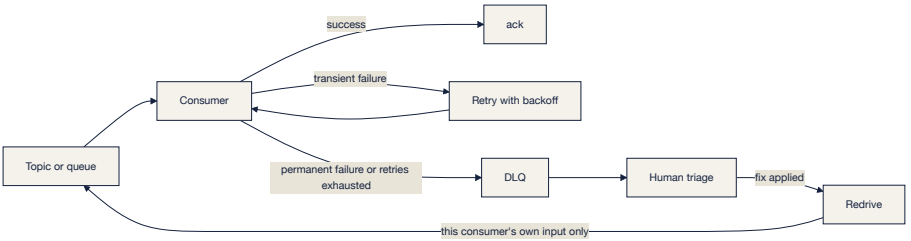
Die dritte Option liegt dazwischen. Die Nachricht beiseitelegen, mit genug Kontext, dass eine Person sie später verstehen kann, und den Rest des Streams weiterlaufen lassen. Diese Option ist die Dead-Letter-Queue.

Das Dead-Letter-Queue-Pattern

Eine **Dead-Letter-Queue (DLQ)** ist ein separates Ziel, in das ein Consumer oder ein Broker Nachrichten legt, die er aufgegeben hat. Auf einem logbasierten Broker ist dieselbe Idee meist ein **Dead-Letter-Topic**: ein gewöhnlicher Topic, der fehlgeschlagene Records aufnimmt. Wenn eine Aussage für beide Familien gilt, sagt dieses Kapitel DLQ.

Das Pattern hat einen einfachen Kern. Hat eine Nachricht ihre Retries ausgeschöpft oder schlägt sie auf eine Weise fehl, die der Consumer als dauerhaft einstuft, verschiebt der Consumer sie in die DLQ und bestätigt das

Original. Der Stream läuft weiter. Die fehlgeschlagene Nachricht geht nicht verloren; sie ist dort geparkt, wo man sie untersuchen kann.



Sehen Sie sich die letzten beiden Pfeile im Diagramm an. Die DLQ ist kein Ausgang. Sie ist ein Wartezimmer mit einer Tür zurück zum eigenen Input des fehlgeschlagenen Consumers, und die ganze Schwierigkeit dieses Kapitels liegt in der Entfernung zwischen diesen beiden Türen. Die Tür führt nicht zu einem gemeinsamen Topic: Andere Consumer haben das Ereignis bereits verarbeitet und sollen es nicht noch einmal sehen.

Wo das Verschieben stattfindet

Es gibt zwei Wege, das Verschieben umzusetzen, und die Wahl bestimmt, was Sie speichern können.

Vom Broker verwaltetes Dead-Lettering. Viele warteschlangenbasierte Broker lassen sich so konfigurieren, dass eine Nachricht, die öfter als eine bestimmte Anzahl von Malen zugestellt wurde, automatisch in eine festgelegte Warteschlange verschoben wird. Das ist mit wenig Aufwand eingerichtet und funktioniert auch, wenn der Consumer abstürzt, bevor er irgendetwas tun kann. Es liefert einen Zustellzähler, aber meist nicht den Grund, aus dem der Consumer gescheitert ist, weil der Broker ihn nicht kennt.

Vom Consumer verwaltetes Dead-Lettering. Der Consumer fängt den Fehler ab, baut einen Record, der den Grund enthält, veröffentlicht ihn in der DLQ und bestätigt erst dann das Original. So lassen sich Fehler klassifizieren, und ein dauerhafter Fehler wandert schon beim ersten Versuch in die DLQ statt beim fünften. Das kostet Code in jedem Consumer und muss sorgfältig geschrieben sein: erst in die DLQ veröffentlichen, danach bestätigen, sodass ein Absturz zwischen beiden ein Duplikat in der DLQ erzeugt, nie einen Verlust.

Ich würde beides einsetzen, wo der Broker das erste anbietet. Die vom Broker verwaltete Regel gibt es nur bei einigen warteschlangenbasierten Brokern, und nicht jeder Warteschlangentyp begrenzt die Zustellungen. Wo es sie gibt, ist sie das Auffangnetz für Consumer, die abstürzen oder auf eine Weise hängen, die Code nicht abfangen kann, und zusammen mit dem vom Consumer verwalteten Pfad sorgt sie dafür, dass eine Poison Message nicht ewig kreisen kann, auch wenn Ihr eigenes Handling einen Bug hat.

Logbasierte Broker haben im Allgemeinen weder einen Zustellzähler pro Nachricht noch eine DLQ auf Broker-Seite, und das hinterlässt eine Lücke. Eine Nachricht, die den Prozess zum Absturz bringt, etwa ein Deserialisierungsfehler vor dem Handler, ein Out-of-Memory-Fehler oder ein nativer Absturz, erreicht Ihren Block `try` nie. Der Consumer startet am festgeschriebenen Offset neu, liest denselben Record und stürzt erneut ab. Kapitel 6 hat festgehalten, dass der Versuchszähler nach einem Absturz von vorn beginnt. Bei einem Poison Record ist das das ganze Problem.

Die Abhilfe ist, vor der Verarbeitung zu zählen, damit der Zähler den Absturz überlebt:

```
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)
```

Der Zähler ist nach Partition und Offset geschlüsselt (oder nach `event_id`, wenn Sie sie lesen können, ohne zu deserialisieren). Der Preis ist ein dauerhafter Schreibvorgang pro Record im Hot Path und ein Limit, das hoch genug sein muss, damit ein gewöhnlicher Neustart durch ein Deployment gesunde Records nicht darüber hebt. Die Alternative ist, hinzunehmen,

dass ein Poison Record in der Absturzschleife seine Partition blockiert, bis jemand es bemerkt, und deshalb ist der weiter unten beschriebene Lag-Alarm wichtig.

```
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)
```

Im Schritt `deadLetter` wird der Kontext erfasst, den die Triage braucht, und die Reihenfolge seiner letzten beiden Zeilen ist der springende Punkt.

```
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)
```

Schlägt `publish` selbst fehl, löst `deadLetter` eine Exception aus, und die Nachricht wird nicht bestätigt. Bei einem Broker aus der Familie der Warteschlangen genügt das: Die Nachricht kommt über die normale erneute Zustellung zurück. Bei einem Broker aus der Familie der Logs genügt es nicht, weil ein nicht festgeschriebener Offset den Record nicht zurückbringt; der Consumer muss auf die Position des fehlgeschlagenen Records zurückspringen oder die Partition pausieren und die Veröffentlichung in die DLQ wiederholen, bevor er weiterliest. In beiden Fällen ist der Fehlermodus derselbe: Die Nachricht bleibt im System, bis sie ein sicheres neues Zuhause hat.

Was zu einem Dead Letter gespeichert wird

Ein Dead Letter ohne Kontext ist ein Rätsel. Ein Team, das sechs Wochen später eine Nachricht in einer DLQ findet, muss vier Fragen beantworten können: Was war das, woher kam es, warum ist es fehlgeschlagen, und ist es noch sicher zu verarbeiten? Der Record sollte sie beantworten, ohne dass jemand ein Quell-Repository öffnet.

Feld	Warum es da ist
Originaler Body und Header, Byte für Byte	Ein Redrive muss die Nachricht exakt reproduzieren, einschließlich `event_id`
Name und Position der Quelle (Topic und Partition und Offset, oder Warteschlangenname)	Verortet die Nachricht in der Historie, stützt Entscheidungen zur Reihenfolge
Originaler Schlüssel (Record-Key oder Routing-Key) und originaler Zeitstempel	Ein Redrive muss mit demselben Schlüssel veröffentlichen, sonst landet die Nachricht auf einer anderen Partition und verliert ihre Reihenfolge pro Schlüssel
Consumer-Input (die eigene Warteschlange oder der Redrive-Topic des fehlgeschlagenen Consumers)	Der einzige Ort, an dem ein Redrive veröffentlichen darf
`redrive_count`	Getrennt von `attempt_count`: wie oft diese Nachricht schon zurückgekommen ist
`error_class` und `error_message`	Gruppiert Fehler nach Ursache; lange Meldungen kürzen
Stack-Trace oder ein Verweis auf die Log-Zeile	Lässt einen Entwickler sehen, wo es fehlschlug, ohne es zu reproduzieren
`attempt_count`	Unterscheidet „einmal fehlgeschlagen, dauerhaft“ von „nach vielen Retries fehlgeschlagen“
`first_failed_at` und `failed_at`	Zeigt, wie lange die Nachricht schon wartet und wann das Problem begann
Name und Version des Consumers	Bindet den Fehler an ein Deployment; oft der schnellste Weg zur Ursache
`correlation_id`	Verbindet die Nachricht mit dem Rest ihres fachlichen Ablaufs

Zwei Warnungen gelten. Erstens: Lassen Sie den originalen Payload unverändert. Wenn Sie ihn in einen eigenen Umschlag packen, muss das Redrive-Werkzeug ihn exakt wieder auspacken, sonst driften `event_id` und `causation_id` auseinander, und die Idempotenzgarantien aus Kapitel 7 gehen verloren. Zweitens: Bedenken Sie, dass eine DLQ eine Kopie von Produktionsdaten ist, oft lange aufbewahrt. Sie erbt die Zugriffskontrollen und Aufbewahrungspflichten des Originals, und Payloads mit personenbezogenen Daten brauchen in der DLQ dieselbe Sorgfalt wie überall sonst.

Parking Lot gegenüber DLQ

Das Wort DLQ wird für zwei verschiedene Dinge benutzt, und sie zu trennen macht den Betrieb leichter durchschaubar.

Ein **Parking Lot** ist ein Ablageort für Dead Letters, die eine menschliche Entscheidung brauchen. Die DLQ ist der Ort, an den das System eine Nachricht automatisch legt; der Parking Lot ist der Ort, an den eine Person eine Nachricht bewusst legt, nachdem sie sie angesehen und entschieden hat, dass ein Redrive sie nicht reparieren wird. Es ist ein eigener Ort mit einem eigenen Zweck.

Überlegen Sie, was jeder von beiden enthält. Die DLQ enthält Nachrichten, die fehlgeschlagen und noch nicht untersucht sind. Sie sollte sich leeren: Ihr erwarteter Zustand ist leer oder fast leer, und alles Alte darin ist ein Symptom. Der Parking Lot enthält Nachrichten, die jemand untersucht und beiseitegelegt hat: eine Bestellung, die storniert wurde, bevor ihr Ereignis eintraf, einen Payload, der manuell korrigiert werden muss, einen Record, der von Hand mit einem anderen Team abgeglichen werden muss. Diese Nachrichten warten nicht darauf, dass eine Korrektur ausgerollt wird. Sie warten auf eine Entscheidung.

Beides zu vermischen führt zu einem vorhersehbaren Fehler. Bleiben entschiedene und geparkte Nachrichten in der DLQ, kehrt die Tiefe der DLQ nie auf null zurück, der Schwellenwert des Alarms wird angehoben, um das Rauschen abzustellen, und die nächste echte Poison Message kommt unbemerkt an. Hält man sie getrennt, bleibt der DLQ-Alarm scharf.

Ein Parking Lot kann so bescheiden sein wie eine Datenbanktabelle mit Status, Verantwortlichem, Grund und Review-Datum. Er muss überhaupt kein Ziel auf dem Broker sein, denn nichts wird ihn automatisch konsumieren. Das Review-Datum zählt am meisten: Eine geparkte Nachricht ohne Termin für den nächsten Blick ist bereits vergessen.

Verantwortung und Alarme

Eine DLQ ohne Verantwortlichen ist ein Abfluss, der sich unbemerkt füllt. Die erste Entscheidung, noch vor jedem Werkzeug, ist, wer für jede einzelne verantwortlich ist.

Meine Voreinstellung ist, dass das Team, das den Consumer verantwortet, auch die DLQ verantwortet. Es hat den Code geschrieben, der fehlgeschlagen ist, es weiß, was die Nachricht bedeuten sollte, und es kann korrigieren und ausrollen. Eine DLQ pro Consumer, oder pro Consumer-Group und Quelle, macht diese Verantwortung offensichtlich. Eine einzige gemeinsame DLQ für das ganze Unternehmen macht sie mehrdeutig, und Mehrdeutigkeit heißt, dass niemand hinsieht. Stellt ein Plattform-Team das Werkzeug bereit, verantwortet das Plattform-Team den Mechanismus und das konsumierende Team den Inhalt.

Worauf alarmiert wird

Alarmieren Sie auf **Eintreffen**, nicht nur auf Tiefe. Eine einzelne Nachricht in einer DLQ, die normalerweise leer ist, ist eine Benachrichtigung wert, denn der erste Dead Letter ist der günstigste Moment, einen Bug zu finden. Ein Tiefenschwellenwert allein sagt nichts, bis die Warteschlange schon tief ist.

Drei Signale zusammen decken die meisten Fälle ab:

- **Jedes neue Eintreffen** in einer DLQ, die leer sein soll. Leiten Sie dies an einen Kanal, den während der Arbeitszeit ein Mensch liest.
- **Eintreffrate.** Ein plötzlicher Schub bedeutet meist ein fehlerhaftes Deployment oder eine inkompatible Schemaänderung und rechtfertigt es, jemanden zu alarmieren.
- **Alter der ältesten Nachricht.** Das fängt den schleichenden Ausfall ab: die DLQ, die eine Nachricht pro Tag erhält, auf die niemand reagiert, bis die älteste einen Monat alt ist und der fachliche Kontext verloren ist.

Ich kenne den richtigen Alarmierungsschwellenwert für Ihr System nicht, und niemand kann ihn Ihnen nennen, ohne den fachlichen Wert der betroffenen Nachrichten zu kennen. Eine DLQ für die Consumer von `PaymentCaptured` verdient einen niedrigeren Schwellenwert als eine für Notifications, denn eine verlorene E-Mail ist ein Ärgernis und eine verlorene Erfassung eine Abweichung im Hauptbuch. Die Schwellenwerte sollten diesen Unterschied abbilden, und der Unterschied sollte neben dem Alarm festgehalten sein.

> **Hinweis:** Alarmieren Sie auch auf den Consumer-Lag. Ein Consumer, der in einer Log-Partition an einem Poison Record hängt, erzeugt nicht zwingend Dead Letters, weil er nie aufgibt. Steigender Lag bei ausbleibendem Durchsatz ist das Signal, dass die zu Beginn dieses Kapitels beschriebene Schleife läuft.

Redrive: Nachrichten zurückholen

Redrive bedeutet, Dead Letters zur erneuten Verarbeitung an den Consumer zurückzugeben, der an ihnen gescheitert ist. Das Ziel ist der eigene Input dieses Consumers: sein Retry- oder Redrive-Topic oder seine eigene Warteschlange. Es ist nie der gemeinsame Topic, aus dem die Nachricht ursprünglich kam, denn jeder andere Consumer dieses Topics hat das Ereignis bereits verarbeitet, und ein erneutes Veröffentlichen dort würde es allen ein zweites Mal zustellen. Der originale Schlüssel reist mit der Nachricht, sodass sie auf derselben Partition bleibt und ihre Position pro Schlüssel behält. Der Versuchszähler beginnt wieder bei null, und der separate `redrive_count` steigt um eins, sodass eine Nachricht, die immer wieder zurückkommt, sichtbar wird, statt jedes Mal neu zu wirken. Der Redrive ist die Operation, die eine DLQ wertvoll macht, und zugleich die, die am ehesten einen zweiten Incident auslöst. Kapitel 3 beschrieb den Entwickler, der einen Knopf drückt und 40.000 Nachrichten auf einmal per Redrive zurückgibt. Die Abhilfe ist, den Redrive als entworfenes Verfahren zu behandeln, nicht als Komfortknopf.

Die Ursache vor dem Redrive beheben

Eine Nachricht, die wegen eines Consumer-Bugs fehlgeschlagen ist, schlägt wieder fehl, bis der Bug behoben und ausgerollt ist. Ein Redrive vor der Korrektur bewegt die Nachrichten im Kreis und füllt die DLQ mit denselben Einträgen neu, nun mit höheren Versuchszählern und einer verwirrenden Historie. Die Reihenfolge der Arbeit ist: die Ursache verstehen, die Korrektur ausrollen, eine kleine Stichprobe per Redrive zurückgeben, beobachten, dass sie gelingt, dann den Rest per Redrive zurückgeben.

Ein Redrive muss sich gefahrlos wiederholen lassen

Eine per Redrive zurückgegebene Nachricht ist per Definition ein Duplikat: Das System hat diese `event_id` schon gesehen, und der Consumer womöglich auch. Hat die ursprüngliche Verarbeitung vor dem Fehler teilweise Erfolg gehabt, existieren manche Effekte vielleicht schon. Kapitel 7 ist der Grund, warum das überlebbar ist. Ein idempotenter Consumer mit einer Inbox behandelt die per Redrive zurückgegebene Nachricht wie jede andere erneute Zustellung und wendet ihre Effekte einmal an.

Zwei Punkte sind festzuhalten. Ein Redrive muss die originale `event_id` bewahren, denn sie ist der Idempotenzschlüssel. Ein Redrive-Werkzeug, das für jede Nachricht eine neue Kennung erzeugt, unterläuft die Inbox, und der Effekt kann doppelt angewendet werden. Und ein Consumer, der nicht idempotent ist, sollte gar nicht das Ziel eines Massen-Redrives sein; beheben Sie zuerst das, oder akzeptieren Sie, dass jede per Redrive zurückgegebene Nachricht eine manuelle Risikobewertung ist.

Redrive in kontrollierten Batches

Ein sicheres Redrive-Werkzeug hat einige Eigenschaften, keine davon exotisch:

1. **Einen Auswahlfilter.** Redrive nach Fehlerklasse, Zeitfenster oder `correlation_id`, nicht nur „alles“.
2. **Ein Rate Limit.** Nachrichten kehren in einem Tempo zum Input des Consumers zurück, das er und seine Abhängigkeiten verkraften. Eine Flut von Redrive-Nachrichten ist ein selbst verursachter Retry-Storm, und die Limits für Nachrichten in Bearbeitung und das Load Shedding aus „Retries unter Überlast: Backpressure“ (Kapitel 6) gelten für Redrive-Verkehr genauso wie für Live-Verkehr.
3. **Einen Pausenschalter.** Ein Betreiber kann den Redrive mittendrin anhalten, wenn Fehler wieder auftreten.
4. **Einen Dry Run.** Das Werkzeug meldet, was es verschieben würde, bevor es irgendetwas verschiebt.
5. **Einen Audit-Record.** Wer was wann und warum per Redrive zurückgegeben hat, dauerhaft festgehalten.

```
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"
```

Beachten Sie die Reihenfolge innerhalb der Schleife: Ein Record wird als per Redrive zurückgegeben markiert, nachdem die Veröffentlichung gelungen ist, nicht davor. Stürzt das Werkzeug mittendrin ab, sind einige Nachrichten per Redrive zurückgegeben und in der DLQ dennoch unmarkiert, was zu Duplikaten führt, nicht zu Verlust. Ein Consumer mit Inbox fängt diese Duplikate ab; einer ohne Inbox macht aus einem Absturz des Werkzeugs ein Datenproblem. Der Record wird markiert, nicht gelöscht, damit der Prüfpfad und `redrive_count` erhalten bleiben; die Bereinigung folgt der Aufbewahrungsregel in „Die Deponie verhindern“ weiter unten.

Redrive und Reihenfolge

Ein Redrive bricht die Reihenfolge, wenn Sie nicht dafür entwerfen. Eine Dead-Letter-Nachricht wurde übersprungen; spätere Nachrichten für dasselbe Aggregat wurden inzwischen verarbeitet. Kommt die per Redrive zurückgegebene Nachricht an, kommt sie nach ihnen an. Das ist das Inventory-Beispiel aus Kapitel 6 mit größerem Abstand: Wird `OrderPlaced` statt per Retry in die DLQ verschoben, findet `OrderCancelled` weiterhin keine Reservierung zum Freigeben, und die per Redrive zurückgegebene Platzie-

rung legt Stunden später weiterhin eine an, die niemand freigeben wird. Den Schlüssel zu bewahren behebt das nicht: Der Schlüssel bringt die Nachricht auf die richtige Partition, aber die Partition ist weitergezogen.

Ob das eine Rolle spielt, hängt vom Consumer ab. Ein Consumer, der **kommutativ** ist, bei dem also der Endzustand unabhängig von der Eintreffreihenfolge derselbe ist, ist nicht betroffen. Ein Consumer, der von Zustandsübergängen abhängt, ist es. Kapitel 9 behandelt, wie man Eintreffen in falscher Reihenfolge mit Versionsnummern oder Sequenzprüfungen erkennt und behandelt, und ein zustandsbasierter Consumer sollte sie nutzen, bevor jemand per Redrive in ihn hineinzurückgibt. Tut er das nicht, muss das Redrive-Verfahren eine Prüfung enthalten, dass die Welt seit dem Verschieben der Nachricht in die DLQ nicht weitergezogen ist.

Was ein Dead Letter mit einem geordneten Stream macht

Geordnete Streams erzwingen die schwerste Entscheidung dieses Kapitels, denn die Wahl, die die Reihenfolge schützt, ist die, die das Geschäft anhält.

Betrachten Sie einen logbasierten Topic, geschlüsselt nach `order_id`, in dem Ledger Buchungen vornimmt. Consumer verarbeiten jede Partition der Reihe nach. Nachricht 41 für `order-81724` schlägt dauerhaft fehl. Dem Consumer bleiben zwei Optionen:

- **Anhalten und warten.** Die Partition bei Nachricht 41 festhalten, bis jemand die Ursache behebt. Die Reihenfolge bleibt exakt erhalten. Jede andere Bestellung, die per Hash dieser Partition zugeordnet ist, steht ebenfalls still, obwohl ihre Nachrichten in Ordnung sind. Der Auswirkungsbereich ist die ganze Partition.
- **In die DLQ verschieben und weitermachen.** Nachricht 41 parken, 42 und alles danach verarbeiten. Die Partition fließt. Aber 42 kann von 41 abhängen, und der Consumer hat sie dann gegen einen Zustand verarbeitet, der 41 nicht enthält.

Keine der beiden Wahlen ist umsonst. Anhalten tauscht Verfügbarkeit gegen Korrektheit; Weitermachen tauscht Korrektheit gegen Verfügbarkeit. Keine Einstellung liefert beides, und es ist besser, pro Consumer bewusst

zu wählen, als zu übernehmen, was der Framework-Standard gerade vorgibt.

Meine Faustregel hat drei Teile:

1. **Ist der Consumer unempfindlich gegen die Reihenfolge oder kommutativ**, in die DLQ verschieben und weitermachen. Notifications ist der typische Fall: Eine fehlgeschlagene E-Mail zu einer Bestellung hat keinen Einfluss auf die nächste.
2. **Hängen spätere Nachrichten für denselben Schlüssel von der fehlgeschlagenen ab**, für diesen Schlüssel nicht weitermachen. Die fehlgeschlagene Nachricht in die DLQ verschieben und zusätzlich die folgenden Nachrichten für denselben `partition_key` in einen Haltebereich umleiten, bis die erste geklärt ist. Andere Schlüssel laufen weiter. So bleibt der Auswirkungsbereich bei einem Aggregat statt bei einer Partition, zum Preis eines zusätzlichen Mechanismus: der Nachverfolgung „blockierter“ Schlüssel pro Schlüssel.
3. **Ist der Consumer ein Finanz-Hauptbuch, anhalten.** Die Kosten einer Verarbeitung außer der Reihe sind ein Record, der auf eine schwer erkennbare und teuer zu reparierende Weise falsch ist, und eine pausierte Partition ist laut und offensichtlich. Korrektheit vor Geschwindigkeit war die erklärte Regel für Ledger, und hier gilt sie in ihrer schlichtesten Form.

Die Variante pro Schlüssel in der zweiten Regel verdient eine Skizze, da sie die nützlichste und seltenste ist. Sie setzt eine kleine Tabelle `key_block` mit einer Zeile pro blockiertem Schlüssel (`partition_key`, `blocked_event_id`, `state`) voraus und dass jeweils ein Consumer den Live-Verkehr eines bestimmten Schlüssels bearbeitet, was eine Partition oder eine Message-Group gewährleistet.

```
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)
```

Die Freigabe ist ein separates Verfahren, und ein periodischer Sweep macht sie wiederanlauffähig.

```
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)
```

Drei Details dieser Skizze entscheiden, ob sie funktioniert. Das erste ist der Redrive selbst. Die per Redrive zurückgegebene Nachricht ist genau die, auf die der Schlüssel blockiert ist, also muss sie die Blockade umgehen: Die Prüfung vergleicht ihre `event_id` mit der blockierten. Ohne diese Ausnahme wird der Redrive hinter den Nachrichten geparkt, denen er vorausgehen sollte, und der Schlüssel wird nie freigegeben.

Das zweite ist das Race mit dem Live-Verkehr. Während `resolveBlockedKey` die gehaltenen Nachrichten abarbeitet, treffen weiter neue Nachrichten für denselben Schlüssel ein. Sie müssen weiterhin gehalten und nicht verarbeitet werden, bis die gehaltene Menge leer ist. Das gilt nur, wenn die Entscheidung zu parken und die Entscheidung freizugeben serialisiert sind: Beide nehmen die Zeilensperre des Schlüssels, und das Parken wird in derselben Transaktion wie die Prüfung geschrieben. Wäre

das Parken ein separater Schritt, könnte eine Live-Nachricht einen blockierten Schlüssel sehen, dann die gehaltene Menge bereits geleert und freigegeben vorfinden und hinter einer Blockade geparkt werden, die es nicht mehr gibt. Niemand würde sie abarbeiten, und spätere Nachrichten würden vor ihr verarbeitet.

Das dritte ist ein Absturz mitten in der Freigabe. Sobald die per Redrive zurückgegebene Nachricht verarbeitet ist, wird der Zustand `resolving` vor dem Ack geschrieben, sodass die noch zu erledigende Arbeit in der Datenbank festgehalten ist und nicht nur in einer Nachricht, die nie wiederkommt. Der Sweep nimmt jeden Schlüssel auf, der in diesem Zustand geblieben ist, und macht weiter; eine kurz vor dem Absturz verarbeitete Nachricht fängt die Inbox ab. Schlägt eine gehaltene Nachricht während der Freigabe dauerhaft fehl, wird der Schlüssel erneut blockiert, verankert an dieser Nachricht und mit einem eigenen Dead Letter, und der Rest der gehaltenen Menge wartet auf dessen Redrive. Die Blockade pro Schlüssel ist eine Schleife, kein einmaliges Ereignis.

Die Skizze ist nicht vollständig. Ein Sweep kann sich mit einer laufenden Live-Freigabe überschneiden, sodass zwei Resolver am selben Schlüssel arbeiten. Die Zustandsprüfung unter der Sperre verhindert, dass einer von ihnen eine erneute Blockade überholt: Sobald ein Resolver die Blockade an einer fehlgeschlagenen Nachricht verankert, findet der andere den Schlüssel nicht mehr im Zustand `resolving` vor und hört auf. Übrig bleibt doppelte Arbeit an derselben gehaltenen Nachricht, die die Inbox abfängt; ein Lease auf den Zustand `resolving` würde diese Arbeit sparen, nicht die Korrektheit. Sie setzt außerdem der gehaltenen Menge keine Grenze, und sie lässt offen, wie ein Betreiber eine gehaltene Nachricht verwirft, die nicht mehr aussagekräftig ist. Verstehen Sie sie als die Form des Mechanismus, nicht als einen fertigen.

Das ist echte Maschinerie mit eigenen echten Fehlermodi, etwa einem unbegrenzt wachsenden Speicher für gehaltene Nachrichten, und deshalb würde ich sie nur für Streams bauen, bei denen die Reihenfolge pro Schlüssel zählt und die Dead-Letter-Rate niedrig genug ist, dass die gehaltene Menge klein bleibt. Es ist auch ein Mechanismus, der nicht weiß, ob die gehaltenen Nachrichten noch aussagekräftig sind, sobald die fehlgeschlagene geklärt ist. Müssen die Zustandsübergänge selbst validiert und nicht bloß in eine Reihenfolge gebracht werden, ist der Versionspuffer aus

Kapitel 9 das stärkere Werkzeug: Er hält ein Ereignis zurück, bis sein Vorgänger tatsächlich angewendet wurde, nach Versionsnummer, und er funktioniert ohne einen Dead-Letter-Record als Anker.

Die Deponie verhindern

Eine DLQ wird zur Deponie, wenn Nachrichten schneller hineinkommen, als jemand entscheidet, was sie bedeuten. Das Symptom ist vertraut: tausende Einträge, viele verschiedene Fehler, niemand sicher, welche zählen, und eine ungeschriebene Regel, dass man sie nicht anfasst. An diesem Punkt ist die DLQ kein Sicherheitsmechanismus mehr; sie ist ein Ort, an dem verlorene Daten unter einem höflichen Namen gelagert werden.

Deponien haben Ursachen, und jede hat ein Gegenmittel.

- **Kein Verantwortlicher.** Das Gegenmittel ist die oben genannte Verantwortungsregel, aufgeschrieben.
- **Kein Triage-Rhythmus.** Dead Letters sollten nach einem Plan geprüft werden, nicht nur, wenn ein Alarm auslöst. Für eine DLQ mit geringem Volumen ist eine wöchentliche Prüfung ein vernünftiger Anfang.
- **Aufbewahrung, die den Nutzen überdauert, oder die vor der Triage abläuft.** Ein Dead Letter zu einer Bestandsreservierung, die letzten Monat verfallen ist, lässt sich nicht sinnvoll per Redrive zurückgeben. Legen Sie eine Aufbewahrungsdauer fest, die sich daran orientiert, wie lange die Nachricht fachlich gültig bleibt, und entscheiden Sie im Voraus, was bei Ablauf geschieht: archivieren, mit Audit-Eintrag verwerfen oder eskalieren. Der umgekehrte Fehler ist ebenso häufig: Eine Aufbewahrung, die kürzer ist als der Triage-Rhythmus, löscht Nachrichten, bevor jemand sie angesehen hat. Ist die DLQ eine Broker-Warteschlange, deren Aufbewahrung kürzer ist als Ihr Prüfintervall, entfernt der Broker die Beweise zuerst. Setzen Sie die Aufbewahrung der DLQ über die längste Pause zwischen Prüfungen, und lesen Sie in der Dokumentation Ihrer Plattform nach, ab wann die Aufbewahrungsuhr läuft. Beginnt sie mit dem ursprünglichen Zeitpunkt des Einreihens und nicht mit dem Moment des Verschiebens in die DLQ, kommt eine Nachricht, die vor dem Fehlschlag lange gewartet hat, schon kurz vor dem Ablauf an. Das ist ein weiterer Grund, warum der Record den originalen Zeitstempel speichert.

- **Alles landet in der DLQ.** Werden routinemäßige fachliche Ergebnisse, etwa eine Ablehnung bei der Validierung, die normaler Teil des Ablaufs ist, an die DLQ geschickt, gehen die echten Fehler unter. Eine abgelehnte Bestellung ist ein fachliches Ereignis mit eigener Behandlung, kein Dead Letter.
- **Vorübergehende Fehler werden zu früh in die DLQ verschoben.** Gibt die Retry-Richtlinie bei einem kurzen Ausfall nach Sekunden auf, landet jede Nachricht in Bearbeitung während dieses Ausfalls in der DLQ. Die Abhilfe steht in Kapitel 6: eine Retry-Richtlinie, deren Versuche und Deadline auf den Ausfall zugeschnitten sind, den man überstehen will.

Außerdem würde ich jedem Dead Letter einen Ausgang geben. Die Triage endet mit einem von wenigen Ergebnissen: nach einer Korrektur per Redrive zurückgegeben, korrigiert und per Redrive zurückgegeben, für eine manuelle Entscheidung geparkt oder mit festgehaltenem Grund verworfen. „Liegt noch da“ ist kein Ausgang.

Abwägungen und Kosten

Eine DLQ verlagert einen Ausfall von einem Ort, an dem er laut ist, dem blockierten Consumer, an einen Ort, an dem er leise ist. Das ist zugleich der Nutzen und die Gefahr. Der Stream fließt, und die fehlgeschlagene Nachricht kann vergessen werden. Alles in diesem Kapitel, was nicht die DLQ selbst ist, die Alarme, der Verantwortliche, der Prüfrhythmus und die Aufbewahrung, existiert, um dieser Stille entgegenzuwirken.

Konkrete Kosten, die abzuwägen sind:

- **Betriebslast.** Jemand muss die Triage machen, entscheiden und den Redrive ausführen. Eine DLQ, für die niemand Zeit hat, ist schlimmer als keine, weil sie das Problem hinter dem Anschein eines Mechanismus versteckt.
- **Daten-Governance.** Dead Letters enthalten Produktions-Payloads, manchmal mit personenbezogenen Daten, außerhalb des üblichen Lebenszyklus. Sie brauchen Zugriffskontrollen, Aufbewahrungsregeln und Löschwege.
- **Risiko für die Reihenfolge.** Eine Nachricht aus einem geordneten Stream in die DLQ zu verschieben, überspringt sie. Solange Sie nicht

eine Blockade pro Schlüssel bauen oder die Partition anhalten, werden spätere Ereignisse ohne sie angewendet. Die Blockade pro Schlüssel bringt selbst einen Speicher für gehaltene Nachrichten, eine Redrive-Umgehung, eine Freigabeschleife, die sicher mit dem Live-Verkehr konkurrieren muss, und einen Sweep, der eine unterbrochene Freigabe neu startet.

- **Gespeicherte Identität.** Ein sicherer Redrive braucht in jedem Record den originalen Schlüssel, die Partition und den Consumer-Input und neben dem Versuchszähler einen `redrive_count`. Das ist mehr zu speichern und konsistent zu halten, und ohne es kann ein Redrive nur ein Raten sein.
- **Investition in Werkzeuge.** Ein sicheres Redrive-Werkzeug mit Filterung, Rate Limits, Ausgang pro Record und Prüfpfad ist ein kleines Projekt. Es zu kaufen oder zu bauen gehört zu den Kosten einer DLQ.

Wann man keine einsetzt: Lässt sich eine Nachricht neu erzeugen oder ist ihr Verlust harmlos, bringt eine DLQ mehr Prozess als Wert. Ein Ereignis zum Aufwärmen des Caches oder eine rein kosmetische Aktualisierung kann verworfen werden, mit einer Metrik, die die Verwürfe zählt. Und wo die richtige Reaktion auf einen Ausfall das Anhalten ist, wie im Hauptbuch-Beispiel, arbeitet eine DLQ, die den Stream weiterlaufen lässt, gegen Sie. Halten Sie in diesem Fall den Consumer an, alarmieren Sie laut und beheben Sie die Ursache.

Auch dem Pattern sind Grenzen gesetzt. Eine DLQ enthält Nachrichten, die sichtbar fehlgeschlagen sind. Für Nachrichten, die fehlerhaft verarbeitet wurden, ohne dass ein Fehler auftrat, leistet sie nichts; das ist das Gebiet von Kapitel 13 und seiner Abgleichsarbeit.

> **Warnung:** Bestätigen Sie die Originalnachricht nie, bevor der Dead Letter dauerhaft gespeichert ist. Schlägt die Veröffentlichung in die DLQ nach dem Ack fehl, existiert die Nachricht nirgends, und kein Alarm sagt es Ihnen.

Häufige Fehler

- **Dead-Lettering ohne Kontext.** Es ist verlockend, weil der rohe Body zu verschieben der kürzeste Code ist. Die Lösung ist der oben be-

schriebene Metadaten-Record, einschließlich der exakten originalen Header, damit sowohl Redrive als auch Diagnose möglich sind.

- **Dauerhafte Ausfälle wiederholen.** Ein Parse-Fehler wird mit der Zeit nicht besser, doch eine pauschale Retry-Richtlinie behandelt ihn wie einen Timeout. Klassifizieren Sie Ausfälle und schicken Sie dauerhafte direkt in die DLQ, statt das Retry-Budget zu verbrauchen.
- **Redrive vor der Behebung der Ursache.** Der Druck, „die Nachrichten in Bewegung zu bringen", ist real, und das Ergebnis sind dieselben Nachrichten wieder in der DLQ. Rollen Sie die Korrektur aus, geben Sie eine Stichprobe per Redrive zurück, dann den Rest.
- **Alles auf einmal per Redrive zurückgeben.** Eine Flut alter Nachrichten konkurriert mit dem Live-Verkehr und kann einen neuen Incident auslösen. Begrenzen Sie die Rate, arbeiten Sie in Batches und halten Sie einen Pausenschalter bereit.
- **Nur auf die Tiefe alarmieren.** Sie wirkt wie die naheliegende Metrik, löst aber spät aus. Alarmieren Sie auf das erste Eintreffen, auf die Rate und auf das Alter der ältesten Nachricht.
- **Eine gemeinsame DLQ für alles.** Sie ist billig zu erstellen und unmöglich zu verantworten. Geben Sie jedem Consumer eine eigene, und bestimmen Sie den Verantwortlichen namentlich.
- **Geparkte und ununtersuchte Nachrichten vermischen lassen.** Die DLQ leert sich nie, der Alarm wird heruntergestellt, und echtes Gift bleibt unbemerkt. Führen Sie einen separaten Parking Lot mit Verantwortlichen und Review-Terminen.
- **Die Reihenfolge beim Redrive ignorieren.** Eine per Redrive zurückgegebene Nachricht kommt nach späteren Ereignissen für dasselbe Aggregat an. Nutzen Sie die Versionsprüfungen aus Kapitel 9, oder vergewissern Sie sich, dass der Zustand nicht weitergezogen ist.
- **Redrive in den gemeinsamen Topic.** Es sieht aus wie „die Nachricht dorthin zurücklegen, woher sie kam", und es stellt das Ereignis jedem anderen Consumer dieses Topics ein zweites Mal zu. Geben Sie per Redrive an den eigenen Input des fehlgeschlagenen Consumers zurück, mit dem originalen Schlüssel.
- **Ewig auf einen Record in der Absturzschleife warten.** Auf einem logbasierten Broker erzeugt ein Consumer, der nie aufgibt, keine Dead Letters und keinen Alarm außer dem Lag. Zählen Sie Versuche dauer-

haft nach Partition und Offset, damit ein Neustart den Zähler nicht zurücksetzt.

Zentrale Erkenntnisse

- Eine Poison Message schlägt bei jeder Verarbeitung fehl, sodass ewiges Wiederholen aus einer schlechten Nachricht einen Ausfall für alles macht, was dahinter in der Schlange steht.
- Eine fehlgeschlagene Nachricht zu verwerfen und sie ewig zu wiederholen sind beides Wege, einen Ausfall zu verstecken; eine Dead-Letter-Queue legt die Nachricht mit Kontext beiseite, damit der Rest des Streams weiterlaufen kann.
- Ein Dead Letter sollte den originalen Body und die Header unverändert, seine Quellposition, den Fehler, den Versuchszähler und die Consumer-Version tragen, denn wer ihn Wochen später liest, hat nur diesen Record.
- Veröffentlichen Sie in der DLQ, bevor Sie das Original bestätigen, damit ein Absturz ein behebbares Duplikat erzeugt und nie eine verlorene Nachricht.
- Jede DLQ braucht ein benanntes verantwortliches Team, einen Alarm auf erstes Eintreffen, Rate und Alter und einen regelmäßigen Triage-Rhythmus, sonst wird sie zur Deponie.
- Ein Redrive ist nur sicher, wenn die Ursache behoben ist, die originale `event_id` und der originale Schlüssel erhalten bleiben, die Nachricht an den eigenen Input des fehlgeschlagenen Consumers geht und nie an einen gemeinsamen Topic, der Consumer idempotent ist und die Nachrichten in ratenbegrenzten Batches mit einem Pausenschalter zurückkehren.
- Auf einem geordneten Stream erzwingt das Verschieben einer Nachricht in die DLQ die Wahl zwischen dem Anhalten der Partition und dem Weitermachen über eine Lücke hinweg; wählen Sie pro Consumer bewusst, halten Sie bei Hauptbüchern an und blockieren Sie pro Schlüssel nur dort, wo die Maschinerie ihre Kosten wert ist.

Dead-Lettering und Redrive stützen sich beide auf eine Frage, die dieses Kapitel immer wieder aufgeschoben hat: Was bedeutet es, dass Nachrichten in der richtigen Reihenfolge eintreffen, und was lässt sich bauen, wenn

sie es nicht tun? Kapitel 9 nimmt sich dieser Frage direkt an.

Kapitel 9 — Nachrichtenreihenfolge

Stellen Sie sich einen Donnerstagabend bei Parcelly vor. Ein Käufer gibt eine Bestellung auf, überlegt es sich binnen Sekunden anders und storniert sie. Orders sendet `OrderPlaced`, dann `OrderConfirmed`, dann `OrderCancelled`, alle für `order-81724` und alle in der richtigen Reihenfolge. Auch der Relay veröffentlicht sie in dieser Reihenfolge.

Der Topic, der Shipping speist, wurde mit einer Standardeinstellung angelegt: Nachrichten werden ohne Rücksicht auf ihren Inhalt auf sechs Partitionen verteilt. `OrderConfirmed` landet auf Partition 2 und `OrderCancelled` auf Partition 5. Shipping betreibt einen Consumer pro Partition. Der Consumer auf Partition 5 ist untätig und schnell, liest also zuerst die Stornierung und findet eine Sendung, die noch nicht existiert. Er zuckt mit den Schultern und bestätigt. Einen Moment später liest der Consumer auf Partition 2, der mit einem langsamen Batch beschäftigt war, die Bestätigung und plant eine Sendung für eine Bestellung, die es nicht mehr gibt.

Nichts ist abgestürzt, keine Nachricht ist verloren gegangen, und keine wurde dupliziert. Jede Komponente hat getan, was für sie konfiguriert war. Am Montag wird ein Paketdienst kommen, um ein Paket abzuholen, das niemand bestellt hat. Wie viel Reihenfolge braucht ein solches System, wo geht sie verloren, und was lässt sich bauen, wenn man sie nicht haben kann?

Das Problem und warum es schwierig ist

Reihenfolge ist keine einzelne Eigenschaft, die ein System hat oder nicht hat. Sie ist eine Kette, und jedes Glied kann sie unabhängig von den anderen brechen. Der Relay kann Zeilen in einer anderen Abfolge veröffentlichen, als die Datenbank sie festgeschrieben hat (Kapitel 4 und Kapitel 5 behandeln dieses Glied). Der Broker kann zusammengehörige Nachrichten

auf Partitionen oder Warteschlangen verteilen. Consumer können parallel verarbeiten. Ein Retry kann eine Nachricht hinter ihre Nachfolger setzen (Kapitel 6), und ein Dead Letter kann sie ganz überspringen (Kapitel 8).

Glied	Wie Reihenfolge verloren geht	Wo man nachsieht
Producer und Relay	Parallele Relays, nebenläufige Commits, IDs, die nicht zur Commit-Reihenfolge passen	Kapitel 4 und 5
Broker	Zusammengehörige Nachrichten auf verschiedenen Partitionen oder Warteschlangen	Dieses Kapitel
Consumer	Mehrere Worker bearbeiten Nachrichten aus einer Partition oder Warteschlange	Dieses Kapitel
Fehlerbehandlung	Retry-Topics, Nacks, Dead Letters, Redrive	Kapitel 6 und 8
Topologieänderung	Änderungen der Partitionsanzahl, Consumer-Rebalances	Dieses Kapitel

Die naheliegende Reaktion ist, eine globale Reihenfolge zu fordern: jedes Ereignis im System in einer einzigen Abfolge, von jedem Consumer gesehen. Das klingt nach einer sicheren Voreinstellung. Es ist teuer und für die meisten Systeme unnötig.

Warum globale Reihenfolge teuer ist

Eine einzige totale Ordnung bedeutet einen einzigen Sequenzpunkt. Auf einem logbasierten Broker ist das eine Partition, die in einer Consumer-Group von höchstens einem Consumer gleichzeitig gelesen wird. Ihre Durchsatzobergrenze wird zur Geschwindigkeit eines Writers und eines Readers. Zusätzliche Kapazität bringt nichts, denn Kapazität entsteht dadurch, dass Dinge parallel laufen, und genau das verbietet eine totale Ordnung.

Ein System, das mehrere Services umspannt, macht es schlimmer. Orders, Payments und Inventory haben jeweils eine eigene Datenbank und eine eigene Outbox. Eine globale Reihenfolge über alle drei bräuchte einen Koordinator, der jedes Ereignis mit einem Stempel versieht, und dieser Koordinator läge auf dem Schreibpfad jedes Services. Sie würden die Kopplung zurückkaufen, die die Architektur beseitigen sollte.

Warum globale Reihenfolge meist unnötig ist

Fragen Sie, was das Geschäft tatsächlich braucht. Parcelly ist es gleichgültig, ob `PaymentCaptured` für `order-81724` vor `OrderPlaced` für `order-90311` kam. Diese Bestellungen beeinflussen sich nicht. Wichtig ist, dass bei einer Bestellung `OrderPlaced` vor `OrderConfirmed` gesehen wird und `OrderConfirmed` vor `OrderCancelled`.

Das ist **Reihenfolge pro Aggregat**. Ein Aggregat ist die Konsistenzgrenze, deren Kennung als Partitionsschlüssel dient. Wird jedes Ereignis zu einem Aggregat in der Reihenfolge gesehen, in der es erzeugt wurde, und dürfen sich Ereignisse zu verschiedenen Aggregaten frei vermischen, funktionieren die meisten Zustandsautomaten korrekt. Diese Reihenfolge können Sie sich leisten, und aus meiner Sicht ist sie die, um die herum Sie entwerfen sollten.

Eine Grenze überschreitet selbst die Reihenfolge pro Aggregat nicht. Ereignisse zur selben Bestellung, die von verschiedenen Services erzeugt und auf verschiedenen Topics übertragen werden, haben untereinander keine Reihenfolge. `PaymentAuthorized` und `StockReserved` können in beliebiger Abfolge bei Orders eintreffen, und genau deshalb fordert der durchgängig in diesem Buch verwendete Bestellablauf die Erfassung erst an, wenn beide eingetroffen sind. Behandeln Sie einen gemeinsamen Partitionsschlüssel nicht als Zusage über Topics hinweg.

Reihenfolge pro Aggregat

Der Mechanismus ist einfach zu beschreiben und anspruchsvoll durchzuhalten: einen Schlüssel wählen, alles mit demselben Schlüssel über einen Pfad senden und diesen Pfad eine Nachricht nach der anderen verarbeiten.

Der Partitionsschlüssel

Auf einem logbasierten Broker bildet der Producer einen Hash des `partition_key`, um eine Partition auszuwählen. Jede Nachricht mit demselben Schlüssel landet auf derselben Partition, und eine Partition ist eine geordnete Abfolge. Der Umschlag in diesem Buch trägt aus diesem Grund `partition_key`, und die Outbox speichert ihn, damit der Relay ihn setzen kann, ohne den erzeugenden Service zu fragen (Kapitel 4).

Die Garantie hat eine Bedingung: **gleicher Schlüssel, gleicher Partitionierer, gleiche Partitionsanzahl**. Der Partitionierer ist die Hash-Funktion plus die Art, wie der Schlüssel vor der Hash-Berechnung in Bytes serialisiert wird. Verwenden zwei Producer verschiedene, geht eine `order_id` auf zwei Partitionen, und die Reihenfolge pro Schlüssel ist dahin, ohne dass irgendwo ein Fehler auftritt. Das ist der konkreteste polyglotte Fehler bei der Reihenfolge. Client-Bibliotheken in verschiedenen Sprachen teilen nicht alle dieselbe Standard-Hash-Funktion, sodass ein in einer Sprache geschriebener Service und ein in einer anderen geschriebener Relay dieselbe `order_id` an verschiedene Partitionen senden können. Zwei Abwehrmaßnahmen funktionieren. Legen Sie den Partitionierer in jedem Producer ausdrücklich fest, mit demselben Algorithmus und derselben Schlüsselkodierung. Oder nehmen Sie die Entscheidung den Producern ab: Lassen Sie den Relay die Partition selbst aus dem gespeicherten `partition_key` berechnen, sodass ein einziges Stück Code die Zuordnung besitzt. Letzteres liegt nahe, wenn ohnehin jedes Ereignis durch den Outbox-Relay aus Kapitel 5 läuft.

Warteschlangenbasierte Broker sind anders. Konkurrierende Consumer auf einer Warteschlange geben Ihnen keine Reihenfolgegarantie mehr, sobald es mehr als einen gibt, weil der Broker die nächste Nachricht an den Consumer übergibt, der gerade frei ist. Manche warteschlangenbasierten Broker bieten eine Gruppierungsfunktion, die Nachrichten mit gemeinsamer Gruppenkennung serialisiert, etwa Message-Groups bei Amazon-SQS-FIFO-Warteschlangen, andere einen Modus, in dem jeweils nur ein Consumer auf einer Warteschlange aktiv ist. Diese Funktionen wirken, und jede kostet Parallelität auf andere Weise. Prüfen Sie, was Ihr Broker unter dieser Funktion für erneute Zustellung und Visibility-Timeouts verspricht, denn die Garantie hat oft Kleingedrucktes. Sagen Sie, auf welche Familie Sie sich stützen und was diese Familie wirklich verspricht.

Den Schlüssel wählen

Der Schlüssel ist die Kennung des Aggregats, und die Wahl ist eine Aussage darüber, wo Konsistenz liegt. Für den Bestelllebenszyklus von Parcelly ist der Schlüssel `order_id`. Jedes Ereignis, das eine Bestellung durch ihre Zustände führt, trägt ihn, was Shipping, Orders und Ledger eine einheitliche Abfolge pro Bestellung gibt.

Einige Fragen helfen, wenn die Antwort nicht offensichtlich ist:

- Den Zustand welcher Entität ändert der Consumer? Nach dieser Entität schlüsseln.
- Was ist die kleinste Einheit, innerhalb derer zwei Ereignisse in Konflikt geraten können? Das ist das Aggregat.
- Wird derselbe Stream von mehreren Services mit unterschiedlichen Aggregaten konsumiert? Dann wählt entweder der Producer das feinste gemeinsame, oder jeder Consumer braucht seine eigene Sicht.

Inventory zeigt die Spannung. Bestände gehören zu einer SKU, also schlüsseln Inventorys eigene Commands naturgemäß nach `sku`. Die Ereignisse, die von Orders zu Inventory fließen, schlüsseln nach `order_id`. Ein Reservierungsereignis trägt beide Kennungen, und Inventory schlüsselt intern um, wenn es Reihenfolge pro SKU braucht. Einen Schlüssel für jeden Topic im System zu wählen, ist verlockend und verschenkt den Sinn von Aggregaten.

Hot Partitions

Ein Partitionsschlüssel verteilt die Last nur dann gleichmäßig, wenn die Schlüssel gleich stark beansprucht sind. Das sind sie nicht. Stellen Sie sich einen Flash Sale bei Parcelly vor, bei dem das einzige Produkt eines Verkäufers den Großteil der Tagesbestellungen binnen einer Stunde erhält. Schlüsselt Inventory seinen eigenen Topic nach `sku`, erzeugt dieser eine Schlüssel den größten Teil des Verkehrs, alles landet auf einer Partition, und ein Consumer soll die Arbeit der ganzen Group leisten, während die anderen untätig sind.

Mehr Partitionen helfen nicht. Ein einzelner Schlüssel wird immer auf eine einzelne Partition abgebildet, und zusätzliche Partitionen verschieben nur andere Schlüssel um ihn herum. Es gibt vier ehrliche Antworten:

- **Das Aggregat kleiner machen.** Erlaubt es die Fachregel, nach etwas Feinerem schlüsseln, etwa einer Reservierungs-ID statt der SKU. Die Reihenfolge innerhalb der SKU wird gegen Streuung eingetauscht, und das ist nur akzeptabel, wenn keine Regel Reihenfolge pro SKU verlangt.
- **In den Consumer investieren, nicht in die Partitionen.** Jede Nachricht billig bearbeitbar machen, damit ein Consumer mit dem heißen

Schlüssel Schritt halten kann.

- **Isolieren.** Bekannte heiße Schlüssel auf einen eigenen Topic oder eine eigene Partition leiten, damit sie den gewöhnlichen Verkehr, der die Partition teilt, nicht aushungern.
- **Gelockerte Reihenfolge für diesen Stream akzeptieren.** Ein zufälliges Suffix an den Schlüssel hängen, um die Last zu verteilen, und sich auf Versionsprüfungen (weiter unten in diesem Kapitel) verlassen, um die Reihenfolge beim Consumer wiederherzustellen. Das ist die flexibelste und aufwendigste Variante. Beachten Sie, was Salting bringt und was nicht: Es verteilt die Last auf dem Broker, aber die Versionsprüfung serialisiert den Schlüssel in der Datenbank des Consumers über die Zeilensperre des Aggregats erneut. Der Durchsatz verbessert sich nur, wenn die teure Arbeit vor dem serialisierten Schritt stattfindet oder wenn die Operationen kommutieren.

Ich würde die Schlüsselschieflage messen, bevor ich wähle. Eine Schlüsselverteilung, die in einer ruhigen Woche gleichmäßig aussieht, kann an einem geschäftigen Tag stark einseitig sein, und das erste Anzeichen ist meist Consumer-Lag auf genau einer Partition (Kapitel 13).

Parallelität und Reihenfolge ziehen in entgegengesetzte Richtungen

Eine Consumer-Group skaliert, indem jede Partition einem Consumer zugewiesen wird, sodass die Anzahl der Partitionen die Obergrenze der Parallelität ist. Hat Shipping sechs Partitionen, können sechs Consumer parallel arbeiten. Ein siebter bliebe untätig.

Teams versuchen oft, den Durchsatz innerhalb eines Consumers zu steigern, indem sie Nachrichten an einen Thread-Pool übergeben. Das wirkt harmlos und zerstört die Reihenfolge. Zwei Nachrichten zur selben Bestellung, nacheinander gelesen, gehen an zwei Worker, und der langsamere Worker wird als zweiter fertig. Die Partition hat sie in Reihenfolge geliefert, und die Verarbeitung hat sie nicht bewahrt.

Beides erreicht man, indem man über Schlüssel hinweg parallelisiert und innerhalb eines Schlüssels serialisiert. Bilden Sie im Consumer den Hash des Schlüssels auf eine feste Lane ab, und geben Sie jeder Lane genau einen 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)
```

Drei Details tragen den Entwurf. Das erste ist der Offset. Commits erfolgen pro Partition, und `contiguousDoneOffset` ist der höchste Offset, unterhalb dessen jede Nachricht dieser Partition erledigt ist. Wenn Lanes unterschiedlich schnell fertig werden, würde der Commit des Offsets der schnellsten Lane nach einem Absturz Nachrichten überspringen, die eine langsamere Lane noch nicht abgeschlossen hatte. Der Commit geschieht in der Poll-Schleife, weil viele Log-Clients nicht sicher aus mehreren Threads aufgerufen werden können. Der Preis der Parallelität im Consumer ist diese Buchführung und sind die Duplikate nach einem Absturz, die eine Inbox aus Kapitel 7 abfängt.

Das zweite ist die Begrenzung. Ohne sie hält eine langsame Lane den Commit-Punkt an, während die Poll-Schleife weiter den Speicher füllt, und die Lücke zwischen dem festgeschriebenen Offset und der laufenden Arbeit ist genau die Zahl der Nachrichten, die nach einem Absturz erneut zugestellt werden. Die Partitionen zu pausieren, wenn eine Lane ihre High-Water-Marke überschreitet, macht aus Überlast Lag, der sichtbar und un-

gefährlich ist, statt Speicherwachstum, das beides nicht ist. Der Mechanismus und die Wahl zwischen Pausieren und Verwerfen sind die, die in „Retries unter Überlast: Backpressure“ in Kapitel 6 beschrieben werden.

Das dritte ist der Fehlerfall. Schlägt `handleMessage` fehl und wiederholt die Lane an Ort und Stelle, wartet jeder Schlüssel, der die Lane teilt, hinter ihr, und der festgeschriebene Offset rückt nicht mehr vor. Die Nachricht beiseitezulegen lässt die Lane weiterziehen und ordnet diesen Schlüssel um. Das ist dieselbe Entscheidung, die Kapitel 6 für ganze Partitionen getroffen hat, nun pro Lane getroffen, und die Antwort hängt auf dieselbe Weise vom Consumer ab.

Bei einem warteschlangenbasierten Broker erscheint dieselbe Idee eine Ebene höher. Brauchen Sie Reihenfolge pro Schlüssel über konkurrierende Consumer hinweg, nutzen Sie entweder die Gruppierungsfunktion des Brokers oder nehmen hin, dass die Warteschlange außer der Reihe zustellt, und behandeln das im Consumer. Nehmen Sie nicht an, dass eine Warteschlange mit mehreren Consumern die Reihenfolge bewahrt, weil die Nachrichten geordnet hineingingen. Das tut sie nicht.

Eintreffen in falscher Reihenfolge mit Versionen erkennen

Manchmal kann der Broker Ihnen die benötigte Reihenfolge nicht liefern, wegen eines Retry-Topics, eines Redrives, paralleler Consumer oder einer Änderung der Partitionsanzahl. In diesen Fällen muss der Consumer es selbst bemerken. Dazu müssen die Ereignisse etwas tragen, das ein Consumer vergleichen kann.

Zeitstempel sind das Falsche zum Vergleichen. Kapitel 3 hat erklärt, dass `occurred_at` die Uhr einer einzelnen Maschine widerspiegelt und dass ihre Verwendung zur Entscheidung, welches Update gewinnt, einen Service mit schneller Uhr jeden Konflikt gewinnen lässt, den er verlieren sollte. Verwenden Sie `occurred_at` für Menschen und Analysen.

Was funktioniert, ist eine **Versionsnummer** (auch Sequenznummer genannt): ein Zähler, den das Aggregat bei jeder Änderung erhöht, in derselben Transaktion wie die Zustandsänderung geschrieben und mit dem Ereignis übertragen. Da nur der zuständige Service das Aggregat ändert und dies innerhalb einer Transaktion tut, bilden die Versionen pro Aggregat

eine lückenlose Abfolge: 1, 2, 3. Lückenlos braucht auch eine Regel in der Gegenrichtung: **Jede Versionserhöhung erzeugt genau ein veröffentlichtes Ereignis.** Ein internes Update, das die Version anhebt, ohne etwas zu veröffentlichen, hinterlässt eine dauerhafte Lücke, und jeder Consumer wird auf ein Ereignis warten, das es nicht gibt.

Der Standard-Umschlag hat dafür kein Feld, und ich würde den Umschlag nicht für ein Kapitel ändern. Ich würde sie im Payload tragen, als `order_version` bei Orders-Ereignissen. Sie ist die Aggregatversion (Kapitel 4), dieselbe Zahl, die die Outbox-Zeile speichert. Die tiefere Diskussion des Umschlagentwurfs überlasse ich Kapitel 11. Der Zähler gehört zu einem Producer und einem Aggregat. Ein Consumer vergleicht Werte von `order_version` nur unter Orders-Ereignissen derselben Bestellung und nie mit einem Ereignis von Payments oder Inventory, die einen eigenen Zähler bräuchten. Eine Abhängigkeit zwischen Services löst man nicht mit einer gemeinsamen Sequenz. Man löst sie mit einer Zustands- oder Vorbedingungsprüfung, wie es der Erfassungsschritt tut, wenn er auf `PaymentAuthorized` und `StockReserved` wartet. Die Outbox-Zeile bekommt sie umsonst, da der Service die neue Version, die Zustandsänderung und die Outbox-Zeile in einer Transaktion schreibt.

Die Entscheidung des Consumers

Ein Consumer, der eine Sicht auf ein Aggregat führt, hält auch die letzte Version fest, die er angewendet hat. Für jedes eintreffende Ereignis vergleicht er Versionen und nimmt einen von drei Pfaden.

Eintreffende Version, verglichen mit der zuletzt angewendeten	Bedeutung	Aktion
Gleich oder niedriger	Duplikat oder veraltet	Ignorieren, zählen, bestätigen
Genau eins höher	Das nächste Ereignis	Anwenden, neue Version festhalten
Mehr als eins höher	Eine Lücke: Etwas fehlt oder ist verspätet	Das Ereignis puffern, nicht anwenden

```
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)
```

Zwei Eigenschaften machen das sicher. Die Versionsprüfung und die Zustandsaktualisierung werden atomar festgeschrieben, sodass zwei Worker, die denselben Schlüssel bearbeiten, nicht beide Version 4 anwenden können. Und die Prüfung ist idempotent: Ein Duplikat ist eine veraltete Version wie jede andere und wird ignoriert, sodass sie die Inbox aus Kapitel 7 ergänzt, statt sie zu ersetzen. Auch `holdEvent` muss idempotent sein. Eine doppelte Zustellung einer bereits gehaltenen Version würde sonst die Eindeutigkeitsbedingung des Puffers verletzen, die Transaktion scheitern lassen, die Nachricht unbestätigt lassen und zu einer Poison-Schleife werden.

Zwei Voraussetzungen werden oft übersehen. Die erste ist der Beginn der Sequenz. Ein Consumer, der mitten im Stream beginnt, etwa ein neuer Service oder ein Replay ab einem späteren Offset, sieht zuerst Version 5 und würde ewig auf die Versionen 1 bis 4 warten. Befüllen Sie seine Sicht aus einem Snapshot oder aus der API des zuständigen Services, zusammen mit der Version, die dieser Snapshot widerspiegelt, und beginnen Sie von dort zu vergleichen.

Die zweite ist, dass die Lückenerkennung nur funktioniert, wenn der Consumer jede Version sieht. Ein Consumer, der nur eine Teilmenge der Ereignisse eines Aggregats abonniert, sagen wir nur `OrderCancelled`, sieht die Versionen 3 und 7 und schließt fälschlich, dass 4 bis 6 fehlen. Für solche Consumer schwächt sich die Regel zu „nie etwas anwenden, das nicht neuer ist als das, was ich habe“ ab, was weiterhin vor veralteten Überschreibungen schützt, aber keine Lücken erkennen kann.

Last-Writer-Wins-Fallen

Die einfachste Konfliktrichtlinie ist, dass der letzte Schreibvorgang gewinnt. Sie ist einfach, und sie verbirgt drei verschiedene Fallen, je nachdem, was „letzter“ bedeutet.

Zuletzt eingetroffen. Die Sicht wird von der Nachricht überschrieben, die der Consumer zuletzt bearbeitet hat. Ein erneut zugestelltes `OrderPlaced`, das nach `OrderConfirmed` eintrifft, setzt den Status zurück auf `PLACED`, den Fehler, auf den Kapitel 7 hingewiesen hat. Die Eintreffreihenfolge ist das Einzige, was die Broker-Topologie womöglich nicht verspricht.

Zuletzt nach Zeitstempel. Dem Anschein nach besser, in der Praxis schlechter. Uhrenabweichung zwischen Services bedeutet, dass das siegreiche Ereignis das mit der schnelleren Uhr sein kann. Zwei Ereignisse, die binnen Millisekunden auf verschiedenen Maschinen geschrieben wurden, haben eine bedeutungslose Zeitstempelreihenfolge.

Zuletzt nach Version. Das funktioniert, und es hat seine eigene Grenze. Ein veraltetes Ereignis zu verwerfen, ist nur richtig, wenn das neuere Ereignis bereits alles enthält, was das veraltete gesagt hat. Das gilt für Ereignisse, die Zustand tragen („die Bestellung ist jetzt `CANCELLED`“). Es gilt nicht für Ereignisse, die Änderungen tragen, etwa „4.990 zum Saldo addieren“. Ein veraltetes Änderungsereignis zu ignorieren, verliert die Änderung.

Eine Klarstellung verhindert, dass dies der obigen Tabelle widerspricht. Bei lückenlosen Versionen, die streng der Reihe nach angewendet werden, ist eine niedrigere Version immer ein Duplikat, weil sie bereits angewendet wurde; sie zu ignorieren ist auch bei Deltas richtig. Die Delta-Falle tritt auf, wenn ein Consumer zur neuesten Version springt und eine Lücke überspringt, denn die übersprungenen Änderungen sind dann weg.

Diese Unterscheidung entscheidet, was Sie bei einer veralteten oder außer der Reihe eingetroffenen Version tun:

- **Zustandstragende Ereignisse** (der vollständige Zustand oder ein Zustand, der den vorigen überschreibt): Das veraltete Ereignis zu ignorieren ist richtig, weil das spätere es ablöst.
- **Delta- oder Übergangsereignisse** (Inkremente, Ledger-Einträge, alles, was nichts ablöst): nie verwerfen. Puffern und umordnen, oder in beliebiger Reihenfolge verarbeiten, wenn die Operation kommutativ ist. Ledger insbesondere darf keinen Eintrag verwerfen, weil er älter ist.

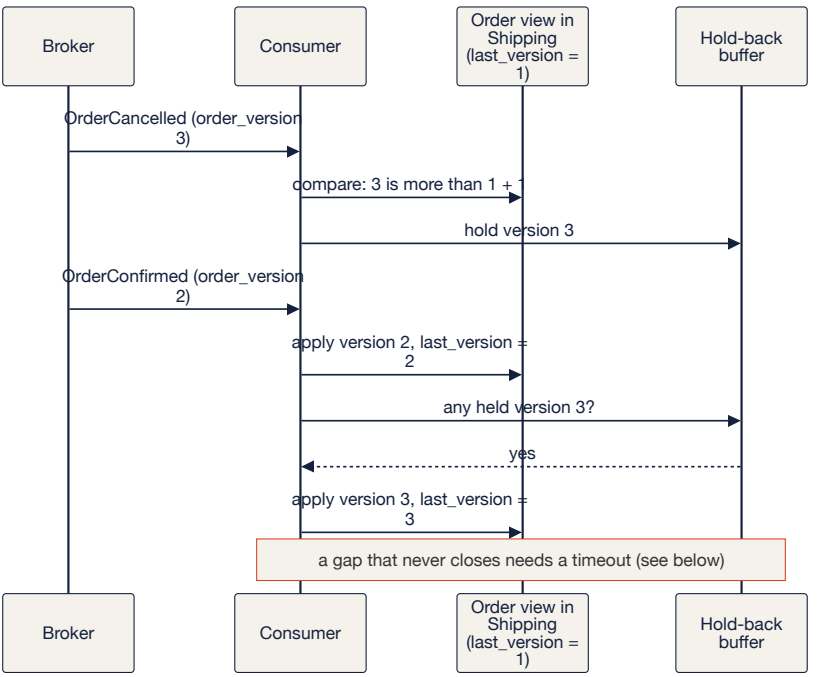
Eine verwandte Falle ist das gelöschte Aggregat. Löscht ein Consumer seine Zeile, wenn er `OrderCancelled` sieht, findet ein verspätetes `OrderPlaced` keine Zeile, besteht die Versionsprüfung, weil nichts zum Vergleichen da ist, und legt eine neue Bestellung an. Wiederauferstehung ist der charakteristische Bug, wenn man die Reihenfolge rund um Löschungen ignoriert. Führen Sie einen Tombstone: den Schlüssel, die letzte Version und eine Löschmarkierung, mindestens so lange, wie veraltete Ereignisse noch eintreffen können. Kapitel 8 hat dieselbe Wiederauferstehung gezeigt, wenn ein per Redrive zurückgegebenes `OrderPlaced` nach einer Stornierung eintrifft, und ein Tombstone mit Versionsprüfung verhindert sie.

Schließlich verdient das Ignorieren eine Spur. Ein stillschweigend verworfenes veraltetes Ereignis ist ein Bug, den niemand sieht. Zählen Sie ignorierte Ereignisse pro Consumer und halten Sie Schlüssel und Versionen in einer Log-Zeile fest, damit ein Anstieg der Zahl auffällt, bevor ein Kunde es bemerkt (Kapitel 13).

Puffern und Umordnen

Sieht der Consumer eine Lücke, hat er zwei Wege weiter. Er kann das Ereignis trotzdem anwenden und einen falschen Zustand riskieren, oder er kann das Ereignis zurückhalten, bis das fehlende eintrifft. Die zweite Möglichkeit ist ein **Hold-back-Buffer**.

Das folgende Sequenzdiagramm zeigt Shipping beim Empfang zweier Ereignisse von Orders für `order-81724`, Version 3 vor Version 2, wie im Eröffnungsszenario. Beide stammen von einem Producer und einem Aggregat, und das macht den Vergleich ihrer Versionen sinnvoll. Shipping handelt nur bei `OrderConfirmed`, aber damit das funktioniert, liest es den gesamten Orders-Stream zur Bestellung und hält die Version jedes Ereignisses fest, auch der Typen, die es sonst ignoriert. Dadurch ist `last_version = 1` der Sicht echt, und dadurch wird Lückenerkennung überhaupt möglich. Achten Sie darauf, wo Version 3 wartet und was sie freigibt.



Der Freigabeschritt ist eine Schleife. Nach dem Anwenden eines beliebigen Ereignisses prüft der Consumer, ob der Puffer die nächste Version enthält, wendet sie an und wiederholt das, bis der Puffer keinen Nachfolger mehr hat.

```
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)
```

Der Puffer selbst ist eine Tabelle, geschlüsselt nach Aggregat und Version, sodass er einen Neustart übersteht. Ein Puffer im Arbeitsspeicher verschwindet mit dem Prozess, und der Broker hat diese Nachrichten bereits als bestätigt betrachtet.

Die Bereinigung braucht noch eine Regel. Springt `last_version`, etwa nach dem Ersetzen der Sicht durch abgerufenen Zustand, sind gehaltene Zeilen mit `version <= last_version` überholt, und der Consumer löscht sie in derselben Transaktion. Sonst liegen sie für immer in der Tabelle, und die Drain-Schleife erreicht sie nie.

Die Lücke, die sich nie schließt

Puffern ist optimistisch: Es nimmt an, dass das fehlende Ereignis lediglich verspätet ist. Manchmal ist es verloren, als Dead Letter abgelegt oder wurde nie an diesen Consumer gesendet. Ohne Begrenzung bleibt das Aggregat für immer blockiert, und die gehaltene Menge wächst.

Jeder Puffer braucht einen Timeout und eine Entscheidung, was bei dessen Ablauf geschieht. Es gibt drei vernünftige Antworten, und ich würde nach den Kosten eines Irrtums wählen:

- **Den fehlenden Zustand abrufen.** Den zuständigen Service per API nach dem aktuellen Zustand fragen und die Sicht ersetzen. Das funktioniert, wenn der Producer eine solche API anbietet, und macht den Puffer zu einem schnellen Pfad mit Sicherheitsnetz.
- **Alarmieren und warten.** Ein Mensch schaut nach. Richtig für Ledger, wo ein falscher Eintrag mehr kostet als ein langsamer.

- **Die Sicht als verdächtig markieren und den Abgleich sie reparieren lassen.** Richtig für Read Models, die Veralterung tolerieren können (Kapitel 13).

Das Blockieren pro Schlüssel aus Kapitel 8 ist die versionsfreie Form dieses Mechanismus. Dort blockiert eine Dead-Letter-Markierung den Schlüssel, spätere Ereignisse warten in einem Speicher für gehaltene Nachrichten, und eine besondere Umgehung lässt das per Redrive zurückgegebene Ereignis durch, um sie freizugeben. Hier erzeugt ein als Dead Letter abgelegtes Ereignis eine Lücke, spätere Ereignisse warten im Puffer, und das per Redrive zurückgegebene Ereignis gibt sie von selbst frei, weil es als `last_version + 1` eintrifft und die Drain-Schleife den Rest erledigt. Die Versionen ersetzen sowohl die Markierung als auch die Umgehung, weshalb dies der stärkere Entwurf ist, wo der Producer sie liefern kann.

Die Kosten sind real: eine Puffertabelle, die dimensioniert und bereinigt werden muss, eine Timeout-Richtlinie, die abzustimmen ist, und eine neue Klasse hängender Aggregate, wegen der jemand alarmiert werden muss. Tragen Ihre Ereignisse den vollständigen Zustand und können Ihre Consumer „das Neueste gewinnt“ gefahrlos anwenden, ist der Puffer unnötig, und ich würde ihn weglassen.

Retries, Dead Letters und Reihenfolge

Kapitel 6 und Kapitel 8 endeten jeweils mit einer Frage zur Reihenfolge, die sie für dieses Kapitel aufgehoben haben. Die Kurzfassung ist, dass jeder Mechanismus der Fehlerbehandlung eine Nachricht in der Zeit verschiebt, und das ist eine Umordnung.

- **Retry an Ort und Stelle** auf einer Log-Partition blockiert die Nachfolger, sodass die Reihenfolge erhalten bleibt und die ganze Partition in Latenz zahlt.
- **Ein Retry-Topic oder eine Delay-Queue** lässt Nachfolger vorbeiziehen, sodass die fehlgeschlagene Nachricht später zurückkehrt als Ereignisse, denen sie vorausging.
- **Ein Visibility-Timeout** auf einem warteschlangenbasierten Broker legt eine Nachricht nach Ablauf des Timeouts in die Warteschlange zurück, wo sie mit Nachrichten konkurriert, die nach ihr gesendet wurden.

- **Ein Dead Letter** überspringt die Nachricht, und ein **Redrive** bringt sie nach allem zurück, was darauf folgte.

Die Versionsprüfung macht die zweite bis vierte Option überlebbar. Nehmen Sie das Beispiel aus Kapitel 6: Inventory erhält von Orders `OrderPlaced` und dann `OrderCancelled` zu einer Bestellung, und `OrderPlaced` wird nach einem vorübergehenden Fehler in einen Retry-Topic verschoben. Mit der Aggregatversion von Orders ist `OrderPlaced` Version 1 und `OrderCancelled` Version 2. Die Stornierung trifft zuerst ein, und Inventory hat noch keine Sicht auf die Bestellung, sodass `last_version` standardmäßig 0 ist und 2 mehr ist als 0 + 1. Die Stornierung wird gehalten und nicht gegen eine Reservierung angewendet, die nicht existiert. Trifft das wiederholte `OrderPlaced` als Version 1 ein, wird es angewendet, und die Drain-Schleife gibt die Stornierung frei. Die verwaiste Reservierung, die Kapitel 6 beschrieben hat, entsteht nicht.

Das funktioniert nur für Consumer, die Versionen und einen Puffer führen. Ein Consumer, der das nicht tut, muss wählen, wie Kapitel 6 argumentiert hat, zwischen dem Blockieren der Partition und dem Tolerieren der Umordnung. Und der Rat von Kapitel 8 für Hauptbücher gilt weiter: Für einen Consumer, bei dem ein verspäteter oder fehlender Eintrag einen falschen Finanz-Record bedeutet, anhalten und warten. Ein pausierter Consumer ist laut, und ein Ledger, das Einträge stillschweigend außer der Reihe anwendet, ist es nicht.

> **Warnung:** Ein Retry-Topic in Kombination mit einem Consumer ohne Versionsprüfung ordnet Ereignisse stillschweigend um. Nichts schlägt fehl, kein Alarm löst aus, und der Zustand ist falsch. Wenn Sie Fehler beiseiteleiten, machen Sie die Versionsprüfung zum Teil derselben Änderung.

Die Anzahl der Partitionen ändern

Die Partition für einen Schlüssel ist eine Funktion des Schlüssels und der Anzahl der Partitionen, meist ein Hash modulo der Anzahl. Ändert man die Anzahl, ändert sich die Funktion.

Angenommen, zur Veranschaulichung, der Hash eines Schlüssels ist 17. Bei vier Partitionen wird er auf Partition 1 abgebildet, bei sechs Partitionen auf Partition 5. Ein Hash von 20 wandert von Partition 0 zu Partition 2, während 14 und 15 bleiben, wo sie waren. Die meisten Schlüssel wandern. Bei Modulo-Hashing behält ein Schlüssel beim Wechsel von vier auf sechs seine Partition nur, wenn sein Hash modulo 12 gleich 0, 1, 2 oder 3 ist, sodass etwa zwei Drittel der Schlüssel anderswo landen. Der Fehler ist nicht selten: Er trifft jeden Schlüssel, der während der Änderung Nachrichten in Bearbeitung hat.

Die Gefahr ist die Überlappung. Nachrichten für einen wandernden Schlüssel, die vor der Änderung geschrieben wurden, liegen auf der alten Partition, noch unkonsumiert. Nachrichten, die nach der Änderung geschrieben werden, gehen auf die neue Partition. Zwei Consumer bearbeiten nun denselben Schlüssel gleichzeitig, ohne Reihenfolge untereinander. Führt ein Consumer außerdem lokalen Zustand pro Schlüssel, etwa einen Cache oder ein Aggregationsfenster, liegt dieser Zustand auf der falschen Instanz.

Wege, damit umzugehen, in grober Reihenfolge der Präferenz:

1. **Von Anfang an genug Partitionen bereitstellen.** Gegenüber der erwarteten Spitzenparallelität überdimensionieren, denn die Zahl ist billig zu halten und teuer zu ändern. Viele logbasierte Broker erlauben das Hinzufügen von Partitionen, aber nicht das Entfernen, sodass ein Fehler in einer Richtung dauerhaft ist.
2. **Vor der Änderung leerlaufen lassen.** Producer anhalten oder den Stream pausieren, die Consumer das Ende der alten Partitionen erreichen lassen, dann die Anzahl ändern. Die Reihenfolge bleibt erhalten, zum Preis eines Fensters der Nichtverfügbarkeit.
3. **Auf einen neuen Topic migrieren.** Einen Topic mit der richtigen Anzahl anlegen, eine Umschaltung durchführen, bei der Consumer den alten Topic beenden, bevor sie den neuen beginnen, und den alten stilllegen. Es ist die sicherste und aufwendigste Variante.
4. **Sich auf Versionsprüfungen verlassen.** Erkennt und puffert jeder Consumer Ereignisse außer der Reihe, ist der Übergang ohne Pause überlebbar, und der Übergang selbst ist zugleich ein guter Test, ob die Prüfung funktioniert.

Rebalances und Entzug

Rebalances einer Consumer-Group haben im Kleinen eine ähnliche Gestalt, und sie geschehen bei jedem Deployment. Wandert eine Partition zu einem anderen Consumer, kann Arbeit in Bearbeitung beim alten mit den ersten Nachrichten beim neuen überlappen. Was der Consumer beim Entzug tut, entscheidet, wie groß diese Überlappung ist.

Beim Entzug hört der Consumer auf, neue Nachrichten aus den entzogenen Partitionen zu verteilen, lässt die Lanes beenden, was sie halten, oder verwirft es, und schreibt den lückenlosen Offset fest, bevor er die Partition abgibt. Ob ein Rebalance jede Partition entzieht oder nur die, die wandern, hängt von der Zuweisungsstrategie Ihres Clients ab, und eine Alles-auf-einmal-Strategie legt die ganze Group für die Dauer lahm; prüfen Sie, welche Sie einsetzen. Ein Consumer, der langsam bemerkt, dass er eine Partition verloren hat, verarbeitet als Zombie weiter, während der neue Besitzer dieselbe Arbeit tut, also rechnen Sie mit etwas Überlappung, was immer Sie tun.

Die Inbox und die Versionsprüfung halten diese Überlappung nur für Effekte innerhalb der Datenbanktransaktion des Consumers selbst unschädlich. Für eine E-Mail oder einen Aufruf der API des Paketdienstes, den beide Consumer absetzen, leisten sie nichts. Diese externen Seiteneffekte brauchen einen eigenen Idempotenzschlüssel, wie Kapitel 7 beschreibt, denn die Inbox schützt lokalen Zustand und sonst nichts.

Was jeder Parcelly-Consumer braucht

Nicht jeder Consumer braucht dieselbe Reihenfolge, und sie alle gleich zu behandeln, ist eine Quelle von Bugs wie von Verschwendung. Eine für Parcelly gezeichnete Tabelle zeigt, wie ich entscheiden würde.

Consumer	Reihenfolgebedarf	Mechanismus	Wenn eine Nachricht fehlt
Notifications	Keiner; jede Nachricht ist ein unabhängiger Fakt	Beliebige Reihenfolge, kein Schlüssel nötig	Dead Letter und weitermachen
Orders-Statussicht	Pro Bestellung, zustandstragend	Nach `order_id` schlüsseln, veraltete Versionen ignorieren	Aktuellen Zustand abrufen

Consumer	Reihenfolgebedarf	Mechanismus	Wenn eine Nachricht fehlt
Inventory-Reservierungen	Pro Bestellung, Übergänge	Nach `order_id` schlüsseln, Versionen plus Hold-back-Buffer	Schlüssel halten, alarmieren
Shipping	Pro Bestellung, `OrderConfirmed` vor `OrderCancelled`	Nach `order_id` schlüsseln, Versionen plus Hold-back-Buffer	Schlüssel halten, alarmieren
Ledger	Pro Bestellung, keine Lücken, kein Verwerfen	Nach `order_id` schlüsseln, bei Lücke in der von Payments verantworteten Sequenz pro Bestellung anhalten	Anhalten und auf einen Menschen warten

Die Ledger-Zeile benennt ihren Zähler mit Absicht. Ledger konsumiert Payments-Ereignisse, vergleicht also eine Sequenz, die Payments pro Bestellung erhöht, und nicht `order_version`, die zu Orders gehört. Ein Zähler ist nur unter Ereignissen des einen Services vergleichbar, dem er gehört.

Die erste Zeile ist die günstigste und verdient mehr Anerkennung, als sie bekommt. Der beste Umgang mit Reihenfolge ist, den Consumer so zu entwerfen, dass sie keine Rolle spielt. Eine kommutative Operation, etwa einer Menge ein Element hinzuzufügen oder das Maximum zweier Werte zu nehmen, liefert in jeder Reihenfolge dasselbe Ergebnis. Ein Consumer, der aus solchen Operationen besteht, braucht keinen Schlüssel, keinen Puffer und keine Version. Lässt sich ein Handler so umformen, würde ich das tun, bevor ich zu Maschinerie greife.

Abwägungen und Kosten

Sie tauschen Parallelität gegen Reihenfolge, und der Wechselkurs ist das Aggregat. Ein gröberer Schlüssel gibt stärkere Reihenfolge und weniger Streuung. Ein feinerer Schlüssel gibt Streuung und schwächere Reihenfolge. Es gibt keinen Schlüssel, der beides liefert, und die Entscheidung gehört demjenigen, der die Fachregeln kennt.

Heiße Schlüssel sind Entwurfskosten, kein Betriebsunfall. Ein Partitionsschlüssel, der aus einer fachlichen Kennung gewählt wird, erbt die Schieflage des Geschäfts. Ein Flash Sale deckt sie auf. Sie können sie ab-

mildern, und Sie können sie nicht beseitigen, ohne die Reihenfolge für diesen Schlüssel aufzugeben.

Versionen verlangen Disziplin von den Producern. Das Aggregat muss die Version in derselben Transaktion wie die Zustandsänderung erhöhen, bei jeder Änderung, aus jedem Codepfad. Ein einziger Pfad, der es vergisst, hinterlässt eine Lücke, auf die Consumer für immer warten. Umgekehrt gilt es ebenso: Jede Erhöhung muss genau ein Ereignis veröffentlichen, sonst hat die Sequenz eine Lücke, die kein Consumer schließen kann. Backfills, Admin-Werkzeuge und Datenmigrationen sind die Stellen, an denen das scheitert, und ein Consumer, der mitten im Stream beginnt, braucht eine befüllte Sicht, bevor er irgendetwas vergleichen kann.

Puffer fügen Zustand und einen neuen Fehlermodus hinzu. Eine Holdback-Tabelle braucht Dimensionierung, Bereinigung, einen Timeout, einen Verantwortlichen und einen Alarm. Sie ist auch ein weiterer Ort, an dem sich ein hängendes Aggregat verstecken kann. Für Consumer, die „das Neueste gewinnt“ anwenden können, überspringen Sie sie.

Versionsprüfungen ordnen nicht über Aggregate oder Topics hinweg. Sie schützen die Sequenz eines Aggregats. Muss eine Regel wissen, dass ein Ereignis auf Topic A vor einem Ereignis auf Topic B kam, liefert eine Versionsnummer das nicht. Das ist ein Workflow-Problem, und Kapitel 10 behandelt es als eines.

Wann man sich nicht darum kümmern muss. Ist der Consumer kommutativ, tragen die Ereignisse den vollständigen Zustand und „das Neueste gewinnt“ ist richtig, oder ist ein verspätetes Ereignis harmlos und ein periodischer Abgleich repariert es, überspringen Sie die Maschinerie. Weniger zu tun ist eine Entscheidung mit Kosten, und sie sollte eine festgehaltene sein.

Häufige Fehler

- **Globale Reihenfolge fordern.** Sie fühlt sich wie die sichere Voreinstellung an und begrenzt den Durchsatz auf einen Reader. Benennen Sie die echte Anforderung, Reihenfolge pro Aggregat, und entwerfen Sie dafür.

- **Nach etwas schlüsseln, das nicht das Aggregat ist.** Nach Ereignistyp, nach Service oder nach einem Zufallswert zu schlüsseln, geschieht, weil die Voreinstellung es für Sie tut. Die Lösung ist, nach der Kennung der Entität zu schlüsseln, deren Zustand der Consumer ändert.
- **Zeitstempel zur Ordnung von Ereignissen verwenden.** Sie sehen aus wie natürliche Versionsnummern. Uhrenabweichung macht die Reihenfolge bedeutungslos, tragen Sie also stattdessen eine Version vom Aggregat.
- **Innerhalb eines Consumers ohne geschlüsselte Lanes parallelisieren.** Ein Thread-Pool steigert den Durchsatz und zerstört die Reihenfolge der Partition. Bilden Sie den Hash des Schlüssels auf eine Lane ab und geben Sie jeder Lane einen Worker.
- **Veraltete Delta-Ereignisse verwerfen.** Eine Versionsprüfung passt zu zustandstragenden Ereignissen. Auf Inkremente und Ledger-Einträge angewendet, verwirft sie echte Änderungen. Klassifizieren Sie jedes Ereignis als Zustand oder Delta, bevor Sie eine Richtlinie wählen.
- **Tombstones vergessen.** Eine Zeile bei der Stornierung zu löschen, lässt ein verspätetes Ereignis die Bestellung neu anlegen. Behalten Sie nach dem Löschen Schlüssel und letzte Version, solange veraltete Ereignisse eintreffen können.
- **Ereignisse im Arbeitsspeicher halten.** Ein Puffer, der mit dem Prozess stirbt, verliert Ereignisse, die der Broker bereits als zugestellt betrachtet. Persistieren Sie ihn, in derselben Transaktion wie die Sicht.
- **Die Partitionsanzahl in Produktion ohne Plan ändern.** Die Zuordnung ändert sich beim Wechsel von vier auf sechs Partitionen für etwa zwei Drittel der Schlüssel, und die Reihenfolge ändert sich stillschweigend mit. Leerlaufen lassen, migrieren oder mit Versionen absichern.
- **Partitionierer über Producer hinweg mischen.** Services in verschiedenen Sprachen berechnen für denselben Schlüssel womöglich unterschiedliche Hashes. Legen Sie überall einen Partitionierer fest, oder berechnen Sie die Partition im Relay.
- **Unbegrenzte Lanes.** Eine langsame Lane füllt dann den Speicher, während der Commit-Punkt stillsteht. Begrenzen Sie jede Lane, pausieren Sie die Partitionen und schreiben Sie pro Partition aus der Poll-Schleife fest.

- **Beim Rebalance der Inbox für externe Effekte vertrauen.** Die Inbox deckt nur die eigene Datenbank des Consumers ab. Eine E-Mail oder ein Aufruf beim Paketdienst braucht einen eigenen Idempotenzschlüssel.
- **Annehmen, dass gemeinsame Schlüssel Ereignisse über Topics hinweg ordnen.** Dieselbe `order_id` auf zwei Topics ergibt keine Abfolge zwischen ihnen. Lassen Sie den Consumer auf beide warten, oder modellieren Sie die Abhängigkeit in einem Workflow.

Zentrale Erkenntnisse

- Globale Reihenfolge kostet Sie jede Parallelität und wird selten gebraucht; Reihenfolge pro Aggregat ist die Reihenfolge, für die sich der Entwurf lohnt.
- Ein Partitionsschlüssel, der das Aggregat identifiziert, liefert auf einem logbasierten Broker geordnete Ereignisse pro Entität, aber nur mit demselben Partitionierer und derselben Partitionsanzahl überall; warteschlangenbasierte Broker brauchen eine Gruppierungsfunktion oder einen Consumer, der Unordnung toleriert.
- Parallelität innerhalb eines Consumers muss geschlüsselt sein, mit einem Worker pro Lane, begrenzt und pro Partition festgeschrieben, sonst macht sie stillschweigend die Reihenfolge rückgängig, die die Partition geliefert hat.
- Versionsnummern, die das Aggregat innerhalb seiner Transaktion erhöht, lassen einen Consumer veraltete, doppelte und fehlende Ereignisse erkennen, Zeitstempel können das nicht.
- Ein veraltetes Ereignis zu ignorieren ist für zustandstragende Ereignisse richtig und für Deltas und Ledger-Einträge falsch, die gepuffert oder in beliebiger Reihenfolge angewendet werden müssen.
- Retry-Topics, Redrive und Änderungen der Partitionsanzahl ordnen alle Nachrichten um, sodass jeder Consumer, der von der Reihenfolge abhängt, vor deren Einsatz eine Versionsprüfung braucht.
- Jeder Hold-back-Buffer braucht einen Timeout, einen Verantwortlichen und einen Alarm, denn eine Lücke, die sich nie schließt, ist ein hängendes Aggregat; das Blockieren pro Schlüssel aus Kapitel 8 ist die versionsfreie Form derselben Idee.

Die Reihenfolge gibt jedem Ereignis einen Platz innerhalb eines Aggregats, aber ein Geschäftsprozess umspannt mehrere Aggregate und Services, und manche seiner Schritte müssen rückgängig gemacht werden, wenn ein späterer fehlschlägt. Kapitel 10 wendet sich den Sagas zu, bei denen die Reihenfolge der Schritte der Entwurf ist und die Kompensation der Weg, auf dem sich ein Workflow erholt.

TEIL IV

Workflows und Verträge

Kapitel 10 — Sagas und langlaufende Workflows

Stellen Sie sich einen Dienstagnachmittag bei Parcelly vor. Ein Käufer gibt eine Bestellung für eine Keramiklampe auf. Orders emittiert `OrderPlaced`, Payments autorisiert die Karte, und Inventory reserviert die letzte Lampe im Bestand des Verkäufers. Dann wird die Karte abgelehnt, und Payments emittiert `PaymentFailed`. Im selben Moment starrt ein zweiter Käufer auf eine Produktseite mit der Angabe „nicht auf Lager“, weil die Reservierung für die erste Bestellung die Lampe noch festhält.

Niemand hat etwas falsch gemacht. Jeder Service hat seine eigene Aufgabe erledigt, jede lokale Transaktion wurde per Commit abgeschlossen, und jede Nachricht wurde zugestellt. Das Problem ist, dass die Geschäftsoperation „diese Lampe an diesen Käufer verkaufen“ über vier Services und mehrere Minuten verteilt ist und keine Transaktion sie abdeckt. Die Hälfte davon ist passiert. Die andere Hälfte wird nicht passieren, und jemand muss die Lampe zurückgeben.

Der Zeitpunkt spielt eine Rolle. Es wurde noch nichts eingezogen, deshalb ist das Rückgängigmachen billig: die Reservierung freigeben und die Bestellung enden lassen. Wäre dieselbe Lampe bereits bezahlt gewesen und der Fehler später aufgetreten, wäre das Zurückgeben eine andere und teurere Handlung, und das Kapitel kommt auf diese Grenze zurück.

Nehmen wir nun an, derselbe Fehler tritt auf, während der Verkäufer das Angebot storniert, während der Käufer die Lieferadresse bearbeitet oder während die Antwort auf den Einzug auf dem Rückweg verloren geht. Dieses Kapitel handelt von der Maschinerie, mit der ein solcher Ablauf sauber zu Ende gehen oder sich sauber selbst rückgängig machen kann. Wie bringen Sie einen Workflow über mehrere Services dazu, in einem Zustand zu enden, den das Geschäft akzeptiert, wenn Sie keine Sperre über ihn halten können und jeder Schritt fehlschlagen kann?

Das Problem und warum es schwierig ist

Innerhalb einer einzelnen Datenbank bietet eine Transaktion Atomarität: Entweder finden alle Änderungen statt oder keine. Über Services hinweg haben Sie das nicht. Jeder Service besitzt seine Datenbank, und der Message-Broker sitzt dazwischen. Kapitel 4 hat gezeigt, wie sich die Zustandsänderung eines einzelnen Services und sein veröffentlichtes Ereignis atomar machen lassen. Die Reaktion des nächsten Services atomar mit der Änderung des ersten zu machen, hat es nicht geleistet, und es kann es auch nicht.

Die alte Antwort waren verteilte Transaktionen, meist Zwei-Phasen-Commit. Sie halten in jedem Teilnehmer Sperren, bis ein Koordinator entscheidet, und fällt der Koordinator aus, nachdem die Teilnehmer abgestimmt haben, warten diese, während sie diese Sperren halten, bis er zurückkehrt. Sie setzen Teilnehmer voraus, die das Protokoll unterstützen, und machen den langsamsten oder am wenigsten verfügbaren Teilnehmer zur Grenze für alle. Ein externer Zahlungsanbieter oder eine Paketdienst-API nimmt an Ihrer Transaktion überhaupt nicht teil. Für einen Ablauf, der einen Aufruf an einen Dritten enthält, ist diese Option bereits ausgeschlossen, bevor Sie sie abwägen.

Eine **Saga** ist die Alternative. Sie zerlegt eine Geschäftsoperation in eine Folge lokaler Transaktionen, eine pro Service. Schlägt ein Schritt fehl, führt die Saga keinen Rollback durch, weil die früheren Transaktionen bereits per Commit abgeschlossen sind und andere ihre Ergebnisse gelesen haben können. Stattdessen führt sie **Kompensationen** aus: neue lokale Transaktionen, die die früheren Schritte semantisch rückgängig machen. Eine Kompensation ist eine Vorwärtsaktion, die das Geschäft in einen akzeptablen Zustand zurückführt, keine Zeitmaschine.

In der Praxis machen drei Dinge das schwierig.

Erstens sind die Zwischenzustände sichtbar. Zwischen „Bestand reserviert“ und „Zahlung eingezogen“ ist die Bestellung weder bestätigt noch storniert, und alles, was sie liest, sieht diesen Zwischenzustand. Die Atomarität ist weg; die Isolation ebenso.

Zweitens ist Kompensation eine Designentscheidung, kein Mechanismus. Manche Schritte haben eine saubere Umkehrung, etwa das Freigeben einer Reservierung. Andere haben nur eine ungefähre, etwa die Erstattung

einer eingezogenen Zahlung, die Gebühren kostet und Tage dauert. Manche haben keine: Eine E-Mail, die rausgegangen ist, bleibt verschickt.

Drittens kann die Saga selbst fehlschlagen. Der Prozess, der den Workflow vorantreibt, kann zwischen Schritten abstürzen, eine Nachricht kann doppelt eintreffen, eine Antwort kann nie ankommen, und eine Kompensation kann genauso fehlschlagen wie eine Aktion. Alles, was die Kapitel 4 bis 9 für einzelne Nachrichten aufgebaut haben, muss jetzt für einen ganzen Ablauf gelten.

> **Hinweis:** Ein nützlicher Test, bevor Sie eine Saga einführen: Listen Sie jeden Schritt vor dem Pivot (dem Schritt, nach dem die Saga nur noch vorwärts geht; er wird weiter hinten in diesem Kapitel definiert) mit seiner Kompensation auf und benennen Sie, was sie kostet. Schritte nach dem Pivot sollen keine haben. Steht bei mehreren Schritten davor „keine“ oder „ein Support-Ticket“, liegt die Grenze zwischen diesen Services womöglich an der falschen Stelle, und es kann billiger sein, die Schritte in die lokale Transaktion eines Services zu verlegen, als sie zu koordinieren.

Parcellys Bestellung als Saga

Nehmen Sie den Ablauf aus dem Happy Path und benennen Sie seine Schritte als Saga. Jeder Schritt ist eine lokale Transaktion in einem Service, gefolgt von einem Ereignis, das in derselben Transaktion in die Transactional Outbox dieses Services geschrieben wird.

Schritt	Service	Lokale Transaktion	Kompensation
1	Orders	Bestellung in einem ausstehenden Zustand anlegen, `OrderPlaced` emittieren	Als storniert markieren, `OrderCancelled` emittieren
2	Payments	Karte autorisieren, `PaymentAuthorized` emittieren	Autorisierung aufheben, `PaymentVoided` emittieren
3	Inventory	Bestand reservieren, `StockReserved` emittieren	Reservierung freigeben, `StockReleased` emittieren

Schritt	Service	Lokale Transaktion	Kompensation
4	Payments	Zahlung einziehen, `PaymentCaptured` emittieren	Keine innerhalb der Saga: Dies ist der Pivot (eine Erstattung gehört zu einem separaten Stornierungsablauf)
5	Orders	Bestellung bestätigen, `OrderConfirmed` emittieren	Keine: wird wiederholt, bis sie gelingt

Die beiden Payments-Schritte sind keine Datenbanktransaktionen im strengen Sinn. Jeder ist ein Aufruf an einen externen Anbieter mit einem stabilen Idempotenzschlüssel, gefolgt von einer lokalen Aufzeichnung des Ergebnisses; das ist das Pattern aus Kapitel 7. Die Garantien der Saga beruhen auf dieser Paarung, und ein Timeout beim Anbieteraufruf trägt die Unsicherheit in sich, die Kapitel 3 beschrieben hat.

Die Schritte 2 und 3 laufen parallel, und der Einzug wird erst angefordert, nachdem sowohl `PaymentAuthorized` als auch `StockReserved` eingetroffen sind. Shipping reagiert auf `OrderConfirmed`, und Notifications reagiert auf das, was Shipping emittiert. Diese beiden letzten behandeln wir vorerst als außerhalb der Saga liegend und kommen darauf zurück, warum.

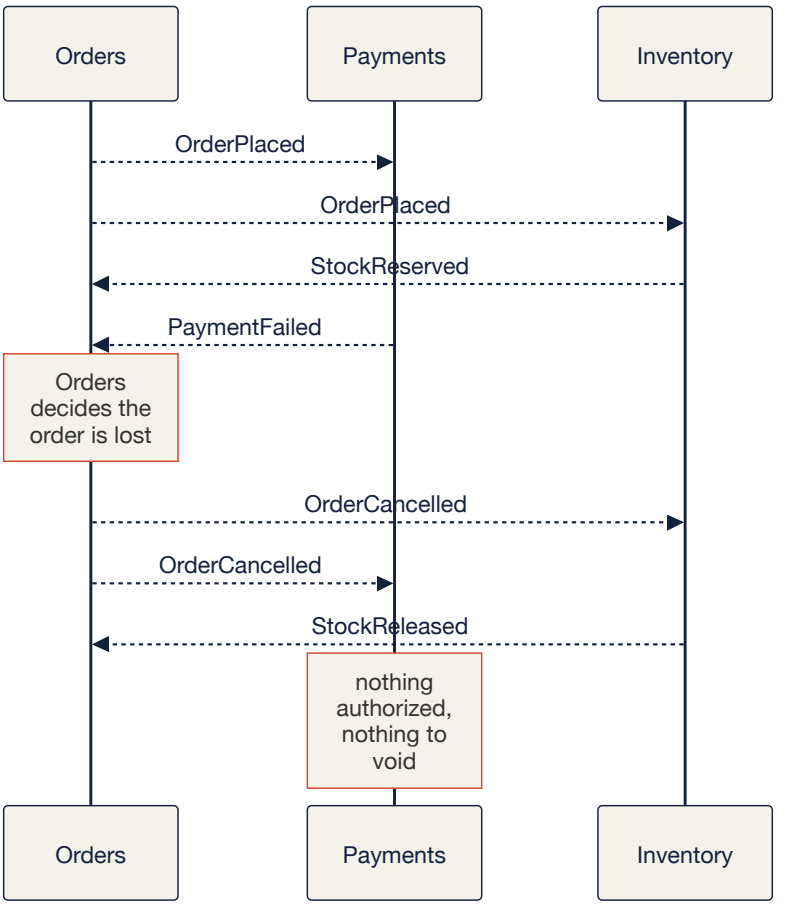
Die Fehlerzweige folgen der Tabelle aus dem Ereignisfluss. Trifft `PaymentFailed` ein, existiert keine Autorisierung, also besteht die einzige Kompensation darin, dass Inventory eine Reservierung freigibt, die möglicherweise gemacht wurde oder nicht, und `StockReleased` emittiert. Trifft `StockReservationFailed` ein, wird eine Autorisierung gehalten und nichts ist eingezogen, also hebt Payments sie auf und emittiert `PaymentVoided`. Eine Erstattung taucht erst nach dem Einzug auf. Vor dem Einzug ist das Rückgängigmachen eine Freigabe oder eine Aufhebung, und es Erstattung zu nennen, wäre im Hauptbuch falsch und auf dem Kontoauszug des Käufers falsch.

In beiden Fehlerzweigen reagiert Orders und emittiert `OrderCancelled`, und die anderen Services kompensieren als Reaktion darauf. Dieser Satz ist das gesamte Choreografie-Design. Ob es das gesamte Design ist, das Sie wollen, ist die nächste Frage.

Choreografie

In einer **choreografierten** Saga gibt es keinen zentralen Verantwortlichen. Jeder Service hört auf Ereignisse, führt seine lokale Transaktion aus und veröffentlicht das nächste Ereignis. Der Workflow existiert nur als Summe dieser Abonnements.

Das Diagramm unten zeigt den Zahlungsfehler-Zweig als Choreografie. Achten Sie darauf, wer was entscheidet: Niemand gibt eine Anweisung, und jede Reaktion wird durch ein Ereignis ausgelöst, das der Service ohnehin abonniert hat.



Die Stärken sind real. Die Services bleiben lose gekoppelt: Inventory weiß, was `OrderCancelled` bedeutet, und nichts über Payments. Es gibt keine einzelne Komponente, deren Ausfall jeden Workflow stoppt. Einen Consu-

mer hinzuzufügen, etwa einen Analytics-Service, der auf `OrderConfirmed` reagiert, erfordert keine Änderung an den anderen.

Die Schwächen zeigen sich, wenn der Ablauf wächst. Die Workflow-Definition ist über Abonnements in mehreren Codebasen verteilt, und die Frage „Was passiert nach `StockReserved`?“ lässt sich nicht beantworten, indem man eine Datei liest. Zyklische Abhängigkeiten schleichen sich ein: Orders hört auf Payments, das auf Orders hört. Jeder neue Zweig fügt Ereignisse hinzu, und jeder Service muss in sorgfältiger Reihenfolge angepasst werden. Vor allem ist niemand für die Frage „Wo steht Bestellung 81724 gerade?“ verantwortlich. Die Antwort ist ein Join über die Logs mehrerer Services, und das ist das Problem, das Kapitel 1 benannt hat und auf das Kapitel 13 mit Tracing und Abgleich zurückkommt.

Ich finde, dass Choreografie gut funktioniert, wenn der Ablauf kurz ist, wenige Zweige hat und kein Schritt von einer Und-Verknüpfung früherer Schritte abhängt. Zwei oder drei Services, die auf ein Ereignis reagieren, passen natürlich dazu. Der Bestellablauf bei Parcelly liegt bereits jenseits dieser Grenze, weil der Einzug zwei frühere Ergebnisse braucht.

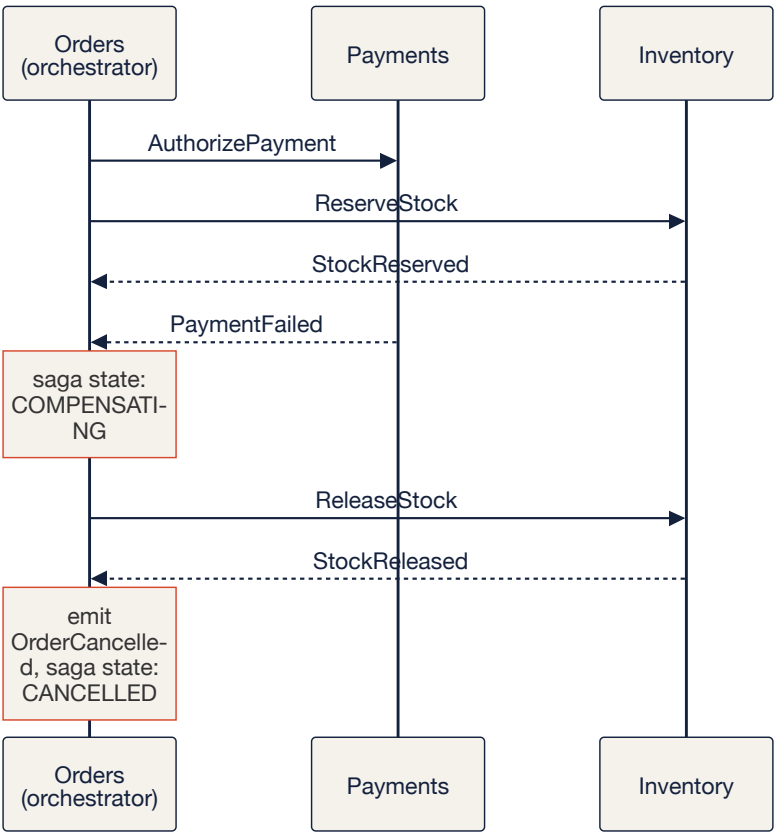
Orchestrierung

In einer **orchestrierten** Saga besitzt eine Komponente den Workflow. Sie hält den Zustand, entscheidet über den nächsten Schritt, sendet Commands an die Teilnehmer und reagiert auf deren Antworten. Die Teilnehmer wissen, wie sie ihre lokale Aufgabe erledigen und das Ergebnis melden; sie wissen nicht, wo sie im größeren Ablauf stehen.

Dieser Abschnitt führt zwei Commands über die kanonischen hinaus ein: `AuthorizePayment` (Orders an Payments) und `ReserveStock` (Orders an Inventory). Die Bestätigung braucht kein Command, weil sie die eigene lokale Transaktion des Orchestrators ist. Wie alle Commands in diesem Buch tragen sie eine `command_id` anstelle von `event_id`, und die Antworten sind die gewöhnlichen Ereignisse. In der choreografierten Version starteten Payments und Inventory bei `OrderPlaced`; hier weist der Orchestrator sie dazu an.

Das Diagramm zeigt denselben Fehler wie zuvor, aber nun ist jeder von Orders ausgehende Pfeil ein Command. Achten Sie darauf, wie der Saga-Zustand an einer Stelle liegt und wie die kompensierenden Commands ge-

nau die Arbeit benennen, die rückgängig gemacht werden soll.



Der Orchestrator ist der Ort, an dem die Frage „Was passiert als Nächstes?“ beantwortet wird. Er ist ein Zustandsautomat, und ein Zustandsautomat lässt sich lesen, testen und zeichnen. Ein neuer Zweig ist eine Änderung an einer Stelle. Der Orchestrator weiß, welche Schritte abgeschlossen sind, sodass eine Kompensation gezielt das trifft, was tatsächlich geschehen ist, statt alles, was geschehen sein könnte.

Die Kosten sind das Spiegelbild. Der Orchestrator kennt jeden Teilnehmer und wird zum Kopplungspunkt. Geht er nachlässig vor, wandert Domänenlogik aus den Services in ihn, und die Teilnehmer werden zu Remote Procedure Calls mit zusätzlichen Schritten. Er ist außerdem eine Komponente, die korrekt, immer verfügbar und selbst wiederherstellbar sein muss, denn ein abgestürzter Orchestrator mit verlorenem Zustand ist eine Saga, die nie jemand zu Ende bringen wird.

Meiner Ansicht nach ist ein Orchestrator, der in dem Service lebt, der den Geschäftsprozess besitzt, die vertretbarste Form. Bei Parcelly ist das Orders, das den Lebenszyklus der Bestellung bereits besitzt. Ein eigenständiger Orchestrierungs-Service, der die Workflows für alle besitzt, wird tendenziell zu dem, was jedes Team ändern muss, um irgendetwas auszuliefern.

Die Wahl zwischen beiden

Keiner der beiden Stile ist generell besser. Die Wahl hängt von der Form des Ablaufs ab und davon, wer Fragen dazu beantworten muss.

Frage	Choreografie	Orchestrierung
Wo ist der Ablauf definiert?	Über Abonnements verteilt	In einem Zustandsautomaten
Kopplung zwischen Services	Nur über Ereignisverträge	Der Orchestrator kennt jeden Teilnehmer
Schritt oder Zweig hinzufügen	Mehrere Services anfassen	Orchestrator und Teilnehmer anfassen
Parallele Schritte und Joins	Umständlich; jeder Service merkt sich, was er gesehen hat	Natürlich; der Orchestrator hält den Join
„Wo ist Bestellung 81724?“	Join über Logs	Eine Zeile lesen
Single Point of Failure	Keiner, aber auch kein Verantwortlicher	Der Orchestrator, der wiederherstellbar sein muss
Risiko beim Wachstum	Zyklische, verdeckte Abhängigkeiten	Ein God Service voller Domänenlogik
Gute Eignung	Kurze Abläufe, wenige Zweige	Lange Abläufe, Bedingungen, Timeouts, Joins

Beides zu mischen ist normal und vernünftig. Eine orchestrierte Bestell-Saga kann `OrderConfirmed` weiterhin als einfaches Ereignis emittieren, auf das Shipping und Notifications choreografiert reagieren. Die Saga sollte den Teil des Ablaufs abdecken, der ein koordiniertes Rückgängigmachen braucht; Benachrichtigungen und Analytics brauchen das nicht, und sie hineinzuziehen fügt nur Zustände hinzu.

Kompensationen und Schritte, die sich nicht rückgängig machen lassen

Entwerfen Sie die Kompensation eines Schritts, wenn Sie den Schritt entwerfen, nicht erst, wenn der erste Incident sie verlangt. Halten Sie für jeden Schritt fest, was das Geschäft will, wenn die Saga danach fehlschlägt.

Ein paar Regeln halten Kompensationen ehrlich.

Eine Kompensation ist eine neue Geschäftsaktion, kein Zurückdrehen der Datenbank. `StockReleased` löscht nicht die Historie der Reservierungszeile. Es hält fest, dass die Reservierung endete. `PaymentRefunded` löscht den Einzug nicht; das Hauptbuch enthält beide Einträge. Deshalb ist Ledger Append-only, und deshalb erscheinen Kompensationen dort als echte Ereignisse.

Ordnen Sie die Schritte danach, wie schwer sie sich rückgängig machen lassen. Setzen Sie leicht umkehrbare Schritte nach vorn und schwer oder gar nicht umkehrbare ans Ende. Autorisierung und Reservierung lassen sich billig rückgängig machen, deshalb stehen sie vor dem Einzug. Beim Einzug beginnt eine Erstattung, Geld und Zeit zu kosten. Daraus folgt die Idee eines **Pivot-Schritts**: der Punkt, nach dem die Saga nicht mehr kompensiert und nur noch vorwärts geht. Bei Parcelly ist der Einzug der Pivot. Davor bedeutet ein Fehler Freigabe und Aufhebung. Danach ist ein Fehler bei der Bestätigung kein Rückzug, sondern ein Retry, denn das Geschäft hat das Geld des Käufers und schuldet eine Bestellung.

Manche Aktionen haben keine Umkehrung. Eine E-Mail, die verschickt wurde, ein Paketdienst, der die Sendung erhalten hat, ein Webhook, auf den das System eines Verkäufers reagiert hat: Sie können eine Folgenachricht senden, aber nicht zurückholen. Diese Schritte gehören hinter den Pivot, und ihr Fehlschlagen sollte durch Retry und Korrektur behandelt werden, nicht durch Kompensation. Deshalb ist Notifications rein reaktiv und nie Teil der Saga: Seine Schritte lassen sich nicht kompensieren und sollten nichts blockieren können.

Kompensationen müssen wiederholt werden, bis sie gelingen. Das Fehlschlagen einer Aktion führt zu einer Kompensation. Das Fehlschlagen einer Kompensation führt zu nichts, außer zu einer festgefahrenen Saga und einem inkonsistenten System. Behandeln Sie eine Kompensation als

etwas, das schließlich abgeschlossen werden muss, mit der Retry-Disziplin aus Kapitel 6 und dem Umgang mit Poison Messages aus Kapitel 8. Eine Kompensation, die dauerhaft fehlschlägt, sollte einen Menschen über einen Parking Lot erreichen, mit aufgezeichnetem Saga-Zustand, und nicht verworfen werden.

Kompensationen müssen idempotent sein und es vertragen, zu früh einzutreffen. Das Erste ist das Thema von Kapitel 7: Ein doppeltes `ReleaseStock` darf nicht zweimal freigeben. Das Zweite ist subtiler. Angenommen, Orders sendet `ReserveStock`, läuft in einen Timeout und beginnt zu kompensieren, indem es `ReleaseStock` sendet. Die Reservierung war langsam, nicht verloren, und die Freigabe kann Inventory zuerst erreichen. Behandelt Inventory eine Freigabe für eine unbekannte Reservierung als No-op, gelingt das verspätete `ReserveStock` anschließend, und die Lampe wird für immer gehalten. Die sichere Form ist, dass Inventory die Freigabe zur Bestellung festhält, sodass die spätere Reservierung sie sieht und ablehnt. Manche Texte nennen das einen Tombstone. Wie auch immer man es nennt: Die Kompensation muss eine Markierung hinterlassen, die die Aktion lesen kann.

Saga-Zustand und Timeouts

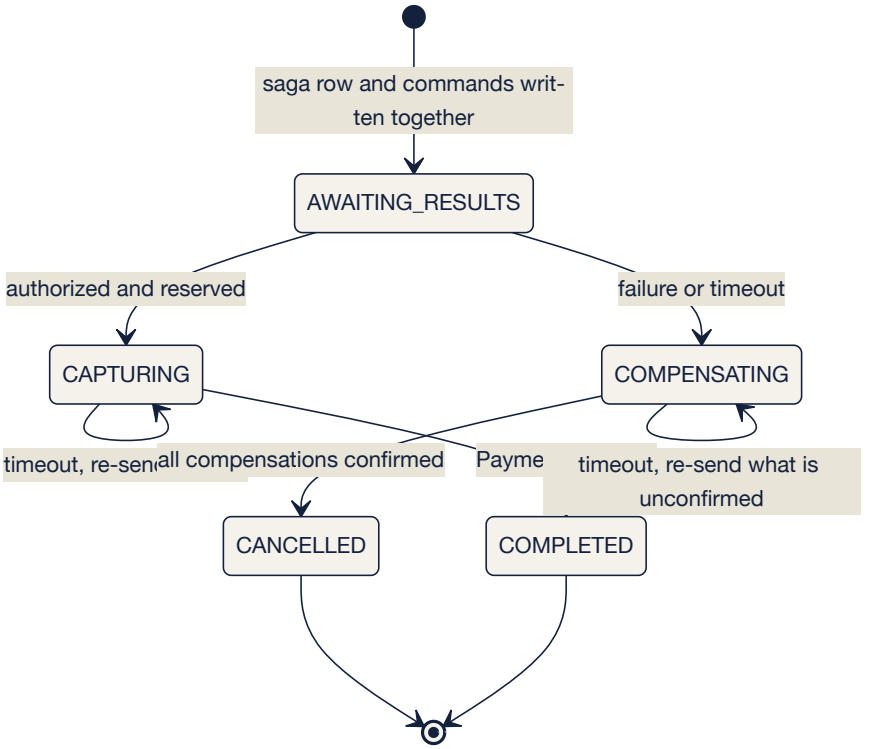
Eine orchestrierte Saga braucht dauerhaften Zustand, und er gehört in dieselbe Datenbank wie der Service, der sie ausführt, damit Zustandsänderungen in dieselbe lokale Transaktion fallen wie die Schreibvorgänge in Outbox und Inbox.

```
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
```

Die Spalte `state` ist eine kleine Menge benannter Positionen: `AWAITING_RESULTS`, `CAPTURING`, `COMPLETED`, `COMPENSATING`, `CANCELLED`. Die Saga-Zeile und die ersten Commands werden in einer Transaktion geschrieben, es gibt also keine separate Position „gestartet“, die jemand beobachten könnte. Die beiden Booleans `authorized` und `reserved` halten fest, welche Ergebnisse eingetroffen sind; so funktioniert der Join: Der Einzug wird angefordert, wenn beide wahr sind und der Zustand `AWAITING_RESULTS` ist. Die beiden Flags `stock_released` und `auth_voided` halten fest, welche Kompensationen bestätigt wurden, sodass „alle Kompensationen bestätigt“ etwas ist, das die Zeile beantworten kann. Eine Kompensation, die nicht gebraucht wird, startet als bestätigt: Nach `PaymentFailed` gibt es keine Autorisierung aufzuheben, also ist `auth_voided` beim Eintritt in `COMPENSATING` wahr.

Ein `unique` Constraint auf `order_id` verhindert, dass für eine Bestellung zwei Sagas laufen. Die Spalte `version` schützt davor, dass zwei Handler dieselbe Zeile gleichzeitig aktualisieren, und die Handler weiter unten hängen davon ab.

Ein einmal gezeichneter Zustandsautomat ist mehr wert als jede Beschreibung in Prosa. Das nächste Diagramm zeigt die zulässigen Übergänge; achten Sie auf die Ausgänge, die nicht zu `COMPLETED` führen, und auf die beiden Zustände, die bei ihrem eigenen Timeout eine Schleife in sich selbst bilden.



Beachten Sie, dass `CAPTURING` keinen Ausgang zu `COMPENSATING` hat. Das ist der Pivot, ausgedrückt im Zustandsautomaten. Ein Einzug, dessen Antwort nie kam, wird erneut gesendet, und meldet der Anbieter eine dauerhafte Ablehnung, wird das als eigener, expliziter Geschäftspfad behandelt, in dem ein Mensch vorkommt.

Die Schleifen sind ebenso wichtig wie die Ausgänge. Eine Saga in `CAPTURING` oder `COMPENSATING` wartet auf eine Antwort, und eine Antwort kann wie jede andere Nachricht verloren gehen. Hätte nur `AWAITING_RESULTS` eine Deadline, würde eine Saga, die in einem der beiden anderen Zustände festhängt, dort liegen, ohne dass etwas sie weckt. Deshalb hat jeder nicht terminale Zustand eine Deadline, und die Deadline wird bei jedem Übergang zurückgesetzt.

Timeouts

Jedes Warten in einer Saga braucht eine Deadline, denn eine Antwort, die nie kommt, ist ebenfalls ein Fehlschlag, und zwar derjenige, der am ehesten vergessen wird. Kapitel 3 hat beschrieben, wie ein Timeout drei Möglichkeiten verbirgt: Die Aktion ist fehlgeschlagen, die Aktion war erfolg-

reich und die Antwort ging verloren, oder die Aktion ist noch nicht fertig. Die Saga kann nicht unterscheiden, welche gilt, also braucht sie für jedes Warten eine Richtlinie, die aufgeschrieben ist.

Ein Timeout ist auch eine Nachricht. Dieses Kapitel führt dafür ein internes Ereignis ein, `SagaTimedOut`, das vom eigenen Timer des Orchestrators emittiert wird. Die übliche Umsetzung ist ein Timer-Prozess, der Sagas auswählt, deren `deadline_at` verstrichen ist, und jeder von ihnen ein Ereignis `SagaTimedOut` über denselben Handler zuführt wie jede andere Eingabe. Auf diese Weise nimmt der Timeout denselben Weg wie alles andere, mit derselben Inbox, denselben Zustandsprüfungen und derselben 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)
```

Der Zweig `AWAITING_RESULTS` sendet kompensierende Commands, ohne zu fragen, welcher Schritt abgeschlossen wurde. Das ist Absicht. Zum Zeitpunkt des Timeouts weiß der Orchestrator nicht, ob ein verspätetes `StockReserved` unterwegs ist, also verlangt er die noch ausstehenden Rückgängigmachungen und verlässt sich darauf, dass die Teilnehmer „nichts rückgängig zu machen“ als Erfolg behandeln. Kapitel 7 ist der Grund, warum das sicher ist.

`armNextDeadline` tut zwei Dinge innerhalb derselben Transaktion. Es setzt `deadline_at` auf `now() + timeoutFor(saga.state)`, sodass kein nicht terminaler Zustand ohne Uhr bleibt. Und sobald `saga.rounds` den

Wert `MAX_ROUNDS` überschreitet, sagen wir fünf, ist die Saga kein Timing-Problem mehr, sondern ein menschliches: Sie löst einen Alarm aus und hält den Saga-Zustand im Parking Lot aus Kapitel 8 fest, und die Runden laufen weiter. Terminale Sagas erhalten keine neue Deadline. Weil Deadline, Zustandsänderung und Commands gemeinsam per Commit abgeschlossen werden, kann ein Absturz keine Saga zurücklassen, die weitergezogen, aber nicht mehr beobachtet ist.

```
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))
```

Eine Saga in `CAPTURING` wird nie umgedreht. Der Pivot ist überschritten, also bleiben nur zwei Möglichkeiten: den Einzug mit demselben Schlüssel erneut zu senden und einen Menschen zu fragen.

Die Länge eines Timeouts zu wählen kann ich Ihnen nicht abnehmen. Sie hängt davon ab, wie lange der langsamste gesunde Pfad braucht, wie lange ein Käufer wartet und was es einen Verkäufer kostet, Bestand zu halten. Eine sehr kurz gesetzte Deadline verwandelt langsam in fehlgeschlagen und überflutet das System mit Kompensationen. Eine sehr lange hält Reservierungen, während Käufer auf eine Seite schauen, die den Artikel als nicht auf Lager ausweist.

Auch für verspätete Ankünfte braucht es eine Regel. Trifft `PaymentAuthorized` ein, nachdem die Saga `CANCELLED` ist, muss der Orchestrator erneut `VoidAuthorization` senden, weil jemand Geld auf einer Karte für eine Bestellung blockiert hält, die nicht existiert. Das verspätete Ereignis beweist, dass eine Autorisierung existiert. Die beim Timeout gesendete Aufhebung kann noch unterwegs sein, sie kann unmittelbar nach Abschluss der Autorisierung verarbeitet worden sein, oder der Anbieteraufruf kann gelaufen sein, bevor Payments auf die Markierung geprüft hat, die die Aufhebung hinterlässt. Die Aufhebung erneut zu senden regelt alle drei Fälle.

Dieses erneute Senden funktioniert nur, wenn es nicht für eine Wiederholung gehalten wird. Hier treffen zwei Regeln zur Identität aufeinander. Ein Command, dessen Wiederholung keine neue Arbeit auslösen darf, wie `CapturePayment`, behält eine `command_id` für die Saga und den Schritt, sodass ein Retry erkannt und mit dem ersten Ergebnis beantwortet wird. Eine Kompensation, die legitim erneut ausgegeben werden darf, bezieht ihre `command_id` aus der Nachricht, die sie ausgelöst hat: Saga, Schritt und die verursachende `event_id`. Die erneut ausgegebene Aufhebung ist dann ein neues Command. Sie muss außerdem idempotent bezogen auf den Zielzustand sein, nicht nur auf das Command, sodass Payments `VoidAuthorization` als „für Bestellung 81724 darf keine aktive Autorisierung bestehen bleiben“ liest und genau das prüft, statt die zwischengespeicherte Antwort „nichts aufzuheben“ vom ersten Versuch erneut abzuspielen. Ein Zustandsautomat, der Ereignisse ignoriert, die im falschen Zustand eintreffen, verliert diese. Behandeln Sie sie explizit: Entscheiden Sie pro Zustand und pro Ereignis, ob angewendet, ignoriert oder kompensiert wird.

Semantische Sperren

Weil Sagas auf Isolation verzichten, können andere Abläufe eine Bestellung mitten in der Saga lesen und ändern. Ein Käufer kann die Lieferadresse bearbeiten, während der Einzug aussteht, ein Verkäufer kann stornieren, ein Support-Mitarbeiter kann erstatten. Die Versionsnummern aus Kapitel 9 helfen bei veralteten Ereignissen, verhindern aber nicht, dass zwei Abläufe unvereinbare fachliche Änderungen vornehmen.

Eine **semantische Sperre** ist ein Flag in den Daten, das sagt: „Eine Saga arbeitet daran“. Sie ist keine Datenbanksperre. Sie ist ein Statuswert, auf dessen Beachtung anderer Code ausgelegt ist. Bei Parcelly ist der Bestellstatus `PENDING` eine solche Sperre: Solange er gilt, kann die Bestellung weder bearbeitet noch versendet noch erstattet werden, und die Bestellseite zeigt „in Bearbeitung“. Ebenso ist die Reservierung in Inventory eine Sperre auf der Lampe, weshalb der zweite Käufer „nicht auf Lager“ sah.

Die semantische Sperre muss auf jedem Pfad freigegeben werden: bei Abschluss, bei Kompensation und bei Timeout. Eine Saga, die ins Stocken gerät, ohne ihre Sperre freizugeben, lässt die Bestellung eingefroren zurück,

ein stiller Fehler, der als Support-Tickets an die Oberfläche kommt. Der Timeout oben ist die Schutzvorrichtung, und der Abgleich aus Kapitel 13 ist die zweite Linie.

Zwei weitere Gegenmaßnahmen sind es wert, bekannt zu sein. Die eine ist, Aktualisierungen wo möglich kommutativ zu gestalten, sodass die Reihenfolge keine Rolle spielt. Die andere ist, eine konservative Sicht zu entwerfen: etwa dem Verkäufer einen Bestandswert zu zeigen, der ausstehende Reservierungen bereits abzieht, damit niemand verspricht, was gleich gebunden wird. Keine von beiden nimmt die Notwendigkeit ab, für jeden Status zu entscheiden, was andere Abläufe tun dürfen, während eine Saga im Gange ist. Diese Tabelle, Status je erlaubter Operation, betrachte ich als Teil des Designs jeder Saga.

Outbox und Idempotenz innerhalb einer Saga

Der Saga-Handler ist der Ort, an dem die früheren Kapitel zusammentreffen. Er konsumiert eine Nachricht, ändert den Zustand und gibt das nächste Command oder Ereignis aus. Nachlässig umgesetzt ist das ein doppelter Dual Write.

Die Form, die funktioniert, kombiniert die Kapitel 4 und 7. Jeder Handler führt eine lokale Transaktion aus, die drei Dinge tut: die Nachricht in der Inbox festhalten, die Saga-Zeile aktualisieren und die ausgehenden Commands oder Ereignisse in die Transactional Outbox schreiben. Der Relay veröffentlicht von dort.

```
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)
```

Das ist die Regel, die Kapitel 7 für Consumer eingeführt hat, die veröffentlichen: Die Inbox-Zeile, die Zustandsänderung und der Outbox-Insert werden in einer Transaktion per Commit abgeschlossen. Stürzt der Prozess vor dem Commit ab, ist nichts passiert, und der Broker stellt erneut zu. Stürzt er nach dem Commit und vor dem Ack ab, findet die erneute Zustellung die Inbox-Zeile, wird bestätigt, und die Arbeit wird übersprungen; ein gleichzeitiges Duplikat verliert am Unique Constraint und wird auf dieselbe Weise bestätigt. Eine veraltete oder irrelevante Nachricht wird ebenfalls festgehalten und bestätigt, sodass sie nie zurückkehrt. Die Saga rückt einmal pro Nachricht vor, unabhängig davon, wie oft die Nachricht zugestellt wird.

Die Prüfung mit `expected_version` ist es, die den Join korrekt macht. `PaymentAuthorized` und `StockReserved` können im selben Moment von zwei Workern verarbeitet werden, und jeder lädt eine Zeile, in der beide Flags falsch sind. Ohne die Prüfung überschreibt der zweite Schreibvorgang den ersten, und die Saga wartet in `AWAITING_RESULTS`, bis ihr Timeout eine gesunde Bestellung kompensiert. Mit der Prüfung verliert eine Transaktion, wird samt ihrer Inbox-Zeile per Rollback verworfen, und ihre erneute Zustellung lädt eine Zeile, die das andere Flag bereits enthält. Die Zeile mit einer Zeilensperre zu laden erreicht dasselbe.

Die Commands, die die Saga sendet, sind selbst Nachrichten, die mindestens einmal zugestellt werden, also braucht der Teilnehmer denselben Schutz, und die `command_id` ist die Quelle dafür. Für eine Aktion, die sich nicht wiederholen darf, leiten Sie sie aus der Saga und dem Schrittnamen ab. So erkennt Payments ein wiederholtes `CapturePayment` und antwortet mit dem ersten Ergebnis. Erzeugt der Orchestrator bei jedem Versuch eine neue zufällige `command_id`, sieht jeder Retry nach neuer Arbeit aus, und der Käufer kann doppelt belastet werden. Kompensationen, die erneut ausgegeben werden dürfen, folgen der anderen Regel aus dem Abschnitt über verspätete Ankünfte: Die ID stammt aus der auslösenden Nachricht, und der Teilnehmer ist idempotent bezogen auf den Zielzustand. Die im Nachrichtenumschlag mitgeführten Felder `causation_id` und `correlation_id` machen außerdem jede Saga nachverfolgbar: Die `correlation_id` ist der Ablauf, und jede `causation_id` ist ein Schritt darin.

> **Warnung:** Eine Kompensation, die ohne Idempotenzschlüssel gesendet wird, ist so gefährlich wie eine Zahlung ohne einen solchen. Ein wiederholtes `RefundPayment` ohne stabilen Schlüssel kann dem Käufer doppelt erstatten, und die zweite Erstattung ist Geld, das nicht zurückkommt.

Wo Workflow-Engines passen

Alles bisher Beschriebene lässt sich mit einer Tabelle, einem Timer und ein paar Handlern bauen, und für einen Ablauf von der Größe der Bestellung bei Parcelly würde ich dort anfangen. Irgendwann wird die Maschinerie

der größere Teil des Codes: dauerhafte Timer, Retries pro Schritt, Versionierung laufender Workflows, eine visuelle Historie jeder Instanz. Das ist das Problem, das Workflow-Engines adressieren.

Die Klasse dieser Tools ist breit. Manche sind Bibliotheken oder Services, die den Fortschritt des Workflows persistieren und ihn nach einem Absturz fortsetzen, mit Code, der aussieht wie ein sequenzielles Programm. Andere interpretieren eine deklarative Definition von Zuständen und Übergängen. Cloud-Anbieter bieten verwaltete Varianten an, und es gibt mehrere Open-Source-Projekte. Ich werde keine Produkte vergleichen, weil sich die Landschaft schneller ändert als ein Buch und die Wahl von Ihrer Sprache, Ihren betrieblichen Fähigkeiten und Ihrer Toleranz dafür abhängt, eine weitere kritische Komponente zu betreiben.

Was ich von jeder Engine verlangen würde, ist eine kurze Liste. Wo lebt der Workflow-Zustand, und können Sie ihn einsehen? Wie macht sie den Seiteneffekt eines Schritts und ihren eigenen Fortschritt atomar, oder überlässt sie Ihnen den Dual Write? Was passiert mit einem bereits laufenden Workflow, wenn Sie eine geänderte Definition ausrollen? Wie werden Kompensationen und Timeouts ausgedrückt? Eine Engine, die den Fortschritt speichert, Ihren Schritt aber in einer separaten Transaktion ausführt, braucht trotzdem idempotente Schritte; die Engine verschiebt das Problem, sie beseitigt es nicht.

Die ehrliche Zusammenfassung lautet: Eine Engine erspart Ihnen, den Zustandsautomaten zu bauen und zu betreiben, und kostet Sie eine Abhängigkeit, ein Modell, das Sie lernen müssen, und eine weitere Sache, die ausfallen kann. Für ein oder zwei kleine Sagas ist eine Tabelle billiger. Für viele langlaufende Workflows mit menschlichen Schritten und langen Wartezeiten rechnet sich eine Engine.

Abwägungen und Kosten

Eine Saga tauscht Atomarität gegen Verfügbarkeit und Unabhängigkeit, und der Preis wird in mehreren Währungen bezahlt.

Komplexität. Jede Saga vervielfacht die Zustände. Ein Ablauf mit drei Schritten, jeweils mit einer Aktion und einer Kompensation, plus Timeouts und verspäteten Ankünften, hat weit mehr erreichbare Zustände, als der

Happy Path vermuten lässt, und jeder braucht eine Entscheidung und einen Test. Kapitel 15 kommt darauf zurück, wie man sie testet.

Sichtbare Zwischenzustände. Es gibt keine Isolation. Käufer, Verkäufer, Support-Tools und andere Services sehen eine Bestellung, die halb erledigt ist. Dafür entwerfen Sie mit semantischen Sperren und klaren Status, und Sie nehmen hin, dass manche Nutzer sie trotzdem sehen.

Eventual Consistency, mit Absicht. Zwischen einem Fehlschlag und der letzten Kompensation ist das System inkonsistent, für Sekunden oder, bei einem schlechten Timeout, Stunden. Das ist nur akzeptabel, wenn das Geschäft zugestimmt hat.

Betriebliche Verantwortung. Jemand muss auf Sagas achten, die über ihre Deadline hinaus in einem Zustand hängen, auf Kompensationen, die sich nicht abschließen lassen, und auf das Alter der ältesten laufenden Saga. Ohne Verantwortlichen und Alarm scheitern Sagas stillschweigend.

Kompensation ist nicht kostenlos. Eine Erstattung hat Gebühren und eine Verzögerung, eine stornierte Sendung kann eine Gebühr nach sich ziehen, und eine Nachricht an einen Käufer, „Ihre Bestellung wurde storniert“, ist selbst ein sichtbares Ereignis. Entwerfen Sie den Ablauf so, dass der häufige Fehler billig rückgängig zu machen ist.

Wann Sie keine Saga einsetzen sollten: wenn die Schritte eine lokale Transaktion teilen können; wenn Fehler selten sind und eine manuelle Korrektur billiger ist, als Kompensationen zu bauen, was für einen internen Prozess mit geringem Volumen eine legitime Wahl ist; wenn der „Workflow“ in Wahrheit unabhängige Reaktionen sind, die kein koordiniertes Rückgängigmachen brauchen, etwa das Senden einer E-Mail oder das Aktualisieren eines Suchindex. Und wenn Sie die Kompensation nicht definieren können. Kann ein Schritt nicht rückgängig gemacht und nicht verschoben werden, lautet die ehrliche Designfrage, ob die Grenze zwischen den Services an der falschen Stelle liegt.

Häufige Fehler

- **Eine Saga bauen, wo eine Transaktion genügt.** Das ist verlockend, weil die Services schon existieren und das Diagramm modern aussieht.

Haben zwei Schritte denselben Verantwortlichen und dieselbe Datenbank, führen Sie sie in einer lokalen Transaktion zusammen.

- **Den Happy Path entwerfen und Kompensationen auf später verschieben.** Später kommt als Incident. Schreiben Sie die Kompensation neben den Schritt und benennen Sie das Ereignis, das sie emittiert.
- **Jedes Rückgängigmachen Rollback oder Erstattung nennen.** Vor dem Einzug ist die Kompensation eine Freigabe oder eine Aufhebung; eine Erstattung gibt es erst nach dem Einzug. Die falsche Bezeichnung verbirgt den wahren Zustand im Hauptbuch und auf dem Kontoauszug des Käufers.
- **Eine Saga ewig warten lassen.** Eine fehlende Deadline bleibt unsichtbar, bis ein Kunde anruft. Geben Sie jedem Warten in jedem nicht terminalen Zustand einen Timeout und ein benanntes Ergebnis, einschließlich des Wartens auf eine Einzugsantwort und auf Bestätigungen von Kompensationen.
- **Ereignisse ignorieren, die im falschen Zustand eintreffen.** Ein verspätetes `PaymentAuthorized` für eine stornierte Bestellung ist eine blockierte Karte. Entscheiden Sie pro Zustand und Ereignis, ob Sie anwenden, ignorieren oder kompensieren.
- **Nicht idempotente Kompensationen.** Eine wiederholte Freigabe oder Erstattung, die doppelt wirkt, verursacht beim Beheben des ersten Incidents einen zweiten. Geben Sie Aktionen, die sich nicht wiederholen dürfen, eine stabile `command_id`, leiten Sie die ID einer erneut ausgebbaren Kompensation aus ihrer Ursache ab und machen Sie Teilnehmer idempotent bezogen auf den Zielzustand. Eine feste ID für eine erneut gesendete Aufhebung spielt „nichts aufzuheben“ ab und lässt die Autorisierung gehalten.
- **Das ganze Geschäft in den Orchestrator legen.** Der Orchestrator sollte Reihenfolge und Ergebnis entscheiden; die Teilnehmer sollten die Regeln besitzen. Validiert der Orchestrator Bestandsmengen, erledigt er die Arbeit von Inventory.
- **Nicht kompensierbare Schritte früh setzen.** Die Übergabe an den Paketdienst vor dem Einzug zu senden bedeutet, dass ein Fehler Ihnen eine Sendung ohne Geld hinterlässt. Ordnen Sie die Schritte so, dass die schwer rückgängig zu machenden zuletzt kommen.

- **Die Freigabe der Sperre überspringen.** Eine semantische Sperre, die bei `PENDING` gesetzt und bei Timeout oder Kompensation nie gelöst wird, friert die Bestellung ein. Geben Sie sie auf jedem Austrittspfad frei.

Zentrale Erkenntnisse

- Eine Saga ersetzt eine verteilte Transaktion durch eine Folge lokaler Transaktionen und Kompensationen und verzichtet dafür auf Isolation.
- Eine Kompensation ist eine neue Geschäftsaktion, etwa `StockReleased` oder `PaymentVoided`, kein Rollback, und vor dem Einzug ist sie nie eine Erstattung.
- Choreografie passt zu kurzen Abläufen mit wenigen Zweigen, Orchestrierung passt zu Abläufen mit Joins, Timeouts und vielen Zweigen, und beide werden oft gemischt, wobei der Orchestrator in dem Service liegt, der den Prozess besitzt.
- Ordnen Sie die Schritte danach, wie schwer sie sich rückgängig machen lassen, und behandeln Sie den ersten Schritt, der sich nicht kompensieren lässt, etwa den Einzug, als Pivot, nach dem die Saga nur noch vorwärts geht.
- Jedes Warten in jedem nicht terminalen Zustand braucht eine Deadline, jeder Zustand braucht eine Regel für verspätete und doppelte Ereignisse, und jede Kompensation muss idempotent sein und es vertragen, vor der Aktion einzutreffen, die sie rückgängig macht.
- Innerhalb eines Handlers sollte eine lokale Transaktion Inbox, Saga-Zustand und Outbox vereinen, abgesichert durch eine Versionsprüfung, damit gleichzeitige Handler einander nicht überschreiben können, und mit einer `command_id`, die für eine Aktion, die sich nicht wiederholen darf, wie den Einzug, fest ist, für eine Kompensation, die erneut ausgegeben werden kann, aber aus der auslösenden Nachricht abgeleitet wird.
- Eine Workflow-Engine kann den Zustandsautomaten, die Timer und die Historie besitzen, nimmt Ihnen aber weder die Notwendigkeit idempotenter Schritte ab noch die klare Verantwortung für festhängende Workflows.

Eine Saga setzt voraus, dass Ereignisse beschreiben, was geschehen ist, und zwar in einer Form, die jeder Teilnehmer lesen und der er vertrauen kann. Ob sie das tun, hängt davon ab, was jedes Ereignis trägt, wie es heißt und wem es gehört; das ist das Thema von Kapitel 11.

Kapitel 11 — Ereignisdesign

Stellen Sie sich vor, das Shipping-Team von Parcelly möchte Sendungen schneller einplanen. Heute erhält es `OrderConfirmed`, das eine `order_id` und sonst kaum etwas trägt, und ruft dann Orders per HTTP auf, um die Lieferadresse und die Bestellpositionen abzuholen. Orders ist montagmorgens langsam, also wird Shipping mit langsam. Ein Tech Lead schlägt die naheliegende Lösung vor: die gesamte Bestellung ins Ereignis packen, damit Shipping nie fragen muss.

Innerhalb einer Woche ist das Ereignis gewachsen. Orders fügt die E-Mail-Adresse des Käufers hinzu, die vollständige Adresse, das Auszahlungskonto des Verkäufers und ein Feld namens `internal_status`, das die eigene Admin-Oberfläche verwendet. Notifications beginnt, die E-Mail-Adresse aus dem Ereignis zu lesen statt aus der eigenen Tabelle. Dem Ledger-Team fällt auf, dass das Auszahlungskonto nun durch den Broker wandert und in einem Topic mit langer Aufbewahrungsdauer liegt. Dann benennt Orders beim Refactoring `internal_status` um, und drei Consumer fallen am selben Nachmittag aus.

Niemand in dieser Geschichte hat einen Zuverlässigkeitsfehler im Sinne der früheren Kapitel gemacht. Die Outbox funktionierte, die Consumer waren idempotent, die Retries waren vernünftig. Der Schaden entstand durch das Design der Nachricht selbst: was sie sagt, was sie trägt und wer sie ändern darf. Dieses Kapitel fragt, wie Sie ein Ereignis so entwerfen, dass es auch Jahre nach dem Lesen durch den ersten Consumer nützlich, sicher und änderbar bleibt.

Das Problem und warum es schwierig ist

Kapitel 1 hat darauf hingewiesen, dass Services in einem ereignisgetriebenen System über den Vertrag des Ereignisses gekoppelt sind: seinen Namen, seine Felder und seine Bedeutung. Diese Kopplung ist zu dem Zeitpunkt, an dem Sie das Ereignis schreiben, unsichtbar, denn in diesem Mo-

ment gibt es einen Producer und einen Consumer, und beide stammen von Leuten, die nahe beieinander sitzen. Der Vertrag fühlt sich wie ein internes Detail an.

Intern bleibt er nicht mehr, sobald ein zweiter Consumer abonniert, und der Producer erfährt davon meist spät oder nie. Bei einem logbasierten Broker kann eine neue Consumer-Group den Topic lesen, ohne es jemandem zu sagen. Bei einem warteschlangenbasierten Broker bindet jemand eine neue Warteschlange an denselben Exchange oder dasselbe Topic-Abonnement. Von da an ist jedes Feld im Payload ein Versprechen an Leute, die Sie vielleicht nie treffen.

Drei Dinge machen das in der Praxis schwierig.

- **Zu Ereignissen lässt sich leicht etwas hinzufügen, und schwer etwas wegnehmen.** Ein Feld hinzuzufügen kostet eine Codezeile. Es zu entfernen, erfordert, jeden Consumer zu finden, auch die in Repositories, die Sie nicht lesen, und die, die alte Historie neu abspielen.
- **Der Designdruck zeigt in die falsche Richtung.** Das produzierende Team will ausliefern, also emittiert es, was gerade zur Hand ist. Die konsumierenden Teams wollen zusätzliche Aufrufe vermeiden, also fordern sie mehr Felder. Niemand wird dafür bezahlt, Nein zu sagen.
- **Fehler sind dauerhaft.** Ein Ereignis, das heute geschrieben wird, kann monatelang in einem Ereignislog liegen und in einen Consumer neu abgespielt werden, den es noch gar nicht gibt. Ein schlecht benanntes Ereignis oder ein durchgesickertes personenbezogenes Feld bleibt im Speicher, lange nachdem der Code verschwunden ist, der es erzeugt hat.

Ereignisdesign ist also keine Dekoration. Es ist der Teil des Systems, in dem einige Stunden Nachdenken darüber entscheiden, wie teuer jede spätere Änderung wird.

Ereignisse, Commands und Dokumente

Die erste Designentscheidung betrifft die Art der Nachricht, die Sie senden. Drei Arten sind verbreitet, und für sie gelten unterschiedliche Regeln.

Ein **Ereignis** stellt eine Tatsache fest, die bereits eingetreten ist: `PaymentCaptured`. Dem Producer ist egal, wer zuhört, und er erwartet nichts zurück. Ereignisse werden in der Vergangenheitsform benannt, weil sie etwas melden, das bereits geschehen ist; ein späteres Ereignis kann ihm entgegenwirken, es aber nicht ungeschehen machen. Ein Consumer darf ein Ereignis ignorieren, und das ist eine legitime Reaktion.

Ein **Command** bittet einen bestimmten Service, etwas zu tun: `CapturePayment`. Es hat einen einzigen vorgesehenen Empfänger und die Erwartung, dass der Empfänger handelt. Ein Command kann abgelehnt werden, und der Sender interessiert sich meist für das Ergebnis. Commands stehen im Imperativ, tragen eine `command_id` anstelle von `event_id` und werden nie Ereignisse genannt. Eine orchestrierte Saga, die Kapitel 10 behandelt, steuert ihre Teilnehmer mit Commands dieser Art.

Eine **Dokumentnachricht** trägt einen Datenblock, ohne zu sagen, dass etwas geschehen ist oder dass jemand handeln soll, etwa ein nächtlicher Katalog-Snapshot eines Verkäufers. Der Empfänger entscheidet, was sie bedeutet. Dokumentnachrichten sind legitim, aber sie sind die Art, die am häufigsten als Ereignis getarnt wird, und die Tarnung stiftet Verwirrung.

Art	Sagt	Empfänger	Benennung	Kann der Empfänger ablehnen?
Ereignis	Eine Tatsache ist eingetreten	Alle Abonnenten	Vergangenheitsform: `OrderPlaced`	Ja, durch Ignorieren
Command	Tu das	Ein benannter Service	Imperativ: `CapturePayment`	Ja, durch eine Fehlerantwort
Dokument	Hier sind Daten	Wer sie liest	Nomen: `CatalogSnapshot`	Nicht anwendbar

Der häufige Fehler ist das Command im Ereigniskostüm. Angenommen, Orders veröffentlicht `SendConfirmationEmail`, einen Namen im Imperativ, und nennt es ein Ereignis, weil es durch den Topic läuft. Orders hängt nun davon ab, dass Notifications sich auf eine bestimmte Weise verhält, kann aber keine Ablehnung hören. Ist Notifications ausgefallen, weiß Orders das nicht. Abonniert ein zweiter Service, steht er vor einer Anweisung, die nicht an ihn gerichtet ist.

Ich würde einen schnellen Test anwenden. Fragen Sie, ob der Producer auch dann noch korrekt wäre, wenn niemand die Nachricht konsumiert. Bei `OrderPlaced` lautet die Antwort Ja: Die Bestellung wurde ohnehin aufgegeben. Bei `SendConfirmationEmail` lautet sie Nein, denn die Nachricht hat nur dann Bedeutung, wenn jemand handelt. Diese zweite Art gehört auf einen Kanal, auf dem der Empfänger benannt ist und ein Fehlschlag zum Sender zurückkommt.

Ein Teil davon ist auch eine Frage der Richtung. Ereignisse zeigen vom Verantwortlichen für eine Tatsache zu allen Interessierten. Commands zeigen von einem Koordinator zu einem Ausführenden. Beginnt ein Ereignisname wie eine Anweisung an einen bestimmten Service zu klingen, hat sich die Richtung umgekehrt.

Wie ein Command aussieht

Die beiden Arten schlagen auch unterschiedlich fehl, und der Nachrichtenumschlag muss das widerspiegeln. Ein verlorenes Ereignis ist von Natur aus still: Der Producer hat nie erwartet, dass jemand Bestimmtes handelt, also bemerkt niemand das Fehlen. Ein verlorenes oder verspätetes Command ist anders. Der Sender wartet auf ein Ergebnis, und das Einzige, was den Verlust aufdeckt, ist seine eigene Deadline, weshalb die Saga in Kapitel 10 in jedem nicht terminalen Zustand eine Deadline setzt. Ein Command kann außerdem eintreffen, nachdem der Sender aufgegeben und kompensiert hat, sodass der Empfänger eine Möglichkeit braucht, es abzulehnen.

Ein Command trägt daher einen kleineren, anderen Umschlag als ein Ereignis:

```
{
    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` ersetzt `event_id`, und Kapitel 10 erklärt, wie sie abgeleitet wird: aus Saga und Schritt für eine Aktion, die sich nicht wiederholen darf, und mit der auslösenden `event_id` für eine Kompensation, die erneut ausgegeben werden darf. `reply_to` und `expects` geben dem Ergebnis einen Rückweg, sodass eine Ablehnung etwas ist, das der Sender hören kann. `expires_at` erlaubt es einem Empfänger, ein Command zu verwerfen, das eintrifft, nachdem der Sender aufgehört hat zu warten. Es ist optional, weil es etwas kostet: Es wird mit der Uhr des Empfängers verglichen, und Kapitel 3 hat die Gründe genannt, aus denen man dieser Uhr nicht sekundengenau trauen kann. Setzen Sie es mit einem Puffer und betrachten Sie es als Schutz vor stundenalten Commands, nicht vor sekundenalten.

Domänenereignisse und Integrationsereignisse

Innerhalb eines Services löst Code häufig Ereignisse über die eigenen Aggregate aus: `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. Diese **Domänenereignisse** beschreiben das interne Modell, und sie ändern sich, wann immer sich das Modell ändert. Sie sind innerhalb des Services nützlich, etwa um ein Lesemodell zu aktualisieren oder in derselben Transaktion eine Regel auszulösen.

Ein **Integrationsereignis** ist die Version, die die Servicegrenze überschreitet. Es wird bewusst ausgewählt, nach dem benannt, was andere Services wissen müssen, und stabil gehalten. `OrderPlaced` ist ein Integrationser-

eignis. Es ist kein Abzug dessen, wie das Bestellaggregat in diesem Moment aussah.

Die Versuchung besteht darin, Domänenereignisse direkt an den Broker zu veröffentlichen, weil die Maschinerie schon existiert. Meiner Ansicht nach ist das der häufigste Weg, interne Struktur preiszugeben. Bedenken Sie, was passiert, wenn das Orders-Team ein Aggregat in zwei teilt oder zwei Positionstypen zusammenführt. Wären Domänenereignisse der öffentliche Vertrag, bricht jeder Consumer, obwohl sich am Geschäft nichts geändert hat.

Die Lösung ist ein Übersetzungsschritt im produzierenden Service. Domänenereignisse dürfen sich weiterhin frei ändern. Eine kleine Mapping-Funktion entscheidet, welche von ihnen zu Integrationsereignissen werden, mit welchen Namen und welchen Feldern. In dieser Funktion legen Sie fest, was der öffentliche Vertrag ist, und die Transactional Outbox aus Kapitel 4 ist ein naheliegender Ort, sie aufzurufen: Das Integrationsereignis ist es, was in die Outbox-Zeile geschrieben wird.

```
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
```

Beachten Sie, dass sich der interne Name `OrderSubmitted` und der öffentliche Name `OrderPlaced` unterscheiden. Das ist erlaubt und gesund. Der öffentliche Name ist das, was andere Teams lernen. Der interne Name

kann dem Code folgen.

Thin Events und Fat Events

Wie viel soll der Payload tragen? Die beiden Enden verdienen Namen.

Ein **Thin Event**, manchmal Notification genannt, trägt einen Bezeichner und wenig mehr: „Bestellung 81724 wurde aufgegeben“. Ein Consumer, der Details braucht, ruft den Producer zurück. Ein **Fat Event** trägt genug Zustand, damit Consumer ohne Rückruf handeln können. Dieser Stil wird oft Event-carried State Transfer genannt, weil das Ereignis ein Stück Zustand von einem Service zu einem anderen befördert.

Jede Seite hat einen Fehlermodus, den die früheren Kapitel erklären helfen.

Thin Events bringen die synchrone Kopplung zurück, die Ereignisse beseitigen sollten. Shipping kann nur fortfahren, wenn Orders in diesem Moment erreichbar ist, und das ist genau das Montagmorgen-Problem vom Anfang. Sie schaffen außerdem eine Race Condition: Bis der Consumer zurückruft, können sich die Daten geändert haben, und der Consumer liest einen neueren Zustand als den, der das Ereignis ausgelöst hat. Schlägt der Rückruf fehl, landet der Consumer in der Welt der Retries aus Kapitel 6, gegen eine Abhängigkeit, die lange langsam sein kann.

Fat Events kosten an anderen Stellen. Der Payload wächst, und mit ihm die Zahl der Felder, die Sie stabil zu halten versprochen haben. Consumer beginnen, das Ereignis als Replikat der Daten des Producers zu behandeln, und das Replikat veraltet. Ein Consumer, der am Montag die Adresse aus `OrderConfirmed` zwischengespeichert hat, nutzt sie womöglich am Freitag, nachdem der Käufer sie korrigiert hat, sofern er nicht auch das spätere Ereignis liest. Und alles im Payload wird an jeden Consumer und in jedes aufbewahrte Log-Segment kopiert, was die Einsätze für sensible Felder erhöht.

	Thin	Fat
Payload	Bezeichner und Ereignistyp	Bezeichner plus der Zustand, den Consumer brauchen

	Thin	Fat
Consumer braucht den laufenden Producer	Ja, um Details abzuholen	Nein
Risiko veralteter Daten	Gering beim Lesen, aber Race mit späteren Änderungen	Consumer hält einen Snapshot, der veraltet
Größe des Vertrags	Klein	Groß
Datenschutzrisiko	Gering	Höher
Replay	Braucht die aktuellen Daten des Producers, die vom damaligen Stand abweichen können	In sich geschlossen, gibt das damalige Wissen wieder

Ich neige zu einem Mittelweg: **Tragen Sie, was der Consumer zum Entscheiden und Handeln braucht, und nichts, was er nur vielleicht wollen könnte**. Bei `OrderConfirmed` braucht Shipping die Bestell-ID, die zu packenden Positionen und eine Lieferadresse, also gehören sie ins Ereignis. Es braucht weder die Zahlungsmethode des Käufers noch die Auszahlungsdaten des Verkäufers noch die Rabattregeln, die die Summe ergeben haben. Eine nützliche Methode, die Grenze zu finden, ist, jedes konsumierende Team zu fragen, was es mit jedem Feld tun würde. Ein Feld ohne benannten Consumer und ohne angegebenen Zweck ist ein Kandidat für die Entfernung.

Ein zweites Detail ist hier wichtig. Trägt das Ereignis Zustand, sagen Sie im Payload, zu welchem Zeitpunkt dieser Zustand gehört. Mit der Aggregatversion (Kapitel 4), die Kapitel 9 für Parcelly-Bestellungen `order_version` nennt, kann ein Consumer erkennen, ob der Snapshot, den er hält, neuer oder älter ist als ein zweites Ereignis; `occurred_at` ist aus den weiter unten genannten Gründen nur ein grober Hinweis. Kapitel 9 behandelt, wie sich eine Ankunft außer der Reihenfolge mit solchen Nummern erkennen lässt; dieses Kapitel hält daher nur fest, dass ein Fat Event ohne Version die dort beschriebene Last-Writer-Wins-Falle einlädt. Verwechseln Sie diese Nummer nicht mit `schema_version` im Nachrichtenumschlag: Die Aggregatversion zählt Änderungen an einer Bestellung und rückt bei jeder Änderung vor, während `schema_version` die inkompatible Version der Form des Payloads benennt und sich selten ändert (Kapitel 12).

Die Wahl muss nicht global sein. Ein Team kann bei selten genutzten Ereignissen thin beginnen und Felder hinzufügen, wenn ein echter Consumer danach fragt. Thin zu beginnen und zu wachsen ist billiger, als fat zu beginnen und zu schrumpfen, denn Schrumpfen bedeutet eine inkompatible Änderung, die Kapitel 12 ausführlich behandelt.

Ereignisse benennen

Die Parcelly-Ereignisnamen, die im ganzen Buch verwendet werden, folgen einer kleinen Menge von Regeln, und dieser Abschnitt nennt sie mit ihren Gründen.

- **Vergangenheitsform**, weil das Ereignis eine Tatsache ist: `PaymentCaptured`, nicht `CapturePayment`, das ist das Command.
- **Subjekt zuerst, PascalCase**: erst das Aggregat, dann das Geschehene. `ShipmentDispatched`, nicht `DispatchedShipment`. Alphabetisch sortiert gruppieren sich Ereignisse über ein Aggregat in Katalogen und Dashboards.
- **Fachsprache, nicht Technik-Sprache.** `OrderConfirmed` sagt etwas, das ein Product Owner versteht. `OrderRowUpdated` sagt etwas über eine Tabelle.
- **Weder Technologie noch Consumer im Namen.** `OrderPlacedKafkaEvent` gibt den Broker preis. `OrderPlacedForShipping` gibt einen Consumer preis, und der nächste Consumer muss sich fragen, ob das Ereignis für ihn gedacht ist.
- **Kein Service-Präfix, es sei denn, zwei Services emittieren Tatsachen über dasselbe Nomen.** Angenommen, Orders und Shipping können beide ein `DeliveryDelayed` melden: Das eine bedeutet, der Verkäufer hat das Paket nicht übergeben, das andere, dass der Paketdienst spät ist. Consumer können sie am Namen nicht unterscheiden, also wird eine Qualifizierung nötig (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). Das Beispiel ist illustrativ; die Parcelly-Services in diesem Buch emittieren nicht beide. Andernfalls erscheint der Producer bereits im Feld `producer`.

Die wichtigste Benennungsregel ist die über die Bedeutung. `OrderUpdated` ist ein Name, der den Tatsachentest nicht besteht: Er sagt, dass sich etwas geändert hat, und sonst nichts. Ein Consumer muss den

neuen Zustand mit dem alten vergleichen, um zu erfahren, was geschehen ist, und kann eine geänderte Adresse nicht von einer geänderten Summe unterscheiden. Bevorzugen Sie Namen, die die fachliche Tatsache nennen: `OrderCancelled`, `ShipmentAddressCorrected`. Finden Sie keinen fachlichen Namen, ist das oft ein Zeichen dafür, dass das Ereignis eine Datenbankänderung beschreibt und keine fachliche.

Eine weitere Regel ist Stabilität. Sobald Consumer von einem Ereignisnamen abhängen, ist eine Umbenennung eine inkompatible Änderung mit denselben Kosten wie das Entfernen eines Felds. Investieren Sie die zusätzlichen 10 Minuten in den Namen, bevor das erste Release erscheint.

Der Nachrichtenumschlag

Jedes Ereignis in diesem Buch trägt denselben Nachrichtenumschlag: `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` und `payload`. Der Umschlag ist der Teil der Nachricht, auf den sich Werkzeuge verlassen können, ohne die Fachlichkeit zu verstehen. Ein Relay, ein Dead-Letter-Handler, ein Tracing-System und ein Replay-Werkzeug können alle damit arbeiten, ohne den Payload zu parsen.

Jedes Feld beantwortet eine Frage, die irgendeine Komponente stellt:

Feld	Die Frage, die es beantwortet	Wer fragt
`event_id`	Habe ich genau dieses Ereignis schon gesehen?	Inbox, Deduplizierung (Kapitel 7)
`event_type`	Wie interpretiere ich den Payload?	Routing, Deserialisierung
`schema_version`	Welche inkompatible Version der Form hat der Payload?	Consumer, Schema-Registry (Kapitel 12)
`occurred_at`	Wann ist die Tatsache eingetreten, aus Sicht des Producers?	Abgleich, Analytics, Debugging
`producer`	Welcher Service ist für diese Tatsache verantwortlich?	Verantwortung, Support, Weiterleitung von Fragen
`correlation_id`	Zu welchem fachlichen Ablauf gehört das?	Support, Abgleich, Log-Suche
`causation_id`	Was hat das unmittelbar verursacht?	Rekonstruktion der Kette

Feld	Die Frage, die es beantwortet	Wer fragt
`partition_key`	Welches Aggregat betrifft dieses Ereignis?	Broker, Reihenfolge (Kapitel 9)
`payload`	Was sind die Details?	Geschäftslogik

Zwei Punkte brauchen Sorgfalt. `occurred_at` ist die Uhr des Producers, und Kapitel 3 hat erklärt, warum Sie Ereignisse nicht danach ordnen sollten. Es sagt Ihnen ungefähr, wann die Tatsache eingetreten ist, und dient nicht als Sequenz. Der `partition_key` ist der Aggregatbezeichner, und seine Wahl war das Thema von Kapitel 9.

`correlation_id` ist keine Trace-ID. Der Trace-Kontext (W3C Trace Context) reist in Headern neben dem Umschlag, und Kapitel 13 zeigt, wie er in die Outbox übernommen wird. Von Traces wird nur eine Stichprobe erfasst, und sie verfallen; `correlation_id` ist der fachliche Schlüssel, der beides übersteht, und derjenige, nach dem ein Support-Ingenieur sucht.

Der Nachrichtenumschlag über Sprachen hinweg

Die Services von Parcelly sind nicht alle in einer Sprache geschrieben. Orders und Payments laufen auf der JVM, Notifications ist Node, und die Abgleich-Jobs des Ledger-Teams sind Python. Die Feldliste oben sagt, was der Umschlag enthält. Sie sagt nicht, wie jedes Feld auf der Leitung geschrieben wird, und genau dort bricht eine zweite Sprache einen Vertrag, der in der ersten in Ordnung aussah.

Feld	Typ auf der Leitung	Regel
`event_id`, `correlation_id`, `causation_id`, `partition_key`	String	Bezeichner sind Strings, auch wenn sie numerisch aussehen. Eine 64-Bit-ID als JSON-Zahl wird von einem JavaScript-Consumer stillschweigend gerundet
`event_type`	String	Ein stabiler, sprachneutraler Name wie `PaymentCaptured`, nie ein Klassen- oder Paketname und nie ein Typ-Header, den ein Framework standardmäßig füllt
`schema_version`	Ganzzahl	Klein, und in jeder Sprache eine Ganzzahl

Feld	Typ auf der Leitung	Regel
`occurred_at`	String	RFC 3339 in UTC mit dem Suffix `Z`, fest auf Millisekundengenauigkeit, keine Offsets. Parser sind sich bei allem anderen uneinig
Geldbeträge in `payload`	Ganzzahl oder String	Ganzzahlige kleinste Einheiten (`total_minor`) in einem dokumentierten Bereich oder ein Dezimal-String. Nie ein Float. JavaScript-Zahlen sind nur bis 2 ^ 53 - 1 exakt, daher reist ein Feld, das darüber hinausgehen kann, als String
Enum-artige Felder	String	Behandeln Sie die Menge als offen: Ein Consumer muss einen Wert verarbeiten, den er noch nicht gesehen hat (Kapitel 12)

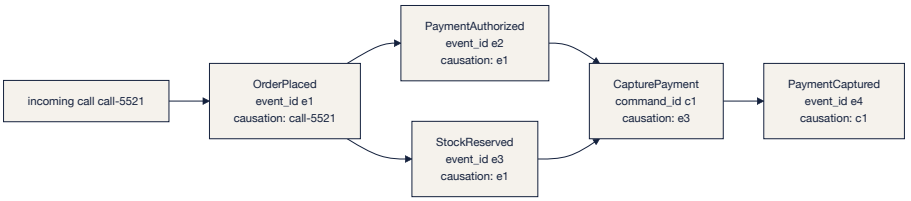
Zwei Platzierungsregeln vervollständigen den Vertrag. Der Umschlag lebt im Nachrichtenkörper, und Broker-Header spiegeln nur, was der Körper bereits sagt, denn Header sind der Teil, der verloren geht, wenn eine Nachricht kopiert, in eine Dead-Letter-Queue verschoben oder zu einem anderen Broker überbrückt wird. Und der Content-Type wird einmal im Schema deklariert, nicht pro Nachricht geraten. Was mit Feldern passiert, die ein Consumer nicht kennt, und wie sich Formate beim Dekodieren unterscheiden, ist eine Kompatibilitätsfrage und gehört in Kapitel 12.

Correlation und Causation

Die beiden Bezeichner, die Leser am häufigsten verwirren, sind `correlation_id` und `causation_id`. Sie beantworten verschiedene Fragen.

Die `correlation_id` teilen sich alle Ereignisse eines fachlichen Ablaufs. Wenn ein Käufer Bestellung 81724 aufgibt, trägt alles, was folgt, `correlation_id: "order-81724"`: die Autorisierung, die Reservierung, der Einzug, die Sendung. Danach suchen Sie, wenn ein Käufer schreibt und fragt, was geschehen ist.

Die `causation_id` zeigt auf die eine Nachricht, die diese unmittelbar verursacht hat. Sie ist die `event_id` eines Ereignisses oder die `command_id` eines Commands. Sie wiederholt sich nicht über den Ablauf. Jeder Hop setzt sie auf den Bezeichner seines unmittelbaren Auslösers, und das Verfolgen der Kette rückwärts rekonstruiert den Pfad.



Sehen Sie sich die letzten beiden Schritte an. `CapturePayment` wird von zwei Ereignissen ausgelöst, und Orders muss eines für `causation_id` wählen, hier dasjenige, das die Bedingung vervollständigt hat. Die Mehrdeutigkeit ist real, und ich würde die Regel aufschreiben, damit alle Producer auf dieselbe Weise wählen. Das Ereignis `PaymentCaptured` nennt das Command, nicht das Ereignis davor; deshalb erlaubt der Umschlag in diesem Feld beide Arten von Bezeichnern.

Jedes Ereignis im Ablauf teilt sich `correlation_id: "order-81724"`. Das erste Ereignis eines Ablaufs setzt den Korrelationsbezeichner, entweder indem es ihn aus dem Aggregat ableitet oder indem es einen aus der eingehenden Anfrage übernimmt. Spätere Ereignisse kopieren ihn unverändert.

Dieses Buch hält durchgehend an der aggregatbezogenen Form `order-81724` fest, und das ist eine Entscheidung mit Folgen. Eine aggregatbezogene ID ist einfach, aber jeder Ablauf für diese Bestellung verschmilzt darunter: Eine Erstattung drei Wochen später teilt die ID mit dem ursprünglichen Checkout, und beide lassen sich nicht unterscheiden. Die Alternative ist eine **ablaufbezogene** ID, die für jeden Ablauf neu erzeugt wird (`checkout-81724-1`, `refund-81724-1`), wobei `order_id` im Payload durchsuchbar bleibt. Sie kostet eine weitere Sache, die zu erzeugen und weiterzureichen ist, und lohnt sich, wenn ein Aggregat einen zweiten Ablauf starten kann, den Sie vom ersten unterscheiden müssen. Der Rest dieses Kapitels gilt in beiden Fällen. Kopieren ist die ganze Aufgabe, und ein Consumer, der das Kopieren vergisst, unterbricht die Spur für alles Nachgelagerte.

```
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
    }
```

Der `partition_key` des Auslösers darf nur dann kopiert werden, wenn beide Nachrichten dasselbe Aggregat betreffen. Ein Folgeereignis über eine Zahlung oder eine Sendung, das den Schlüssel der Bestellung erbt, würde gegen den falschen Strom geordnet (Kapitel 9). Parcelly schlüsselt fast alles nach Bestellung, weshalb der Fehler schwer zu entdecken ist.

Die Kosten eines Fehlers werden zum schlechtestmöglichen Zeitpunkt gezahlt. Während eines Incidents lautet die Frage: „Was ist mit dieser Bestellung über fünf Services hinweg passiert?“ Ohne konsistente `correlation_id` durchsucht ein Ingenieur fünf Logs von Hand. Kapitel 13 zeigt, wie Tracing und Abgleich diese Felder nutzen; die Aufgabe dieses Kapitels besteht darin sicherzustellen, dass sie vom ersten Ereignis an vorhanden und korrekt sind.

Was nicht in ein Ereignis gehört

Ein Ereignis verlässt die Datenbank seines Producers und gelangt in ein System mit anderen Zugriffsregeln, anderer Aufbewahrung und vielen Lesern. Die Designfrage ist daher auch eine Frage danach, was Sie in dieses System zu kopieren bereit sind.

Personenbezogene Daten. Ein Ereignis mit E-Mail-Adresse, Telefonnummer oder vollständiger Lieferadresse wird dort gespeichert, wo der Broker speichert, womöglich für die Länge des Aufbewahrungsfensters, und von jedem Consumer gelesen. In einem Ereignislog ist es später mühsam, die Daten einer Person zu löschen, denn das Log ist Append-only, und Consu-

mer haben die Felder bereits kopiert. Manche Teams antworten mit Verschlüsselungsschemata oder separaten Topics. Ich würde früher ansetzen: einen Bezeichner ins Ereignis legen, etwa `buyer_id`, und einen Consumer mit berechtigtem Bedarf die personenbezogenen Felder im Moment des Bedarfs vom zuständigen Service abholen lassen, nach den Zugriffsregeln dieses Services. Das neigt bei sensiblen Feldern zu Thin Events, und ich halte den Tausch für lohnend. Braucht ein Fat Event tatsächlich eine Adresse, wie Shipping, dann sollte das Ereignis sie bewusst tragen, für dieses Publikum, mit einer Aufbewahrungsdauer, die in dem Wissen gewählt wurde, dass sie dort liegt.

Welche Felder als personenbezogene Daten gelten und welche Pflichten daraus folgen, hängt von Ihrer Rechtsordnung und Ihren Rechtsberatern ab. Ich kenne Ihre Randbedingungen nicht. Die Designgewohnheit ist überall dieselbe: Jedes sensible Feld in einem Ereignis sollte einen benannten Grund haben.

Geheimnisse und Zugangsdaten. Tokens, Schlüssel und Auszahlungskontonummern gehören nicht in einen Payload. Eine Nachricht wird protokolliert, in Dead-Letter-Ziele kopiert, in Admin-Werkzeugen angezeigt und an Tickets angehängt. Gehen Sie davon aus, dass jeder Payload von einem Menschen gelesen wird.

Interne Details. Tabellennamen, interne Statuscodes, Feature-Flags, Datenbank-IDs, die nur innerhalb eines Services Sinn ergeben, und Werte, deren Bedeutung vom Code des Producers abhängt. `internal_status` im Eingangsszenario ist das Beispiel. Ein solches Feld ist in dem Moment ein öffentliches Versprechen, in dem es jemand liest.

Anweisungen an bestimmte Consumer. Ein Flag wie `skip_email_if_gift` sagt Notifications, wie es sich verhalten soll. Das Ereignis sollte die Tatsache feststellen, und der Consumer entscheidet, was er damit tut.

> **Warnung:** Alles, was in ein Ereignislog gelangt, sollte als dauerhaft und weithin lesbar behandelt werden. Ein Geheimnis oder ein personenbezogenes Feld, das einmal geschrieben wurde, wird in Dead-Letter-Ziele, Backups und Replay-Werkzeuge kopiert, und es überall zu entfernen ist weit schwerer, als es wegzulassen.

Eine kurze Checkliste hilft bei der Prüfung eines neuen Ereignisses: Hat jedes Feld einen Consumer, der es braucht? Ist jedes Feld etwas, das der Producer stabil zu halten bereit ist? Könnte das Feld jemandem schaden, wenn es von einer Person gelesen würde, die es nicht sehen sollte?

Granularität

Wie groß ist ein Ereignis? Es gibt zwei entgegengesetzte Fehler.

Zu fein. Für jedes Spalten-Update ein Ereignis `FieldChanged` zu veröffentlichen, erzeugt einen Strom von Rauschen. Consumer müssen die Bedeutung aus vielen Nachrichten wieder zusammensetzen, und die Reihenfolge dieser Nachrichten beginnt auf eine Weise zu zählen, vor der Kapitel 9 gewarnt hat. Eine Änderung, die eine fachliche Tatsache war, die Korrektur einer Adresse durch den Käufer, kommt nun als vier Ereignisse an, die in einem halb angewendeten Zustand zu sehen sind.

Zu grob. Ein Ereignis `OrderChanged`, das bei jeder Änderung feuert und die ganze Bestellung trägt, zwingt jeden Consumer, den neuen Zustand mit dem alten zu vergleichen und zu raten, was die fachliche Bedeutung war. Es bedeutet auch, dass jeder Consumer bei jeder Änderung aufwacht, auch bei solchen, die ihn nicht interessieren.

Ich würde die fachliche Tatsache als Einheit verwenden: ein Ereignis pro Entscheidung oder Begebenheit, die jemand im Geschäft benennen würde. Dass ein Käufer eine Bestellung aufgibt, ist eine Tatsache. Dass eine Zahlung eingezogen wird, ist eine andere. Erzeugt eine Transaktion mehrere Tatsachen, veröffentlichen Sie mehrere Ereignisse über denselben Transactional-Outbox-Schreibvorgang, sodass sie gemeinsam per Commit abgeschlossen oder per Rollback verworfen werden.

Ein praktischer Test ist die Frage, ob ein Consumer allein auf dieses Ereignis sinnvoll reagieren könnte. Lautet die Antwort „erst nachdem die nächsten drei Ereignisse eingetroffen sind“, ist die Granularität zu fein. Ignorieren die meisten Consumer den größten Teil des Payloads, oder braucht der Ereignisname ein „und“, um es zu beschreiben, ist sie womöglich zu grob und verbirgt zwei Tatsachen.

Verantwortung und der öffentliche Vertrag

Jedes Ereignis hat genau einen Eigentümer: den Service, der das Aggregat besitzt, das das Ereignis beschreibt. Payments besitzt `PaymentCaptured`. Orders besitzt `OrderPlaced`. Consumer dürfen das Ereignis lesen, aber nur der Eigentümer definiert es. Ein Consumer, der eine andere Form möchte, fragt den Eigentümer und veröffentlicht nicht eine eigene Variante unter demselben Namen.

Ein Ereignis als **öffentlichen Vertrag** zu behandeln, hat praktische Folgen, und ich würde sie in die Arbeitsvereinbarungen des Teams schreiben.

- **Der Eigentümer dokumentiert das Ereignis**: seinen Namen, seine Bedeutung, wann es emittiert wird, wann nicht, jedes Feld mit Typ und Bedeutung sowie die Garantien zu Reihenfolge und Partitionsschlüssel.
- **Das Ereignis hat eine Version und einen Änderungsprozess.** Kapitel 12 beschreibt, wie sich Schemas weiterentwickeln lassen, ohne Consumer zu brechen. Die Grundregel gehört hierher: Der Eigentümer ändert ein veröffentlichtes Ereignis nicht, ohne die Folgen für seine Leser zu kennen.
- **Consumer melden sich an.** Ein kurzes Verzeichnis, wer was konsumiert, macht aus einem stillen Abonnement eine bekannte Abhängigkeit. Es muss nicht aufwendig sein. Eine Liste im Repository, die ein Pull Request aktualisieren muss, genügt für den Anfang.
- **Consumer lesen nur, was sie brauchen.** Die Tolerant-Reader-Regel aus Kapitel 12 beginnt hier: Ignorieren Sie Felder, die Sie nicht verwenden, damit additive Änderungen Sie nicht brechen.
- **Die Emission gehört zum Verhalten.** Ein Service, der auf irgendeinem Codepfad aufhört, `OrderCancelled` zu emittieren, hat seine öffentliche Schnittstelle genauso verändert wie einer, der einen HTTP-Endpoint entfernt.

Die Wörter „Vertrag“ und „Eigentümer“ tragen beide Gewicht. Ein Vertrag lässt sich testen. Ein Eigentümer lässt sich alarmieren. Wenn niemand sagen kann, wer ein Ereignis besitzt, kann niemand sagen, wer es ändern darf, und das Eingangsszenario wiederholt sich.

Abwägungen und Kosten

Disziplin kostet Geschwindigkeit. Domänenereignisse auf Integrationsereignisse abzubilden, sorgfältig zu benennen, Dokumentation zu schreiben und eine Consumer-Liste zu führen, verlangsamt das erste Release. Der Gewinn kommt später, als billigere Änderungen. Für einen Prototyp mit einem Producer und einem Consumer, geschrieben von derselben Person, lohnt sich das Zeremoniell womöglich nicht, und ich würde es leicht halten, bis ein zweites Team auftaucht.

Fat Events tauschen Kopplung an die Zeit gegen Kopplung an die Form. Consumer brauchen den Producer nicht mehr erreichbar, aber sie hängen von den Feldern ab, die er kopiert. Ein größerer Payload bedeutet einen größeren Vertrag, mehr stabil zu Haltendes und mehr zu Schützendes.

Thin Events sind billig zu veröffentlichen und teuer zu konsumieren. Jeder Consumer zahlt einen Rückruf, und der Producer muss diese Leselast bedienen, die meist zur selben Zeit ihren Höhepunkt erreicht wie das Ereignis. Ist die API des Producers langsam, verschiebt das Thin Event das Problem vom Producer zu jedem Consumer.

Sorgfältige Nachrichtenumschläge kosten Bytes und Aufmerksamkeit. Neun Felder in jeder Nachricht summieren sich bei hohem Volumen, und jeder Producer muss sie korrekt füllen. Die Alternative ist, die fehlende Information während eines Incidents zu rekonstruieren, was mehr kostet. Wie viel das ausmacht, hängt von Ihrem Nachrichtenvolumen ab. Üblich ist, den Umschlag klein und den Payload schlank zu halten.

Regeln auf Leitungsebene kosten Flexibilität. Strings für IDs und Geldbeträge, ein festes Zeitstempelformat und ein Umschlag im Körper machen ein mehrsprachiges System langweilig im guten Sinn. Sie bedeuten aber auch Konvertierung an den Rändern, und ein Team, das native Typen in seiner eigenen Sprache will, zahlt diese Konvertierung bei jeder Nachricht.

Der Ablauf eines Commands kostet eine Uhrenabhängigkeit. `expires_at` schützt vor veralteten Commands, wird aber auf der Uhr des Empfängers beurteilt. Der Puffer, den Sie hinzufügen, ist Zeit, in der ein tatsächlich verspätetes Command noch angenommen wird.

Anonymisierte oder nur aus Bezeichnern bestehende Ereignisse begrenzen, was Consumer allein tun können. Ein Consumer, der den Käufer kontaktieren muss, braucht einen Aufruf beim Eigentümer oder eine lokale Kopie der Daten, die zu halten er einen Grund bekommen hat. Das ist Reibung, und zwar gewählte Reibung.

Wann Sie nicht so viel Struktur um ein Ereignis legen sollten. Ein rein internes Ereignis innerhalb eines Services, das von derselben Codebasis konsumiert und nie aufbewahrt wird, braucht keinen Umschlag, keine Version und keine Eigentümerliste. Heben Sie sich die volle Behandlung für Ereignisse auf, die eine Grenze überschreiten, und insbesondere für solche, die ein Deployment überleben.

Häufige Fehler

- **Domänenereignisse direkt veröffentlichen.** Das ist verlockend, weil die Maschinerie existiert, und das Ergebnis ist, dass interne Refactorings zu inkompatiblen Änderungen werden. Fügen Sie ein explizites Mapping auf Integrationsereignisse hinzu und behandeln Sie dieses Mapping als den Vertrag.
- **Ein Command ein Ereignis nennen.** Die Vergangenheitsform wird kosmetisch weggelassen oder hinzugefügt, und der Producer hängt am Ende von einem bestimmten Empfänger ab, ohne eine Ablehnung hören zu können. Senden Sie Commands an einen benannten Empfänger und erwarten Sie ein Ergebnis zurück.
- **Ereignisse `Updated`.** `OrderUpdated` sagt nichts darüber, was sich geändert hat, also vergleicht jeder Consumer den Zustand und rät. Benennen Sie die fachliche Tatsache, und wenn es keinen Namen gibt, stellen Sie in Frage, ob das Ereignis gebraucht wird.
- **Das ganze Aggregat in den Payload kippen.** Es erspart ein Gespräch und schafft einen öffentlichen Vertrag in der Größe der Datenbanktabelle. Fragen Sie jeden Consumer, was er mit jedem Feld tut, und streichen Sie, was niemand verwendet.
- **Personenbezogene Daten und interne Felder aufnehmen, weil sie zur Hand waren.** Beheben Sie das beim Review mit einer Checkliste, denn nach der Veröffentlichung liegen die Daten bereits in einem Speicher, den Sie nicht kontrollieren.

- **Kodierung nach Sprachstandard.** Ein Klassenname als `event_type`, IDs als Zahlen, Geldbeträge als Floats, Zeitstempel mit dem Offset, den die Laufzeit gewählt hat. Es funktioniert, bis die zweite Sprache es liest. Schreiben Sie die Leitungsregeln einmal auf, wie es die Tabelle oben tut.
- **Den `partition_key` des Auslösers kopieren.** Er stimmt, wenn beide Nachrichten ein Aggregat betreffen, und ist falsch bei einem Folgeereignis über ein anderes. Leiten Sie den Schlüssel vom Aggregat des Ereignisses selbst ab.
- **`correlation_id` bei jedem Hop neu erzeugen.** Jemand schreibt einen Helper, der standardmäßig einen neuen Bezeichner erzeugt. Kopieren Sie ihn vom Auslöser und testen Sie, dass jeder Consumer das tut.
- **`occurred_at` als Reihenfolge verwenden.** Ein Zeitstempel ist eine Uhrablesung, und Uhren sind sich uneinig. Verwenden Sie den Partitionsschlüssel und eine Versionsnummer, wie Kapitel 9 sie beschreibt.
- **Kein Eigentümer.** Ein Ereignis, das von „der Plattform“ oder von zwei Services unter einem Namen emittiert wird, hat niemanden, der eine Änderung freigeben kann. Weisen Sie jedem Ereignistyp genau einen zuständigen Service zu.

Zentrale Erkenntnisse

- Ein Ereignis stellt eine Tatsache fest, ein Command fordert von einem benannten Empfänger eine Aktion an, und ein Dokument trägt Daten; diese drei zu vermischen führt zu verdeckter Kopplung und verlorenen Ablehnungen, und ein Command braucht einen Antwortpfad und einen optionalen Ablauf, weil ein verlorenes nur an der Deadline des Senders bemerkt wird.
- Domänenereignisse beschreiben das interne Modell eines Services, während Integrationsereignisse der bewusst gewählte öffentliche Vertrag sind, und ein explizites Mapping dazwischen verhindert, dass Refactoring zur inkompatiblen Änderung wird.
- Thin Events halten Verträge klein, führen aber Laufzeitkopplung und Rückruf-Races wieder ein, während Fat Events den Rückruf beseitigen, aber den Vertrag vergrößern; tragen Sie also, was Consumer zum Handeln brauchen, und nichts, was sie nur vielleicht wollen könnten.

- Benennen Sie Ereignisse in der Vergangenheitsform mit dem Subjekt zuerst, verwenden Sie Fachsprache und meiden Sie Namen im Stil von `Updated`, die die Tatsache verbergen.
- Der Nachrichtenumschlag existiert, damit Werkzeuge deduplizieren, weiterleiten, nachverfolgen und ordnen können, ohne den Payload zu lesen; seine Kodierung auf der Leitung (String-IDs, RFC-3339-Zeitstempel in UTC, sichere Geldbeträge, sprachneutrale Typnamen) hält ihn in jeder Sprache lesbar, und `correlation_id` wird durch den Ablauf kopiert, während `causation_id` auf den einen direkten Auslöser zeigt.
- Halten Sie personenbezogene Daten, Geheimnisse, internen Zustand und consumerspezifische Anweisungen aus Payloads heraus, denn alles in einem Ereignislog wird weit kopiert und nur mit Mühe entfernt.
- Jedes Ereignis hat einen zuständigen Service, eine dokumentierte Bedeutung und eine Liste bekannter Consumer; das macht aus einer Nachricht einen Vertrag, der gezielt geändert werden kann.

Ein gut entworfenes Ereignis muss sich dennoch ändern, denn das Geschäft wird sich ändern, und Consumer werden unterschiedlich schnell aktualisieren, während alte Ereignisse im Log bleiben. Kapitel 12 wendet sich der Schemaevolution zu: wie man einen Vertrag ändert, von dem andere Teams bereits abhängen, ohne sie zu brechen.

Kapitel 12 — Schemaevolution

Stellen Sie sich vor, das Payments-Team von Parcelly möchte Teileinzüge unterstützen. Ein Seller versendet die Hälfte einer Bestellung mit zwei Artikeln, und der Käufer soll jetzt für diese Hälfte belastet werden und später für den Rest. Das Ereignis `PaymentCaptured` trägt ein einziges Feld, `amount_minor`, das bisher immer „die gesamte Bestellung“ bedeutet hat. Das Team benennt es in `captured_minor` um und fügt `remaining_minor` hinzu, was sauberer ist und sagt, was gemeint ist. Die Tests im Payments-Repository laufen durch, und das Deployment geht am späten Donnerstagabend raus.

Ledger liegt nicht in diesem Repository. Der Service liest `amount_minor`, um einen doppelten Buchungssatz zu verbuchen, findet nichts und weist jedes neue Ereignis als fehlerhaft formatiert zurück. Notifications liest dasselbe Feld für eine Quittung und druckt einen leeren Betrag. Ein Reporting-Job, der die Ereignisse des letzten Monats per Replay verarbeitet, trifft jetzt im selben Topic auf zwei Populationen, alte Namen und neue Namen, und sein Parser kommt nur mit einer zurecht.

Nichts davon ist ein Bug in Payments. Der Code hat getan, was das Team verlangt hat. Der Ausfall liegt darin, dass ein Ereignis von Programmen gelesen wird, denen die Autorin oder der Autor nie begegnet, und manche davon lesen Ereignisse, die vor Monaten geschrieben wurden. Wie ändert man einen Vertrag, dessen Reader man nicht kontrolliert, solange die alten Schreibvorgänge noch im Log liegen?

Das Problem und warum es schwierig ist

Ein Ereignisschema ist ein Vertrag zwischen einem Producer und jedem Consumer, ob gegenwärtig oder künftig. Eine Funktionssignatur in Ihrer eigenen Codebasis ist ebenfalls ein Vertrag, aber ein Compiler prüft beide Seiten, und Sie können beide in einem Commit ändern. Zwischen den bei-

den Seiten eines Ereignisschemas steht kein Compiler. Producer und Consumer werden an verschiedenen Tagen ausgerollt, gehören verschiedenen Teams, und der Broker nimmt alle Bytes an, die ankommen.

Drei Eigenschaften machen das schwieriger als gewöhnliche API-Versionierung.

Erstens: **Ereignisse überdauern den Code, der sie geschrieben hat.** Bei einem logbasierten Broker bleiben Ereignisse für den konfigurierten Aufbewahrungszeitraum erhalten, und manche Systeme bewahren sie unbegrenzt auf. Bei einem warteschlangenbasierten Broker ist das Fenster kürzer, aber ein Backlog oder eine Dead-Letter-Queue (DLQ) kann Nachrichten tagelang halten. Ein Consumer, der heute ausgerollt wird, muss womöglich etwas lesen, das unter einem Schema von vor einem Jahr geschrieben wurde.

Zweitens: **Der Replay macht die Vergangenheit zu einer lebendigen Eingabe.** Der Replay, definiert in Kapitel 2, ist der Weg, eine Projektion neu aufzubauen, einen neuen Service per Backfill zu befüllen oder sich nach einem Bug zu erholen. Er reicht dem heutigen Code jedes historische Schema auf einmal. Ein Consumer, der nur die neueste Form verarbeitet, funktioniert bis zu dem Tag, an dem jemand einen Replay startet, und scheitert dann zuerst an den ältesten Ereignissen. Die Form ist dabei nur die Hälfte: Ein Replay wendet auch die heutigen Geschäftsregeln und Seiteneffekte auf alte Fakten an, etwa eine Quittung erneut zu versenden oder den Zahlungsanbieter erneut aufzurufen. Ein Consumer, der per Replay neu aufgebaut wird, muss wissen, dass er einen Replay ausführt, über einen expliziten Modus oder einen Offset-Bereich, muss seine externen Effekte unterdrücken und Regeln enthalten, die für historische Fakten gültig sind. Kapitel 7 behandelt die Seite der Deduplizierung.

Drittens: **Sie können nicht alle auf einmal aktualisieren.** Bei schrittweisen Deployments laufen alte und neue Versionen desselben Service Minuten oder Stunden nebeneinander. Eine Änderung, die korrekt ist, sobald alles aktualisiert wurde, ist während des Updates trotzdem ein Incident.

Der naheliegende Ansatz besteht darin, das Schema als internes Detail zu behandeln und die Consumer hinterher anzupassen. Das funktioniert, solange es einen Consumer und ein Team gibt. Es funktioniert nicht mehr, wenn der Producer nicht weiß, wie viele Consumer es gibt, und genau die-

se Situation soll ereignisgesteuertes Design herbeiführen. Kapitel 11 hat argumentiert, dass ein veröffentlichtes Ereignis ein öffentlicher Vertrag ist. Schemaevolution ist der Weg, dieses Versprechen über die Zeit zu halten.

Kompatibilität, definiert

Dieselben Wörter werden an vielen Stellen unscharf verwendet, deshalb hier die Definitionen, die dieses Buch benutzt.

- **Abwärtskompatibilität:** Ein Consumer, der das neue Schema verwendet, kann Ereignisse lesen, die mit dem alten Schema geschrieben wurden. Neue Reader, alte Daten.
- **Vorwärtskompatibilität:** Ein Consumer, der das alte Schema verwendet, kann Ereignisse lesen, die mit dem neuen Schema geschrieben wurden. Alte Reader, neue Daten.
- **Volle Kompatibilität:** beides gleichzeitig.
- **Transitive** Varianten von jeder dieser Prüfungen vergleichen das neue Schema mit jeder früheren Version, nicht nur mit der letzten.

Die Richtung lässt sich im Kopf leicht verdrehen, deshalb stelle ich stattdessen eine konkrete Frage: „Wer geht kaputt, wenn ich diese Änderung ausrolle?“

Kompatibilität	Wer liest was	Sichere Deployment-Reihenfolge	Schützt vor
Abwärts	Neuer Reader, alte Daten	Erst Consumer, dann Producer	Replay der Historie, zurückliegende Backlogs
Vorwärts	Alter Reader, neue Daten	Erst Producer, dann Consumer	Consumern, die nicht auf Abruf aktualisiert werden können
Voll	Beide Richtungen	Beliebige Reihenfolge	Gemischten Flotten, unbekannten Consumern
Keine	Keines von beiden garantiert	Koordiniert, mit Migrationsplan	Nichts; siehe den Abschnitt zur Migration

Eine Anmerkung zu den Namen. Registries und Serialisierungsbibliotheken verwenden diese Wörter mit kleinen Unterschieden. Lesen Sie die Definitionen im Tool, das Sie einsetzen, bevor Sie sich auf eine Einstellung

verlassen, und prüfen Sie, welche Variante Ihre Registry standardmäßig anwendet: Eine nicht transitive Regel genügt für ein Log mit Replay nicht. Testen Sie mit echten Beispielereignissen aus jeder Version.

Für ein Log mit Replay ist die Abwärtskompatibilität gegenüber der gesamten Historie die Eigenschaft, die Sie brauchen, denn ein Consumer im Replay kann auf jede jemals veröffentlichte Version treffen. Für eine Warteschlange mit kurzer Aufbewahrung kann die Prüfung gegen die vorherige Version genügen. Das ist ebenso sehr eine Entscheidung über die Aufbewahrung wie eine Entscheidung über das Schema.

Sichere und inkompatible Änderungen

Eine Änderung ist **additiv**, wenn alte Consumer sie ignorieren können und neue Consumer mit ihrem Fehlen zurechtkommen. Eine Änderung ist **inkompatibel**, wenn irgendein Reader, alt oder neu, irgendein Ereignis nicht mehr interpretieren kann. Die Urteile unten setzen voraus, dass sich Reader so verhalten, wie es der Abschnitt zum Tolerant Reader beschreibt.

Änderung an `PaymentCaptured`	Übliches Urteil	Warum
Ein optionales Feld hinzufügen (`payment_method`)	Sicher	Alte Reader ignorieren es; neue Reader behandeln sein Fehlen
Ein Pflichtfeld ohne Standardwert hinzufügen	Inkompatibel	Neue Reader können alte Ereignisse nicht interpretieren
Ein optionales Feld entfernen	Nur sicher, wenn es niemand liest	Reader, die davon abhängen, sehen nichts
Ein Pflichtfeld entfernen	Inkompatibel	Alte Reader erwarten es
Ein Feld umbenennen	Inkompatibel	Gilt als Entfernung plus Hinzufügung; einige Formate können eine Umbenennung mit Aliasen auflösen
Einen Typ ändern (`int` zu `string`)	Inkompatibel	Parser scheitern oder wandeln stillschweigend um
Einen numerischen Typ erweitern	Hängt vom Format ab	Manche Formate befördern ihn, andere nicht
Die Bedeutung eines Felds ändern	Inkompatibel, und zwar die schlimmste Art	Nichts schlägt fehl; die Ergebnisse sind falsch

Änderung an `PaymentCaptured`	Übliches Urteil	Warum
Einer Enumeration einen Wert hinzufügen	Riskant	Alte Reader lehnen ihn womöglich ab oder behandeln ihn falsch

Zwei Zeilen verdienen einen genaueren Blick.

Eine Bedeutungsänderung ist die gefährliche Änderung

Ein umbenanntes Feld bricht laut. Ledger scheitert am ersten Ereignis, ein Alarm geht los, und jemand liest den Stack-Trace. Wenn sich ändert, was ein Feld bedeutet, während Name und Typ unverändert bleiben, bricht es leise. Stellen Sie sich vor, Payments beschließt, dass `amount_minor` ab jetzt „in diesem Ereignis eingezogener Betrag“ bedeutet statt „Gesamtbetrag der Bestellung“. Jeder Parser ist weiterhin erfolgreich. Ledger verbucht Teilbeträge, als wären es Gesamtbeträge, und der Fehler taucht Wochen später in einem Abgleich (Kapitel 13) als Lücke auf, die sich niemand erklären kann.

Kein Schema-Tool fängt das ab, denn das Schema ist unverändert. Nur ein Mensch, der die Änderung prüft, bemerkt es. Meine Regel: Wenn sich die Bedeutung ändert, ist es ein neues Feld oder ein neuer Ereignistyp, nie eine Bearbeitung des alten.

Enumerationen sind ein verborgener Vertrag

Einer Enumeration einen Wert hinzuzufügen ist im Schema additiv und in den Consumern oft nicht. Ein Consumer, der `payment_method` auf einen Verarbeitungszweig abbildet, mit einem Zweig `else`, der „Karte“ annimmt, behandelt ein neues `"bank_transfer"` ohne jeden Fehler falsch. Producer können dokumentieren, dass Enumerationen offen sind. Consumer können unbekannte Werte als unbekannt behandeln und sie an einen sichtbaren Fehler oder einen sicheren Standardwert leiten, statt sie in einen bekannten Fall zu falten.

Schema-Registries und Formate

Eine **Schema-Registry** ist ein Service, der die in einem System verwendeten Schemas speichert, ihnen Identifikatoren zuweist und ein vorgeschlagenes Schema gegen frühere prüfen kann. Sie erfüllt zwei Aufgaben, die

man trennen sollte.

1. **Nachschlagen.** Eine Nachricht trägt den Identifikator des Schemas, mit dem sie geschrieben wurde, oder verweist darauf. Ein Consumer holt dieses Schema und dekodiert die Bytes. Das macht kompakte Binärformate praktikabel: Feldnamen reisen nicht mit jeder Nachricht mit.
2. **Zugangskontrolle.** Bevor ein Producer ein neues Schema registriert, vergleicht die Registry es unter einer gewählten Kompatibilitätsregel mit früheren Versionen und lehnt es ab, wenn es die Regel verletzt.

Die zweite Aufgabe würde ich am entschiedensten verteidigen. Regeln, die in einer Review-Checkliste stehen, werden an dem Tag übersprungen, an dem das Team beschäftigt ist. Eine Prüfung in der Deployment-Pipeline hat nie zu viel zu tun.

Subjects, Topics und Ereignistypen

Eine Registry prüft die Kompatibilität pro **Subject**, und ob ein Subject ein Topic oder ein Ereignistyp ist, entscheidet darüber, ob sich mehrere Ereignistypen einen Topic teilen können. Dieses Teilen braucht die Reihenfolge pro Aggregat über Ereignistypen hinweg (Kapitel 9): `PaymentAuthorized` und `PaymentCaptured` einer Zahlung bleiben nur dann in Reihenfolge, wenn sie über dieselbe Partition desselben Topics laufen. Die Parcelly-Beispiele gehen von einem Topic pro Aggregattyp aus, etwa `payments.payment-events`, das mehrere nach Aggregat-ID geschlüsselte Ereignistypen trägt, mit einer Kompatibilität, die pro Ereignistyp statt pro Topic konfiguriert wird. Der Preis ist, dass Consumer nach `event_type` filtern müssen und dass ein versionierter Topic (Strategie 3) alle Ereignistypen dieses Aggregats auf einmal verschiebt. Ein Topic pro Ereignistyp ist einfacher zu konfigurieren und gibt die Reihenfolge über Typen hinweg auf.

Drei Formatfamilien

Das Buch ist formatunabhängig, dies ist also eine Orientierung und kein Produktvergleich. Avro, Protobuf und JSON Schema unterscheiden sich darin, wie die Daten auf der Leitung aussehen, und das bestimmt, was sich sicher ändern lässt.

Familie	Auf der Leitung	Felder identifiziert durch	Praktische Folge	Falle
Avro-Stil	Kompaktes Binärformat, keine Feldnamen	Name, aufgelöst zwischen Writer- und Reader-Schema	Der Reader braucht das Schema des Writers; Standardwerte tragen die Kompatibilität	Aliase lassen einen Reader ein umbenanntes Feld auflösen, eine Umbenennung ist also nicht immer inkompatibel. Ein fehlendes Feld wird mit dem Standardwert des Readers gefüllt, sein Fehlen ist also unsichtbar
Protobuf-Stil	Kompaktes Binärformat	Feldnummer	Umbenennen ist billig; eine Nummer wiederzuverwenden ist gefährlich	Billig nur auf der binären Leitung: Proto-JSON und generierter Code verwenden Namen, eine Umbenennung bricht also Quellcode und JSON. Ein proto3-Skalar ohne explizites Vorhandensein dekodiert ein fehlendes Feld als 0
JSON Schema	Text-JSON	Name; Schema zur Laufzeit optional	Einfach zu debuggen; Kompatibilität ist eine Konvention, solange sie nicht erzwungen wird	Bei einem geschlossenen Modell (`additionalProperties: false`) bricht ein hinzugefügtes Feld alte Reader, und Registries beurteilen offene und geschlossene Modelle unterschiedlich

Drei Ideen gelten für alle.

- **Standardwerte machen Ergänzungen sicher, wenn der Standardwert ehrlich ist.** Ein neues Feld mit einem definierten Standardwert lässt einen neuen Reader ein altes Ereignis verarbeiten. Ohne einen wird aus einer Ergänzung ein Bruch. Bei einem Feld, dessen Fehlen der Reader erkennen können muss, etwa einem Betrag, muss der Standardwert „kein Wert“ sein: eine Avro-Union aus `null` und dem Typ mit `null` als Standardwert oder proto3 `optional` oder ein Wrapper-Typ. Ein Standardwert von 0 dekodiert ein altes Ereignis als Einzug von null, und der Reader kann den Unterschied nicht erkennen. Der Reader prüft dann das Vorhandensein, nicht den Wert.
- **Identität sollte nicht wiederverwendet werden.** Ob ein Feld über Namen oder Nummer identifiziert wird: Einer alten Identität eine neue Bedeutung zu geben ist ein Bruch. Legen Sie Identifikatoren still; recyceln Sie sie nicht.

- **Unbekannte Felder sollten toleriert werden.** Ob ein Decoder Felder, die er nicht kennt, behält, ignoriert oder ablehnt, hängt vom Format und seinen Einstellungen ab. Finden Sie das heraus, bevor Sie darum herum entwerfen.

Wenn Sie JSON ohne Schemasprache betreiben, erhalten Sie trotzdem den größten Teil des Nutzens, indem Sie Schemadateien in der Versionsverwaltung halten, in den Tests des Producers validieren und echte Beispielereignisse pro Version aufbewahren. Die Registry fügt zentrales Nachschlagen und Durchsetzung hinzu. Das Nachdenken nimmt sie Ihnen nicht ab.

Wo die Version steht

Der Standard-Nachrichtenumschlag trägt `schema_version` als Ganzzahl neben `event_type`, und ein Consumer liest das Paar, um zu wissen, welche Form er erwarten muss. Steht sie im Umschlag, können ein Router, ein Metrik-Zähler oder ein DLQ-Tool sie sehen, ohne den Payload zu dekodieren. „Wie viele Ereignisse mit `schema_version: 1` treffen noch ein?“ wird zu einer billigen Abfrage statt zu einem Projekt.

Zwei Zahlen lassen sich hier leicht verwechseln, deshalb gibt dieses Kapitel ihnen getrennte Namen. Die **Breaking Version** ist die `schema_version` des Umschlags. Sie ändert sich nur bei einer Änderung, die ein alter Reader nicht verarbeiten kann: eine Umbenennung, eine Entfernung, eine Typänderung oder eine Bedeutungsänderung. Die **Registry-Revision** ist der Identifikator, den die Registry jedem registrierten Schema zuweist, und sie ändert sich bei jeder registrierten Änderung, auch bei additiven. Ein Consumer lehnt ein Ereignis ab, dessen Breaking Version er nicht kennt. Ein Ereignis mit neuerer Registry-Revision, aber bekannter Breaking Version liest er als die neueste Form, die er kennt, und ignoriert, was er nicht erkennt. Keine von beiden ist die Aggregatversion aus Kapitel 4, die Änderungen an einer einzelnen Bestellung zählt und mit der Form des Payloads nichts zu tun hat.

Würde jede additive Änderung `schema_version` erhöhen, würde jeder noch nicht aktualisierte Consumer jedes neue Ereignis ablehnen, und die Vorwärtskompatibilität wäre dahin. Der Preis der Trennung sind zwei Zahlen, die man auseinanderhalten muss. Ein Beispiel für dieselbe Linie ist BabelQueue (babelqueue.com), ein quelloffener, sprachunabhängiger Standard für Nachrichtenumschläge, der seinen Umschlag bei

`schema_version` 1 einfriert, neue Felder nur als optionale und ignorierbare zulässt, für eine Umbenennung, eine Entfernung oder eine Typänderung eine Erhöhung der Version verlangt, Consumer eine ihnen unbekannte Version ablehnen lässt und dafür sorgt, dass unbekannte Felder die Verarbeitung nicht stören und beim erneuten Veröffentlichen der Nachricht erhalten bleiben.

Ein Ereignis, wie Payments es während des Dual-Publish-Schritts der unten beschriebenen Migration veröffentlicht:

```
{
  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"
  }
}
```

Kompatibilität über Sprachen hinweg

Ein Schema kann nach den Regeln der Registry vollkommen kompatibel sein und trotzdem zwischen zwei Sprachen brechen, weil die Regeln die Struktur beschreiben und der Bruch darin liegt, wie eine Laufzeitumgebung sie liest. Kapitel 11 hat die Kodierung des Umschlags festgelegt. Hier folgen die Fallen im Payload und in dem, was ein Consumer mit Feldern tut, die er nicht kennt.

Unbekannte Felder gehen beim erneuten Veröffentlichen verloren. Angenommen, ein Node-Consumer in Notifications deserialisiert jedes `PaymentCaptured` in eine Klasse mit `order_id`, `captured_minor` und `currency` und kopiert das Ereignis in einen Topic für Partner. Wenn Payments später `payment_method` hinzufügt, lässt die Kopie es stillschweigend fallen. Nichts schlägt fehl. Das Feld ist einfach nie da. Binäre Protobuf-Decoder behalten unbekannte Felder in der Regel; viele typisierte

JSON-Mapper tun es nicht. Regel: Ein Hop, der eine Nachricht weiterreicht, bewahrt unbekannte Felder, indem er den rohen Payload oder eine generische Map weiterträgt, oder er veröffentlicht ein neues eigenes Ereignis unter seinem eigenen Vertrag.

Zahlen. `captured_minor` passt bequem in die Ganzzahl jeder Sprache. Ein int64-Identifikator oder -Zähler passt jenseits von 2^53 - 1 nicht in eine JSON-Zahl von JavaScript, und der Consumer rundet ihn ohne Fehler. Regel: Dokumentieren Sie den Wertebereich jedes numerischen Felds, und senden Sie alles, was ihn überschreiten kann, sowie jede Dezimalzahl als String.

Zeitstempel. Die Avro-Typen `timestamp-millis` und `timestamp-micros` halten dieselbe Art von Wert in verschiedenen Maßstäben. Ein Writer in Mikrosekunden, gelesen als Millisekunden, liefert ein Datum zehntausende Jahre entfernt und lässt sich trotzdem parsen. Regel: Legen Sie die Einheit im Schema fest und ändern Sie sie nie; eine Änderung der Einheit ist eine Breaking Version.

Enumerationen. Generierter Code behandelt einen unbekannten Wert unterschiedlich: Manche behalten ihn, manche bilden ihn auf einen Standardwert ab, manche werfen eine Exception. Regel: Behandeln Sie jede Enumeration als offen und entscheiden Sie im Consumer, was „unbekannt“ bedeutet.

Null und fehlend. Viele Decoder unterscheiden die beiden nicht, und manche füllen die Lücke mit einem Standardwert. Regel: Verwenden Sie für Felder, bei denen das Fehlen eine Rolle spielt, die Formen, die das Vorhandensein unterscheiden, aus dem Aufzählungspunkt zu den Standardwerten oben.

Nichts davon braucht eine bestimmte Version einer bestimmten Bibliothek. Alles davon braucht einen Menschen, der mit einem Consumer in jeder Sprache testet, die Parcelly betreibt, und das sind Kosten, die die Vertragstests aus Kapitel 15 tragen können.

Versionierungsstrategien

Es gibt drei Wege, eine Änderung einzuführen, von billig bis teuer. Ich würde sie der Reihe nach versuchen und erst eine Sprosse höher steigen, wenn die niedrigere die Änderung nicht tragen kann.

Strategie 1: ein optionales Feld hinzufügen

Wenn die Änderung etwas Neues ist, das alte Reader ignorieren dürfen, fügen Sie ein optionales Feld hinzu. `payment_method` zu `PaymentCaptured` hinzuzufügen ist der Musterfall. Es registriert eine neue Registry-Revision, und die Kompatibilitätsregel entscheidet, ob die Registry sie akzeptiert. `schema_version` ändert sich nicht, weil kein alter Reader etwas verliert. Die Breaking Version für eine additive Änderung zu erhöhen würde jeden noch nicht aktualisierten Consumer die neuen Ereignisse ablehnen lassen, was den Zweck verfehlt, das Feld kompatibel hinzuzufügen.

Diese Strategie kann keine Umbenennung, keine Typänderung, keine Entfernung eines Pflichtfelds und keine Bedeutungsänderung ausdrücken. Steigen Sie dafür eine Stufe höher.

Strategie 2: einen neuen Ereignistyp einführen

Wenn sich die Bedeutung ändert, sagen Sie das mit einem neuen Namen. Angenommen, Parcelly behandelt einen Teileinzug als anderen fachlichen Fakt als einen vollständigen Einzug. Das ehrliche Modell ist ein neues Ereignis, benannt nach den Regeln aus Kapitel 11. Dieses Kapitel führt `PaymentPartiallyCaptured` als zusätzliches Ereignis ein.

Das alte Ereignis behält seine Bedeutung, und seine Consumer arbeiten weiter. Der Preis ist, dass nun beide Ereignisse existieren und ein Consumer, der „alle Einzüge“ will, beide abonnieren muss. Das ist der richtige Tausch, wenn der neue Fakt konzeptionell ein anderer ist, und der falsche, wenn es derselbe Fakt in besserer Form ist. Dieser zweite Fall ist ein Darstellungsproblem, und dafür passt ein Upcaster besser.

Strategie 3: ein versionierter Topic

Wenn eine inkompatible Änderung den gesamten Stream betrifft, legen Sie die neue Form auf einen neuen Topic, zum Beispiel `payments.payment-events.v2`. Consumer migrieren in ihrem eigenen Tempo, und der alte Topic wird stillgelegt, wenn sein letzter Reader geht. Beim oben beschriebenen Aufbau trägt der neue Topic jeden Ereignistyp des Aggregats.

Das ist die klarste Option und die teuerste. Partitionsschlüssel, Reihenfolge und Offsets der beiden Topics haben nichts miteinander zu tun, deshalb muss ein Consumer, der von einem zum anderen wechselt, entscheiden, wo er zu lesen beginnt und wie er vermeidet, einen Fakt doppelt zu verarbeiten. Die Regeln zur Reihenfolge aus Kapitel 9 gelten für jeden Topic einzeln, und nichts ordnet Ereignisse über beide hinweg. Bei einem warteschlangenbasierten Broker ist das Äquivalent eine zweite Warteschlange mit eigenen Bindings, eigener Dead-Letter-Behandlung und eigenen Alarmen. Zu einem versionierten Topic würde ich nur bei einem echten Bruch greifen, den ein Upcaster nicht überbrücken kann, und nur, wenn ich benennen kann, wer den alten löscht.

Upcasting und der Tolerant Reader

Zwei Techniken lassen Consumer mit mehreren Versionen zurechtkommen, ohne den Producer zum Einfrieren zu zwingen.

Upcasting

Ein **Upcaster** wandelt ein Ereignis, das unter einem alten Schema geschrieben wurde, in das aktuelle Schema um, bevor die Geschäftslogik es sieht. Der Geschäftscode wird gegen eine einzige Form geschrieben, und die Umwandlung für jede historische Version liegt an einer Stelle.

```
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
```

Der Kommentar am Zweig für Version 1 trägt die eigentliche Arbeit. Ein Upcaster kodiert eine Aussage über die Vergangenheit, und diese Aussage muss wahr sein. Wenn manche alten Ereignisse in der Praxis Teileinzüge waren, meldet der Upcaster stillschweigend einen Teilbetrag als vollständigen Einzug. Testen Sie Upcaster mit echten archivierten Ereignissen, nicht mit erfundenen.

`UnsupportedSchemaVersion` braucht eine eigene Klassifizierung. Für den aktuellen Code ist sie dauerhaft, da ein Retry nichts ändert, aber ein Deployment des Consumers behebt sie. Sie wird also für einen Redrive zurückgehalten, nachdem der Consumer aktualisiert wurde (Kapitel 8), und nicht in den Parking Lot gelegt, wo sie auf einen Menschen warten würde, der nichts zu entscheiden hat.

Die Seite des Consumers, im Handler vor der Geschäftslogik, ist der einfachste Ort, um anzufangen. Die Historie in einen neuen Topic umzuschreiben ist nützlich, wenn viele Consumer sonst denselben Upcaster bräuchten. Gespeicherte Ereignisse an Ort und Stelle umzuschreiben ist eine andere Entscheidung: Event-Sourcing-Systeme lehnen das oft ab, weil das Log ein Nachweis ist. Ich würde das respektieren, es sei denn, es gibt einen rechtlichen oder sicherheitstechnischen Grund, und dann würde ich es als Datenmigration mit Plan durchführen.

Der Tolerant Reader

Ein **Tolerant Reader** entnimmt einem Ereignis nur, was er braucht, und ignoriert den Rest. Er liest `order_id` und `captured_minor` und kümmert sich nicht darum, dass es auch ein `payment_method` gibt. Er ist großzügig bei dem, was er akzeptiert, und streng bei dem, wovon er abhängt.

```
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
    }
```

Toleranz hat zwei Seiten. Unbekannte Felder zu ignorieren ist sicher, wenn Producer etwas hinzufügen. Einen fehlenden Betrag stillschweigend durch null zu ersetzen ist gefährlich. Der Reader oben weigert sich weiterzumachen, wenn ein Feld fehlt, von dem er abhängt, und er fragt nach dem Vorhandensein, nicht nach dem Wert. Das ist nur dann ehrlich, wenn der Decoder ein fehlendes Feld von einer Null unterscheiden kann, und deshalb sind die Standardwerte oben wichtig. Diese Weigerung ist eine Klassifizierung: Das Ereignis kann nie erfolgreich sein, nimmt also den Pfad für dauerhafte Ausfälle aus Kapitel 6 und endet als Dead Letter, wie Kapitel 8 es beschreibt.

> **Hinweis:** Toleranz ist eine Vereinbarung zwischen beiden Seiten. Der Producer verspricht, hinzuzufügen und die Bedeutung nicht zu ändern. Der Consumer verspricht, zu ignorieren, was er nicht versteht, und zu verweigern, was er nicht sicher tun kann. Wenn eine Seite ihre Hälfte bricht, kann die andere das System nicht retten.

Die Registry prüft den Producer, bevor er veröffentlicht. Der Tolerant Reader schützt den Consumer, nachdem etwas durchgerutscht ist. Ich würde mich nicht auf das eine anstelle des anderen verlassen.

Eine inkompatible Änderung migrieren

Manchmal lässt sich ein Bruch nicht vermeiden: Die alte Form ist falsch, eine Vorschrift zwingt ein Feld hinaus, oder zwei Services müssen gemeinsam umgebaut werden. Die Antwort besteht darin, ein inkompatibles Deployment in mehrere kompatible zu zerlegen. Das ist das **Expand-and-Contract**-Pattern, angewandt auf Ereignisse: den Vertrag erweitern, sodass er beide Formen unterstützt, jeden Reader migrieren und dann den Vertrag zurückbauen, indem man die alte Form entfernt.

Ein Migrationsplan für `PaymentCaptured`

Angenommen, Parcelly muss tatsächlich `amount_minor` in `captured_minor` umbenennen, und sonst ändert sich nichts am Ereignis. Der Wert behält seine Bedeutung: der für die gesamte Bestellung eingezogene Betrag. Das ist eine reine Umbenennung, und der folgende Plan ist nur deshalb sicher. Teileinzüge sind eine Bedeutungsänderung und werden nach den Schritten gesondert behandelt.

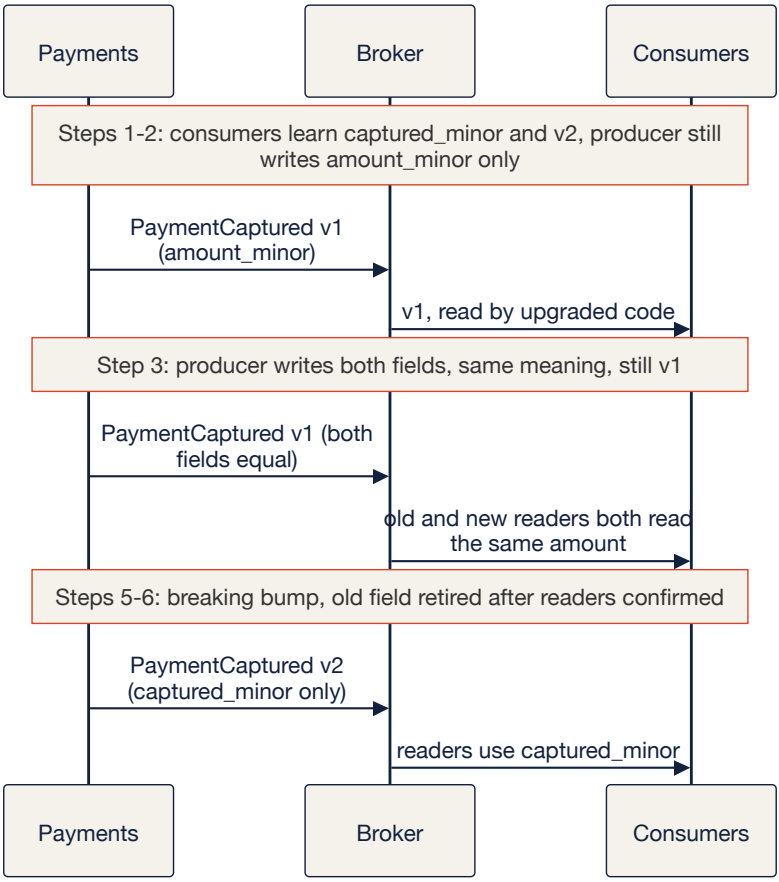
1. **Die Ergänzung registrieren.** Die neue Registry-Revision behält `amount_minor` und fügt `captured_minor` als optionales Feld hinzu, dessen Standardwert „kein Wert“ ist, nicht 0. Unter einer Regel für Abwärtskompatibilität akzeptiert die Registry sie. `schema_version` bleibt 1, weil kein alter Reader etwas verliert.
2. **Zuerst die Consumer aktualisieren.** Jeder Consumer lernt, `captured_minor` zu lesen und auf `amount_minor` zurückzufallen, wenn es fehlt. Er lernt außerdem, die Breaking Version 2 zu lesen, die es noch nicht gibt, damit die Erhöhung in Schritt 5 niemanden findet, der sie ablehnt. Neue Reader existieren vor den neuen Daten.
3. **Dual-Publish.** Payments schreibt in jedes Ereignis beide Felder, weiterhin mit `schema_version` 1, oder veröffentlicht, wenn die Änderung einen neuen Ereignistyp oder Topic braucht, die alte und die neue Form desselben Fakts. Das ist nicht der Dual Write aus Kapitel 4: Beide Nachrichten stammen aus derselben Transaktion der Transactional Outbox, es gilt also die übliche Garantie.
4. **Beobachten.** Messen Sie, wer noch das alte Feld oder den alten Topic liest. Metriken des Producers zu `schema_version` zeigen, was noch geschrieben wird; die Aktivität der Consumer-Group auf dem alten Topic, die Consumer-Liste aus Kapitel 11 und die Vertragstests aus Kapi-

tel 15 zeigen, wer noch liest. Ein Consumer, der nie umzieht, ist ein Gespräch, das geführt werden muss, keine Zahl, die man ignoriert.

5. **Das Schreiben der alten Form beenden.** Payments veröffentlicht nur noch `captured_minor` und erhöht `schema_version` auf 2, denn das ist der eine Schritt, den ein alter Reader nicht übersteht. Ein Consumer, der Schritt 2 verpasst hat, lehnt das Ereignis ab, und es geht in die DLQ für einen Redrive, nachdem dieser Consumer aktualisiert wurde (Kapitel 8). Schritt 4 dient dazu, diese Liste leer zu halten. `amount_minor` bleibt als abgekündigt im Schema, bis das älteste Ereignis, das es trägt, die Aufbewahrung verlassen hat.
6. **Zurückbauen.** Entfernen Sie das alte Feld und legen Sie den alten Topic still, falls es einen gab. Entfernen Sie den Fallback aus den Consumern erst, wenn das älteste Ereignis ohne `captured_minor` die Aufbewahrung verlassen hat, oder verschieben Sie ihn in einen dauerhaften Upcaster.

Teileinzüge bleiben außerhalb dieses Plans. Sobald ein Einzug partiell sein kann, hat `amount_minor` keinen ehrlichen Wert mehr. Trägt es den eingezogenen Betrag, hat sich seine Bedeutung an Ort und Stelle geändert, und ein Reader, der Schritt 2 übersprungen hat, ob unwissentlich oder durch einen Rollback, verbucht den Teilbetrag, als wäre er der Gesamtbetrag der Bestellung, ohne jeden Fehler. Trägt es den Gesamtbetrag der Bestellung, verbucht derselbe Reader zu viel. Die Registry lässt beide Versionen durch, weil die Struktur unverändert ist. Das ist die Lehre: Eine strukturelle Kompatibilitätsprüfung kann eine Bedeutungsänderung nicht sehen. Legen Sie die neue Bedeutung in ein neues Feld, einen neuen Ereignistyp (Strategie 2, `PaymentPartiallyCaptured`) oder eine neue Hauptversion auf einem neuen Topic. Veröffentlichen Sie dann Teileinzüge gar nicht, bis jeder Reader auf die neue Form umgestiegen ist, und danach nur noch in der neuen Form.

Die Reihenfolge ist das Herz des Plans. Bei einer abwärtskompatiblen Änderung kommen die Consumer zuerst; bei einer vorwärtskompatiblen dürfen die Producer zuerst gehen. Wenn Sie nicht sagen können, was zutrifft, führen Sie den Plan wie geschrieben aus und behalten Sie eine Weile beide Felder.



Dual-Publish über Topics hinweg

Wenn der Plan einen versionierten Topic verwendet, veröffentlicht Schritt 3 den Fakt zweimal, einmal in jeden Topic. Ein Consumer, der beide liest, darf denselben fachlichen Fakt nicht zweimal verarbeiten. Ich würde den Kopien verschiedene Werte von `event_id` geben, weil es verschiedene Nachrichten sind, und die ID des Originals in einem dokumentierten Feld mitführen. Wofür Sie sich auch entscheiden: Der Idempotenzschlüssel des Consumers für diesen Fakt muss bewusst definiert sein. Kapitel 7 behandelt die Schlüssel; hier geht es darum, dass eine Migration genau der Moment ist, in dem Annahmen zur Idempotenz getestet werden.

> **Warnung:** Lassen Sie während des Dual-Publish nie einen Consumer sowohl den alten als auch den neuen Stream lesen und beide anwenden. Ein Consumer mit zwei Quellen für einen Fakt braucht eine Deduplizierungsregel, aufgeschrieben, sonst verbucht er einen Buchungssatz doppelt.

Rollbacks und Historie

Ein Rollback gehört zum Plan. Wenn Ereignisse der Version 2 bereits veröffentlicht sind, entfernt ein Rollback des Producers sie nicht aus dem Log. Ein Consumer, der auf Code zurückgesetzt wurde, der Version 2 nicht lesen kann, scheitert nun an Ereignissen, die bereits existieren. Consumer müssen also Version 2 lesen können, bevor irgendein Producer sie schreibt, und das Ziel eines Consumer-Rollbacks muss jede Version in der Aufbewahrung lesen können, Version 2 eingeschlossen. Code, der nur Version 1 liest, ist ein Rollback-Ziel, das man meiden sollte, auch wenn es das letzte Release vor der Änderung war.

Die Abwärtskompatibilität behandelt alte Ereignisse, ohne sie anzufassen. Wenn Sie den Fallback-Code löschen wollen, haben Sie drei Möglichkeiten: den Upcaster behalten, bis die alten Ereignisse ablaufen, die gespeicherten Ereignisse in einen neuen Topic umschreiben oder den Zustand anderswo als Snapshot sichern und die Historie verfallen lassen. Jede kostet etwas: dauerhaften Code, eine Migration, der man vertrauen muss, oder die Fähigkeit zum Replay. Welche Ihr System wählen sollte, weiß ich nicht, ohne zu wissen, wie oft Sie einen Replay ausführen und wie lange Ihre Aufsichtsbehörden erwarten, dass Sie Aufzeichnungen aufbewahren.

Ereignisse abkündigen

Ereignisse lassen sich leichter hinzufügen als entfernen, denn ein Consumer, von dem Sie nichts wissen, kann von jedem davon abhängen. Ein vernünftiger Lebenszyklus hat vier Stufen.

1. **Ankündigen.** Markieren Sie das Ereignis in der Registry oder im Ereigniskatalog als abgekündigt, nennen Sie den Ersatz, geben Sie ein Stilllegungsdatum und eine Kontaktperson an.

2. **Messen.** Finden Sie heraus, wer es noch liest: Mitgliedschaft in Consumer-Groups bei einem Log, Bindings bei einer Warteschlange, Registry-Abfragen. Behandeln Sie die Liste als Indiz, nicht als Beweis, denn ein monatlicher Batch-Job kann heute fehlen und nächste Woche da sein.
3. **Die Erzeugung beenden.** Hören Sie nach dem Datum auf zu veröffentlichen.
4. **Stilllegen.** Entfernen Sie das Schema und den Topic oder die Warteschlange, sobald die Aufbewahrung das letzte Ereignis hat altern lassen und die Reader gegangen sind.

Zwei Warnungen. Eine Abkündigung ohne Frist endet nie, schreiben Sie das Datum also auf und halten Sie es ein. Und ein abgekündigtes Ereignis, das nach der Stilllegung per Replay verarbeitet wird, schlägt fehl, entscheiden Sie also vor der Stilllegung, was der Replay für den betroffenen Zeitraum tun soll.

Die Verantwortung zählt. Das Team des Producers besitzt das Ereignis und seine Abkündigung, wie Kapitel 11 beschrieben hat. Die Consumer besitzen ihre Migrationen. Eine Richtlinie, die den Producer allein entscheiden lässt oder die Consumer unbegrenzt verweigern lässt, wird scheitern. Die brauchbare Richtlinie ist die Ankündigung durch den Producer und ein Datum, das die Consumer einhalten.

Abwägungen und Kosten

- **Eine Registry ist ein weiterer Service, den man betreiben muss.** Sie liegt im Pfad des Deployments jedes Producers und, je nach Aufbau, der Dekodierung der Consumer. Sie braucht Verfügbarkeit, Zugriffskontrolle und eine verantwortliche Person, und Clients brauchen einen Plan für den Fall, dass sie nicht erreichbar ist.
- **Kompatibilitätsregeln bremsen Producer aus, absichtlich.** Ein Team, das ein Feld an einem Nachmittag hätte umbenennen können, führt jetzt eine Migration in sechs Schritten durch. Das ist der Preis für unabhängige Deployments.
- **Upcaster sammeln sich an.** Jede historische Version fügt einen Zweig und einen Test hinzu, und der Code wird nicht einfacher, bis alte Er-

eignisse die Aufbewahrung verlassen. Unbegrenzte Aufbewahrung bedeutet unbegrenzte Upcaster.

- **Tolerant Reader verbergen Fehler.** Ein Consumer, der unbekannte Felder ignoriert, bemerkt auch nicht, wenn ein Producer anfängt, etwas zu senden, das ihn interessieren sollte. Toleranz ohne Monitoring verwandelt laute Ausfälle in stillen Drift.
- **Dual-Publish vervielfacht Verkehr und Deduplizierungsaufwand**, solange die Migration läuft.
- **Zwei Versionsnummern statt einer.** Die Breaking Version und die Registry-Revision verhindern, dass additive Änderungen Consumer scheitern lassen, und kosten ein Team den Aufwand, sie in Dashboards und Gesprächen auseinanderzuhalten.
- **Versionsnummern sind keine Dokumentation.** `schema_version: 4` sagt, dass sich etwas geändert hat, nicht was oder wer betroffen ist. Ein Änderungsprotokoll, das beim Schema liegt, ist mehr wert als die Zahl.
- **Regeln für mehrere Sprachen kosten Komfort.** Strings für große Zahlen, das Bewahren des rohen Payloads und feste Zeitstempeleinheiten bedeuten zusätzlichen Umwandlungscode in jeder Sprache und einen Test pro Sprache.

Wann diese Patterns nicht einzusetzen sind

Bauen Sie nicht die volle Maschinerie für ein Ereignis mit einem Producer, einem Consumer und einem Team, das beide gemeinsam ausrollt. Ein koordiniertes Deployment ist billiger, und das Risiko, gegen das Sie sich absichern würden, besteht nicht. Prüfen Sie erneut, wenn ein zweiter Consumer auftaucht, besonders einer, der einem anderen Team gehört.

Behandeln Sie den strikten Modus einer Registry nicht als Ersatz für Design. Eine Folge sicherer Ergänzungen kann ein Ereignis mit 40 optionalen Feldern und ohne klaren Zweck hinterlassen. Wenn ein Ereignis immer weiter wächst, sind es vermutlich zwei Ereignisse, und die Regeln zur Granularität aus Kapitel 11 greifen.

Legen Sie nicht für jede Änderung einen versionierten Topic an. Ein System, das das tut, endet mit vielen halb stillgelegten Streams und niemandem, der weiß, welcher aktuell ist.

Häufige Fehler

- **Eine Umbenennung als kleine Änderung behandeln.** Das ist verlockend, weil der neue Name sauberer ist und die Tests durchlaufen. Für jeden Reader ist es eine Entfernung plus eine Hinzufügung. Fügen Sie den neuen Namen neben dem alten hinzu, migrieren Sie die Reader und entfernen Sie dann den alten.
- **Die Bedeutung eines Felds an Ort und Stelle ändern.** Name und Typ bleiben, also beanstandet kein Tool etwas. Geben Sie der neuen Bedeutung ein neues Feld oder einen neuen Ereignistyp.
- **Ein Pflichtfeld ohne Standardwert hinzufügen.** Es fühlt sich wie eine Ergänzung an, aber alten Ereignissen fehlt es. Machen Sie es mit einem Standardwert optional, oder führen Sie es mit einer neuen Version und einem Upcaster ein.
- **Die Kompatibilität nur gegen die neueste Version testen.** Ein Replay liest alles in der Aufbewahrung. Prüfen Sie gegen jede noch vorhandene Version und halten Sie Beispielereignisse aus jeder in den Consumer-Tests bereit (Kapitel 15).
- **Producer vor Consumern aktualisieren.** Das Team des Producers besitzt die Änderung und ist als Erstes bereit. Bei einer abwärtskompatiblen Änderung kommen die Consumer zuerst. Bestätigen Sie die Reihenfolge vor dem Deployment.
- **Bei unbekannten Feldern laut scheitern.** Ein strikter Parser macht jede additive Änderung zu einem Incident. Ignorieren Sie, was Sie nicht verwenden; lehnen Sie nur ab, wovon Sie abhängen.
- **Ein fehlendes Feld stillschweigend mit einem Standardwert füllen.** Null für einen fehlenden Betrag einzusetzen verbirgt einen Defekt, und ein Decoder, der 0 einfüllt, verbirgt ihn, bevor Ihr Code läuft. Verwenden Sie für solche Felder einen Standardwert „kein Wert“, prüfen Sie das Vorhandensein und lösen Sie einen Fehler aus, damit der Ausfallpfad das Ereignis für einen Menschen zurückhält.
- **`schema_version` bei jeder Änderung erhöhen.** Dann wird jede additive Änderung für jeden noch nicht aktualisierten Consumer zu einem harten Ausfall. Erhöhen Sie die Breaking Version nur für Änderungen, die ein alter Reader nicht verarbeiten kann, und lassen Sie die Registry-Revision den Rest zählen.

- **Über eine typisierte Klasse weiterleiten.** Die Kopie lässt jedes Feld fallen, das die Klasse nicht deklariert, und nichts beschwert sich. Bewahren Sie unbekannte Felder beim erneuten Veröffentlichen, oder veröffentlichen Sie ein neues Ereignis unter Ihrem eigenen Vertrag.
- **Eine Feldnummer oder einen Topic-Namen wiederverwenden.** Eine recycelte Nummer liest alte Daten als das neue Feld, und ein wiederverwendeter Topic vermischt zwei Formen. Legen Sie Identifikatoren still.
- **Ohne Datum oder verantwortliche Person abkündigen.** Das Ereignis bleibt bestehen, und das nächste Team scheut sich, es zu entfernen. Geben Sie jeder Abkündigung ein Datum, einen Ersatz und eine Person, die Fragen beantwortet.
- **Einen Consumer zurücksetzen, ohne das Schema zu prüfen.** Stellen Sie sicher, dass das Rollback-Ziel jede Version in der Aufbewahrung lesen kann.

Zentrale Erkenntnisse

- Ein Ereignisschema ist ein Vertrag mit Readern, die Sie nicht kontrollieren, und mit Daten, die vor langer Zeit geschrieben wurden; Abwärtskompatibilität heißt, dass neue Reader alte Daten lesen können, Vorwärtskompatibilität, dass alte Reader neue Daten lesen können, und ein Log mit Replay braucht Abwärtskompatibilität gegenüber jeder Version, die noch in der Aufbewahrung liegt.
- Ein optionales Feld hinzuzufügen ist sicher, während Umbenennungen, Typänderungen, neue Pflichtfelder und Bedeutungsänderungen inkompatibel sind, und eine Bedeutungsänderung ist die schlimmste, weil nichts laut fehlschlägt; nur eine inkompatible Änderung ändert `schema_version`, und die Registry-Revision zählt den Rest.
- Eine Schema-Registry verdient ihren Platz, indem sie die Kompatibilität in der Deployment-Pipeline durchsetzt, aber sie prüft die Struktur und nicht die Bedeutung, sodass weiterhin ein Mensch prüft, was ein Feld aussagen soll.
- Bevorzugen Sie die billigste Versionierungsstrategie, die die Änderung trägt: zuerst ein optionales Feld, einen neuen Ereignistyp, wenn die Bedeutung ein anderer Fakt ist, und einen versionierten Topic nur, wenn der gesamte Stream brechen muss.

- Upcaster und Tolerant Reader lassen Consumer mit mehreren Versionen zurechtkommen, vorausgesetzt, der Upcaster kodiert eine wahre Aussage über die Vergangenheit und der Reader prüft das Vorhandensein statt des Werts und verweigert, was er nicht sicher interpretieren kann.
- Über Sprachen hinweg sind die stillen Brüche verlorene unbekannte Felder, große Ganzzahlen, Zeitstempeleinheiten, geschlossene Enums und der Unterschied zwischen null und fehlend. Bewahren Sie also, was Sie nicht kennen, und schreiben Sie die Darstellung auf.
- Migrieren Sie eine inkompatible Änderung als Folge kompatibler Änderungen, mit der Erhöhung an dem einen Schritt, den ein alter Reader nicht übersteht, und planen Sie die Rollbacks im selben Dokument.

Ein Vertrag, der sich sicher weiterentwickelt, beruhigt nur, wenn Sie sehen können, was durch das System fließt, und damit beginnt Kapitel 13.

TEIL V

Betrieb und Nachweis

Kapitel 13 — Observability und Abgleich

Stellen Sie sich einen Donnerstagmorgen bei Parcelly vor. Der Support leitet ein Ticket weiter: Ein Käufer hat vor zwei Tagen eine Bestellung bezahlt, die Kartenabrechnung zeigt die Belastung, und die Bestellseite steht trotzdem auf „Awaiting confirmation“. Der Support öffnet die Orders-Adminoberfläche und sieht `PAYMENT_PENDING`. Er fragt das Payments-Team, das in seiner Datenbank eine Zeile `PaymentCaptured` und eine saubere Outbox findet. Er fragt das Orders-Team, in dessen Consumer-Logs zu dieser Bestellung kein Fehler steht. Alle Dashboards sind grün.

Drei Engineers durchsuchen nun die Logs von drei Services nach der Bestellnummer. Eine Logzeile verwendet `order=81724`, eine andere `81724`, und der Payments-Relay protokolliert nur seine interne Zeilen-ID. Nach einer Stunde finden sie das Ereignis in der Payments-Outbox, als veröffentlicht markiert. Nach einer weiteren Stunde erfahren sie, dass der Broker-Topic, der Orders speist, eine Consumer-Group hatte, die während eines Deployments zurückgefallen war. Die Aufbewahrung des Brokers hatte eine Strecke von Nachrichten gelöscht, die die Group nicht gelesen hatte, und die Offset-Reset-Richtlinie der Group war auf den Sprung zur neuesten Position gesetzt, sodass sie hinter der Lücke weitermachte, ohne einen Fehler auszulösen. Das Ereignis existierte, wurde veröffentlicht und nie gelesen.

Nichts davon war ein Codefehler, den ein Reviewer beanstanden würde. Das System tat, wofür es gebaut worden war, und niemand konnte ihm dabei zusehen. Schlimmer noch: Das Ticket existiert, weil ein Käufer es bemerkt hat, bevor irgendein Alarm anschlug, und niemand weiß, wie viele andere Bestellungen genauso aussehen. Dieses Kapitel stellt zwei Fragen. Was muss ein Team sehen können, um „Wo ist diese Bestellung?“ in Minuten zu beantworten, und wie findet ein Team die Bestellungen, über die sich noch niemand beschwert hat?

Das Problem und warum es schwierig ist

In einem synchronen System kommt ein Fehler meist mit einem Stack-Trace an einer Anfrage an. In einem ereignisgesteuerten System verteilt sich die Arbeit eines fachlichen Ablaufs über mehrere Services, mehrere Datenbanken, einen Broker und eine Reihe von Zeitpunkten. Kein einzelner Prozess kennt die ganze Geschichte, und kein einzelnes Log enthält sie.

Drei Eigenschaften machen das schwierig. Erstens: **Die Hops sind zeitlich entkoppelt.** Der Producer ist fertig und vergisst, und der Consumer läuft womöglich Sekunden oder Stunden später, in einem anderen Prozess, ohne den Call-Stack des Producers. Werkzeuge, die eine Anfrage durch verschachtelte Aufrufe verfolgen, verlieren sie an der ersten Nachricht.

Zweitens: **Die schlimmsten Ausfälle sind still.** Kapitel 3 hat ungelesene Nachrichten beschrieben, die ein Broker löscht, wenn ein Consumer über die Aufbewahrung hinaus zurückfällt. Kapitel 8 hat Nachrichten beschrieben, die fehlerhaft verarbeitet wurden, ohne einen Fehler auszulösen. In beiden Fällen scheitert nichts, also alarmiert nichts. Ein System kann ein Ereignis, das nie angekommen ist, nur bemerken, wenn es weiß, dass es hätte ankommen sollen.

Drittens: **Gesund aussehende Komponenten können zusammen einen kaputten Ablauf ergeben.** Der Relay läuft, der Broker ist oben, der Consumer meldet keine Fehler, und eine Bestellung hängt trotzdem fest. Die Gesundheit einer Komponente beantwortet „Lebt dieser Prozess?“, und das Business fragt „Wurde diese Bestellung abgeschlossen?“. Das sind verschiedene Fragen, und in der Lücke dazwischen arbeitet dieses Kapitel.

Kapitel 3 hat die Semantik eines Ausfalls als drei Fragen beschrieben: was ausfallen kann, wie man es beobachten würde und welche Garantie ihn übersteht. Dieses Kapitel handelt von der zweiten Frage und davon, was zu tun ist, wenn die Antwort lautet: „gar nicht“.

Der Ansatz hat zwei Hälften. Observability gibt Ihnen die Mittel, einem Ablauf zu folgen und Druck entstehen zu sehen. Der **Abgleich**, also das Vergleichen der Sicht zweier Systeme auf die Wahrheit und das Beheben der Unterschiede, gibt Ihnen einen Weg, das zu finden, was kein Signal gemeldet hat. Ein Team mit nur der ersten Hälfte sieht die lauten Ausfälle. Ein Team mit nur der zweiten findet Probleme Tage zu spät. Ich würde beides aufbauen.

Kontext über asynchrone Hops tragen

Jedes Ereignis trägt bereits zwei Identifikatoren aus Kapitel 11: `correlation_id`, geteilt von allen Ereignissen eines fachlichen Ablaufs, und `causation_id`, die auf den direkten Auslöser zeigt. Sie bilden das Rückgrat der Geschichte. Distributed Tracing fügt einen zweiten Faden hinzu, der das Timing und die Eltern-Kind-Beziehung zwischen Arbeitseinheiten festhält.

Ein **Trace** ist ein Baum zeitlich erfasster Spans, jeder eine Arbeitseinheit in einem Prozess. Innerhalb eines Prozesses hält die Tracing-Bibliothek den aktuellen Span im Umgebungszustand. Über eine Nachricht hinweg verschwindet dieser Zustand, wenn man ihn nicht in die Nachricht legt. Der Standardansatz besteht darin, den Trace-Kontext, eine Trace-ID, eine Span-ID und ein Sampling-Flag, in Nachrichten-Header zu serialisieren. W3C Trace Context definiert dafür ein gemeinsames Format mit zwei Headern, `traceparent` und `tracestate`, und die meisten Tracing-Bibliotheken folgen ihm. Header-Kapazität und Weitergabe unterscheiden sich je nach Broker und Bridge: Manche begrenzen die Anzahl oder Größe von Headern, und manche Bridges verwerfen sie, prüfen Sie also den Weg, den Ihre Nachrichten tatsächlich nehmen.

Wo der Kontext erfasst wird, spielt eine größere Rolle, als es zunächst scheint. Kapitel 4 hat genau dafür eine Spalte `headers` in die Outbox aufgenommen. Der Relay läuft später in einem anderen Prozess ohne Umgebungs-Span. Ein Relay, der den Kontext beim Veröffentlichen einfügt, würde also seinen eigenen Trace anhängen statt den, der die Bestellung aufgegeben hat. Der Kontext muss erfasst werden, wenn die Outbox-Zeile geschrieben wird, innerhalb der Transaktion des Producers.

```
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)
```

Auf der Consumer-Seite gibt es zwei Wege, den neuen Span mit dem alten in Beziehung zu setzen. Ein Consumer-Span kann ein Kind des Producer-Spans sein, was einen einzigen Baum zeichnet. Oder er beginnt einen neuen Trace und hält eine **Verknüpfung** auf den Kontext des Producers. Die richtige Wahl hängt vom Fall ab.

Fall	Beziehung	Warum	Was es kostet
Einzelne Nachricht, zeitnah konsumiert	Kind des Producer-Spans	Ein Wasserfall von der Anfrage bis zur Wirkung	Bei verspäteter Zustellung bleibt der Trace für die gesamte Verzögerung offen, und die Dauern führen in die Irre
Batch oder Fan-in	Neuer Trace, eine Verknüpfung pro Producer	Ein Span hat nur einen Parent, ein Batch-Consumer kann also nicht Kind von zwölf Producern sein	Das Backend muss Verknüpfungen unterstützen, und jemand muss sie nachverfolgen
Fan-out	Ein Producer-Span, pro Subscriber ein Consumer-Kind-Span	Der Baum bleibt ein Baum	Der Trace wächst mit jedem hinzugefügten Subscriber
Replay, Redrive oder verspätete Zustellung	Neuer Trace mit Verknüpfung	Ein stundealter Parent würde den ursprünglichen Trace dehnen	Die Sampling-Entscheidung wird erneut getroffen, sodass ein Producer-Trace, der in der Stichprobe landete, auf einen Consumer-Trace verweisen kann, der nicht darin landete

Ich würde trotzdem standardmäßig Verknüpfungen verwenden, denn Verspätung und erneute Zustellung sind der Normalfall, für den dieses Buch plant. Der Preis ist der durchgehende Wasserfall, den die meisten Tracing-Oberflächen für Eltern-Kind-Beziehungen zeichnen. Für einen Consumer, der immer innerhalb von Augenblicken nach seinem Producer läuft, ist die Eltern-Kind-Beziehung die bessere Wahl, und ich würde sie nehmen.

```
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
```

Beachten Sie, dass der Span `event_id` und den Zustellversuch festhält. Logbasierte Broker führen keinen Zustellzähler, und nur manche warteschlangenbasierten Broker legen einen offen, entscheiden Sie also, woher die Zahl kommt, bevor Sie sich auf sie verlassen. Wird eine Nachricht erneut zugestellt, erhalten Sie einen zweiten Span mit derselben `event_id`. Das Paar sagt Ihnen, dass die Nachricht zweimal zugestellt wurde, und das unten beschriebene Feld `outcome` sagt Ihnen, ob sie zweimal angewendet wurde.

Tracing hat Grenzen, und ich würde darauf achten, die Garantie nicht darauf zu stützen. Backends arbeiten mit Sampling, der Trace für die eine Bestellung, die Sie interessiert, existiert also womöglich nicht. Kontext-Header können von einer Bridge, einer Dead-Letter-Kopie oder einem handgeschriebenen Producer verworfen werden. Eine Möglichkeit, nicht von Headern abhängig zu sein, ist, den Trace-Identifikator im Body zu tragen: Der BabelQueue-Umschlag (babelqueue.com), ein quelloffener, sprachunabhängiger Standard, trägt eine `trace_id` im Nachrichten-Body, die der erste Producer erzeugt, spätere Producer unverändert kopieren und die das Dead-Lettering übersteht, sodass die Spur auch hält, wenn ein Hop Header verwirft. Ein Trace, der an einer Outbox-Zeile endet, erreicht nie einen Consumer, der die Zeile nicht gelesen hat. Deshalb zählen die Identifikatoren im Umschlag mehr als die Tracing-Bibliothek: `correlation_id` muss in jeder Logzeile und jeder Datenbankzeile stehen, die das Ereignis verarbeitet, und dann funktioniert sie auch bei abgeschaltetem Tracing.



Lesen Sie das Diagramm auf das, was sich nicht ändert: Die `correlation_id` bleibt an jedem Hop gleich, während jeder Service einen frischen Trace-Kontext in seiner eigenen Outbox speichert.

Die Geschichte einer Bestellung rekonstruieren

Sind die Identifikatoren vorhanden, wird aus der Frage „Wo ist Bestellung 81724?“ eine Abfrage. Die billigste Umsetzung ist eine Konvention statt eines Produkts: Jeder Service schreibt pro verarbeitetem und pro veröffentlichtem Ereignis eine strukturierte Logzeile mit `correlation_id`, `event_id`, `event_type`, `causation_id`, Ergebnis und Zeitstempel als Feldern. Die Suche in diesen Logs nach `correlation_id = "order-81724"` liefert Ihnen jeden Hop in jedem Service.

```
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
}
```

Das Feld `outcome` verdient Aufmerksamkeit. Verwenden Sie eine kleine geschlossene Menge: `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` und `dead_lettered`. Kapitel 9 hat argumentiert, dass ein ignoriertes veraltetes Ereignis eine Spur verdient, und Kapitel 7 behandelt die Inbox-Prüfung, die `duplicate_skipped` erzeugt. Sind diese Ergebnisse erst Felder, wird „Wie viele Ereignisse hat dieser Consumer in der letzten Stunde als veraltet verworfen?“ zu einer Zählung statt zu einer Untersuchung.

Die Sortierung nach `occurred_at` ergibt eine Zeitleiste, und das Verfolgen von `causation_id` ergibt einen Baum: `OrderPlaced` verursachte `PaymentAuthorized`, das zusammen mit `StockReserved` `CapturePayment` verursachte, und so weiter. Im Szenario vom Anfang hätte der Baum `PaymentCaptured` als von Payments veröffentlicht gezeigt und keine Zeile `consumed` in Orders. Der erste fehlende Hop ist die Stelle, an der man

nachsehen muss. Das ist nützlicher als jede einzelne Logzeile: Eine Lücke in der Kette ist ein Beleg dafür, dass etwas verloren ging, während eine lange Kette nur zeigt, dass der Ablauf vorangekommen ist.

Wenn Sie eine Saga aus Kapitel 10 betreiben, ist ihre Zeile `saga_instance` eine zweite Quelle. Sie beantwortet „Was glaubt der Orchestrator, welchen Zustand die Saga hat?“, und die Logs beantworten „Was ist tatsächlich passiert?“. Wenn beide voneinander abweichen, haben Sie etwas gefunden, das ein Ticket wert ist.

Für die wichtigsten Abläufe würde ich außerdem ein **Ereignisjournal** führen: eine Append-only-Aufzeichnung von Ereignisumschlägen, jeweils mit einem Hash des Payloads statt des Payloads selbst, die jeder Service neben seiner Inbox schreibt. Das kostet Speicher und etwas Schreiblast, und dafür können Sie Fragen beantworten, nachdem der Broker die Ereignisse vergessen hat, etwa ob ein neu aufgebautes Ereignis noch mit dem Original übereinstimmt (siehe den Abschnitt zur Reparatur unten). Halten Sie die Aufbewahrung so lang, dass Ihr langsamster Abgleich durchlaufen kann.

Metriken, die zählen

Metriken sagen Ihnen etwas über Druck, bevor eine einzelne Nachricht fehlschlägt. Die Liste nützlicher Metriken in einem ereignisgesteuerten System ist kurz, und auf die meisten wurde in früheren Kapiteln schon hingewiesen. Ich würde mit diesen anfangen.

Signal	Was es misst	Warum es wichtig ist
Consumer-Lag (Nachrichten)	Wie weit der Consumer hinter dem Kopf liegt	Größe des Backlogs; nützlich für die Kapazität
Alter der Nachricht, Consumer-Seite	Zeit seit der Erzeugung der ältesten unverarbeiteten Nachricht	Was das Business erlebt
Outbox-Alter	Alter der ältesten unveröffentlichten Zeile	Gesundheit des Relays (Kapitel 5)
Ende-zu-Ende-Latenz	Zeit vom ersten `occurred_at` des Ablaufs bis zur Wirkung im letzten Consumer	Die tatsächliche Deadline des Ablaufs
Retry-Rate und Versuchszahlen	Anteil der Nachrichten, die mehr als einen Versuch brauchen	Durch Retries verdeckte Verschlechterung (Kapitel 6)

Signal	Was es misst	Warum es wichtig ist
DLQ-Zugänge und -Alter	Neue Dead Letters und die älteste ungeprüfte	Ausfälle, die einen Menschen brauchen (Kapitel 8)
Ignorierte und doppelte Nachrichten	Verworfene veraltete Ereignisse, übersprungene Duplikate	Druck durch Reihenfolge und erneute Zustellung
Abflussrate und Aufholzeit	Verarbeitungsrate im Verhältnis zur Ankunftsrate; Lag geteilt durch die Netto-Abflussrate	Ob sich ein Backlog vor seiner Deadline oder dem Aufbewahrungslimit auflöst (Kapitel 6)

Lag, und warum das Alter der Nachricht besser ist

Der **Consumer-Lag** in Nachrichten ist die Zahl, die die meisten Dashboards zuerst zeigen, und sie lässt sich am leichtesten falsch lesen. Ein Lag von 10.000 Nachrichten bedeutet für sich genommen wenig: Baut der Consumer 5.000 pro Sekunde ab, liegt er zwei Sekunden zurück, bei 10 pro Sekunde eine Viertelstunde. Der Lag in Nachrichten beantwortet eine Kapazitätsfrage. Der Lag in Zeit beantwortet die Frage des Business.

Das **Alter der Nachricht** ist die Zeit seit der Erzeugung der ältesten Nachricht, die noch wartet. Manche warteschlangenbasierten Broker legen das Alter der ältesten Nachricht direkt offen; bei den anderen leiten Sie es aus einem Zeitstempel der ältesten wartenden Nachricht ab. Bei einem logbasierten Broker leiten Sie es aus dem Zeitstempel des Records am festgeschriebenen Offset des Consumers ab. In beiden Fällen ist die Zahl in Sekunden angegeben und lässt sich daher mit einer vom Business gewählten Deadline vergleichen.

Der Zeitstempel, von dem Sie abziehen, spielt eine Rolle, denn die Kandidaten messen Verschiedenes. Die Append-Zeit des Brokers misst, wie lange eine Nachricht am Broker gewartet hat, also die Gesundheit des Consumers. `occurred_at` misst die Zeit seit dem Eintreten des Fakts, einschließlich der Zeit, die eine Zeile in der Outbox auf den Relay wartete, also die Sicht des Business auf Verzögerung. Die Erstellungszeit des Producers hängt vom Aufbau ab: Je nach Konfiguration speichert ein Log-Broker entweder sie oder die Append-Zeit, und ein Relay, der eine alte Zeile erneut veröffentlicht, erhält eine frische Erstellungszeit, sofern er nicht ausdrücklich eine setzt. Ich würde die Append-Zeit für das Consumer-Alter-Gauge verwenden und `occurred_at` für die Ende-zu-Ende-Latenz. Wer `occurred_at` für das Gauge nimmt, lässt einen gesunden Consumer in

dem Moment sehr verspätet aussehen, in dem der Relay eine alte Zeile erneut veröffentlicht. Das Alter einer Group ist das Maximum über ihre Partitionen, und Alarme sollten dieses Maximum verwenden.

Eine Falle verdient einen Namen. Eine Partition ohne neuen Verkehr hat einen Lag von null und bei naiver Berechnung ein Alter von null oder ein Alter, das ewig wächst. Entscheiden Sie, wie sich eine ruhende Partition meldet, und testen Sie es. Dieselbe Funktion muss den weiter unten beschriebenen Aufbewahrungsausfall überstehen: Ein festgeschriebener Offset, der hinter den Anfang des Logs zurückgefallen ist, zeigt auf gelöschte Daten, die Prüfung kommt also zuerst und wird für sich gemeldet. Ein Consumer, der wie in Kapitel 8 an einem Poison Record hängt, zeigt null Durchsatz und ein steigendes Alter, und diese Kombination sollte jemanden alarmieren.

```
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)
```

Die Aufbewahrung ist die Deadline, die Sie nicht sehen

Das Alter der Nachricht hat eine zweite, weniger offensichtliche Aufgabe. Bei einem logbasierten Broker verliert ein Consumer, der um mehr als die Aufbewahrungsdauer zurückfällt, still Nachrichten: Der Broker löscht Daten, die nie gelesen wurden, und der Consumer macht dort weiter, wohin

ihn seine Offset-Reset-Richtlinie setzt. Bei einem warteschlangenbasierten Broker passiert dasselbe, wenn Nachrichten ablaufen. Kapitel 3 hat diesen Ausfall benannt, und kein Fehler begleitet ihn.

Diese Reset-Richtlinie ist eine Entscheidung, und jede Option hat einen Preis. Das Zurücksetzen auf den frühesten verfügbaren Offset spielt wieder ab, was noch übrig ist, der Verlust ist also teilweise, und der Consumer muss eine Flut alter Nachrichten verkraften. Das Zurücksetzen auf den neuesten Offset springt nach vorn und macht den Verlust still, was im Eröffnungsszenario passiert ist. Zu scheitern, sodass der Consumer anhält und auf einen Menschen wartet, macht aus einem stillen Verlust einen Incident. Für jeden Ablauf, bei dem eine Lücke Geld kostet, würde ich das Scheitern wählen und in Kauf nehmen, dass der Consumer ausfällt, bis jemand entscheidet. Kapitel 14 behandelt, wie man sich von der Lücke erholt, sobald man sie bemerkt hat: auf einen Zeitstempel zurücksetzen und mit unterdrückten Seiteneffekten per Replay verarbeiten.

Die Abwehr ist Arithmetik. Vergleichen Sie das Alter der Nachricht mit der Aufbewahrungs- oder Ablaufdauer und alarmieren Sie, wenn das Alter einen Bruchteil davon erreicht, sagen wir die Hälfte. Zu diesem Zeitpunkt liegt die Deadline noch vor Ihnen, und ein Mensch hat Zeit, Consumer hinzuzufügen, einen Producer anzuhalten oder die Aufbewahrung zu verlängern. Nach Ablauf der Deadline ist der einzige Beleg das Ausbleiben von Ereignissen, und genau das aufzuspüren ist der Zweck des Abgleichs. Ich würde das Verhältnis auf dasselbe Dashboard legen wie das Alter selbst, denn ein Lag, der für das Business akzeptabel und für die Aufbewahrung fatal ist, ist eine reale Situation.

Für einen CDC-Connector ist das entsprechende Signal, wie viel vom Log der Datenbank der Connector noch nicht konsumiert hat, im Vergleich dazu, wie viel die Datenbank für ihn aufzubewahren bereit ist. Ein Connector, der länger ausfällt, als die Datenbank ihr Log vorhält, braucht womöglich einen frischen Snapshot zur Erholung, und das ist ein Ausfall für sich.

Latenz, Retries und der Rest

Die **Ende-zu-Ende-Latenz** braucht eine Definition, bevor sie einen Graphen braucht. Ich würde ab `occurred_at` des ersten Ereignisses eines Ablaufs bis zu dem Moment messen, in dem die letzte Wirkung festgeschrie-

ben ist, zum Beispiel von `OrderPlaced` bis `OrderConfirmed`. Berichten Sie Perzentile, keinen Durchschnitt: Der langsame Ausläufer sind die Käufer, die dem Support schreiben. Der letzte Consumer sieht nur den Umschlag, den er konsumiert hat, und dieser Umschlag trägt die Zeit seines eigenen Fakts, nicht den Beginn des Ablaufs. Reichen Sie entweder die Startzeit des Ablaufs in einem Header weiter, oder schlagen Sie sie über `correlation_id` im Ereignisjournal oder im zuständigen Service nach. Beschriften Sie die Messung dann mit dem letzten `event_type`. Beide Zeitstempel stammen von den Uhren verschiedener Services, erwarten Sie also einen gewissen Versatz im Ergebnis; er ist klein gegenüber einer fachlichen Deadline und ein Grund, bei Unterschieden unter einer Sekunde nicht zu alarmieren.

Die **Abflussrate** ist die beobachtbare Form des Backpressure. Teilen Sie den Lag durch die Differenz zwischen der Rate, mit der der Consumer Nachrichten verarbeitet, und der Rate, mit der sie ankommen, und Sie erhalten die Aufholzeit. Übersteigen die Ankünfte die Verarbeitung, löst sich der Lag nie auf, und das ist die Zahl, auf die man alarmieren sollte, lange bevor das Alter das Aufbewahrungslimit erreicht. Kapitel 3 hat Backpressure definiert, und Kapitel 6 behandelt, wie man die Arbeit begrenzt, die ein Consumer übernimmt.

Retry-Rate und Versuchszahlen machen das Problem sichtbar, das Kapitel 6 verdeckte Ausfälle nannte: eine Abhängigkeit, die sich seit Wochen verschlechtert hat, während Retries sie leise auffangen. Das Histogramm der Versuchszahlen ist aufschlussreicher als die Fehlerzahl, denn eine Nachricht, die immer beim dritten Versuch gelingt, erzeugt überhaupt keine Fehler.

DLQ-Zugänge und das Alter des ältesten ungeprüften Dead Letters stammen aus Kapitel 8. Dazu kommen die **Zähler für Duplikate und veraltete Ereignisse** aus den Kapiteln 7 und 9. Ein steigender Duplikatzähler direkt nach einem Deployment ist normal, da die Zuständigkeit für Arbeit wechselt, und ein steigender Zähler zu einer ruhigen Stunde deutet auf langsame Handler hin. Bei einem warteschlangenbasierten Broker ist die übliche Ursache ein Visibility-Timeout, der kürzer ist als die Laufzeit des Handlers. Bei einem logbasierten Broker ist es ein Rebalance oder ein Consumer, der seinen Poll-Timeout überschreitet und seine Partitionen verliert.

Alarme, die etwas bedeuten

Das Versagen der meisten Alarmierungsaufbauten liegt nicht an zu wenigen Daten. Die Daten alarmieren Menschen wegen Ursachen statt wegen Wirkungen, und nach ein paar Wochen Rauschen traut niemand mehr dem Pager. Ich würde Alarme an einem Maßstab messen: Wenn dieser um drei Uhr nachts auslöst, kann eine namentlich bekannte Person sagen, was zu tun ist.

Alarmieren Sie auf **Symptome, die an eine fachliche Deadline gebunden sind.** „Die älteste unveröffentlichte Outbox-Zeile ist älter als zwei Minuten“ ist ein Symptom dafür, dass zwischen einer Zustandsänderung und ihrer Bekanntgabe etwas feststeckt. „Die CPU des Consumers liegt über 80 Prozent“ ist eine Ursache, und sie kann relevant sein oder nicht. Verwenden Sie die Ursachenmetriken auf Dashboards, auf denen jemand untersucht, und behalten Sie den Pager für die Symptome.

Geben Sie jedem Alarm einen **Verantwortlichen und einen Schwellenwert mit Begründung.** Kapitel 8 hat denselben Punkt für DLQs gemacht: Eine verlorene E-Mail und ein verlorener Einzug verdienen unterschiedliche Schwellenwerte, und der Unterschied sollte neben dem Alarm aufgeschrieben sein. Die Begründung zählt, weil Schwellenwerte driften, und eine Zahl ohne genannten Grund wird beim ersten Ärgernis hochgesetzt.

Alarmieren Sie auf **Abwesenheit ebenso wie auf Anwesenheit.** Ein Relay, der abstürzt, meldet keinen Fehler; er meldet nichts. Eine Consumer-Group, die alle Mitglieder verloren hat, zeigt Lag, aber ein Topic, der gar keine Ereignisse mehr empfängt, zeigt weder Lag noch Fehler. Für Abläufe mit gleichmäßigem Rhythmus, etwa Bestellungen während der Geschäftszeiten, fängt ein Alarm wie „Seit 10 Minuten keine Ereignisse `OrderPlaced` beobachtet“ eine Stille auf der Producer-Seite ab, die jedes andere Signal übersieht. Der Schwellenwert muss dem Verkehrsmuster folgen, und für Abläufe ohne Rhythmus ist es das falsche Werkzeug.

Schließlich genügen den meisten Teams zwei Stufen: eine Alarmierung für alles, was eine Deadline bedroht oder Daten verliert, und ein Ticket für langsamen Druck wie eine wachsende DLQ mit geringwertigen Nachrichten. Eine dritte Stufe wird tendenziell ignoriert.

> **Warnung:** Einen Alarmschwellenwert anzuheben, um das Rauschen zu stoppen, ist die Art, wie ein Team ein Ärgernis gegen einen blinden Fleck eintauscht. Wenn ein Alarm zu oft auslöst, beheben Sie seine Ursache oder seine Definition. Heben Sie die Zahl nicht an, bis er nicht mehr auslöst.

Abgleich

Metriken und Traces beobachten, was das System tut. Sie können Ihnen nichts sagen über ein Ereignis, das nie ins System gelangt ist, über einen Handler, der ohne Fehler den falschen Zustand festgeschrieben hat, oder über eine Reparatur, die jemand von Hand eingespielt hat. Dafür vergleichen Sie das System mit einer unabhängigen Aufzeichnung dessen, was wahr sein sollte. Das ist der Abgleich.

Ein Abgleichs-Job nimmt zwei Sichten auf dieselben Fakten und meldet die Unterschiede. Die Paare liegen meist auf der Hand, sobald man nach ihnen sucht:

- **Orders gegen Payments:** Jede Bestellung im Zustand `PAID` hat eine eingezogene Zahlung, und jede eingezogene Zahlung gehört zu einer Bestellung, die davon weiß.
- **Payments gegen Ledger:** Jedes `PaymentCaptured` hat ein passendes `LedgerEntryPosted` über denselben Betrag in derselben Währung.
- **Ihre Aufzeichnungen gegen die eines Anbieters:** der Abrechnungsbericht des Zahlungsanbieters gegen Ihre eigenen Einzugsdatensätze. Kapitel 7 hat für Effekte, die sich nicht idempotent machen lassen, hierher verwiesen, wo der Abgleich die letzte Verteidigungslinie ist.
- **Ein Read-Modell gegen seine Quelle:** ein Suchindex oder eine View, die Kapitel 9 zufolge als verdächtig markiert und später repariert werden konnte.

Drei Arten von Unterschieden decken die meisten Fälle ab: **fehlend** (die Quelle hat es, die abgeleitete Seite nicht), **veraltet** (beide haben es, aber die abgeleitete Seite weicht ab, meist weil sie hinterherhinkt) und **überzählig** (die abgeleitete Seite hat etwas, von dem die Quelle nichts weiß). Jede weist auf eine andere Ursache hin. Fehlend deutet auf verlorene oder

nie konsumierte Ereignisse, veraltet auf Reihenfolge oder Lag oder hin und wieder auf einen Handler, der den falschen Wert geschrieben hat, und überzählig auf einen Bug oder eine manuelle Änderung.

Sorgfalt verdient der Umstand, dass Ereignisse im Flug genau wie Abweichungen aussehen. Eine Zahlung, die vor vier Sekunden eingezogen wurde und noch nicht im Ledger steht, ist normal. Ein Job, der sie meldet, erzeugt Rauschen und bringt den Leuten bei, ihn zu ignorieren. Deshalb vergleicht der Job nur Datensätze, die älter sind als eine **Schonfrist**, die aus Ihrer normalen Ende-zu-Ende-Latenz gewählt wird, und er bestätigt eine Abweichung ein zweites Mal, bevor er sie meldet.

```
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)
```

Mehrere Details dieses Blocks tragen die Korrektheit. Das Fenster ist begrenzt und der Stichtag angewandt, der Job liest also Daten, die Zeit hatten, sich zu setzen. Jeder Batch ist ein Lookup über Schlüssel statt eines Joins über Datenbanken hinweg, weil die Services von Parcelly nicht gegenseitig ihre Tabellen lesen; der Job holt die andere Seite über die eigene Schnittstelle dieses Service oder einen veröffentlichten Auszug. Der Job zeichnet Abweichungen nur auf und ändert nichts. Und weil er per Upsert in eine nach Element und Art geschlüsselte Abweichungstabelle schreibt, ergibt zweimaliges Ausführen, oder nach einem Absturz mitten in einem Fenster, dieselbe Tabelle, ohne einen wachsenden Haufen doppelter Zeilen. Dieselbe Funktion schließt Abweichungen, sobald die Seiten übereinstimmen, und genau diesen Schritt setzt der Reparaturcode unten voraus. Eine Zahlung, die zweimal ins Ledger gebucht wurde, fängt der Vorwärtsdurchlauf ab, der die Treffer zählt. Der Rückwärtsdurchlauf findet einen „überzähligen“ Eintrag ohne Einzug dahinter, etwa eine Buchung für eine Zahlung, die nie eingezogen wurde, und eine Prüfung in nur einer Richtung würde ihn nie finden.

Wählen Sie die Taktung nach dem Schaden, den eine Abweichung anrichtet. Payments gegen Ledger ist finanziell und verdient einen häufigen Lauf, mit einem täglichen vollständigen Vergleich als Rückhalt. Ein Suchindex braucht womöglich einen nächtlichen Lauf. Auch die Gesundheit des Jobs selbst zählt: Alarmieren Sie, wenn er nicht in seinem erwarteten Fenster abgeschlossen wurde, denn ein Abgleich, der still stehen geblieben ist, ist schlimmer als keiner, weil die Leute glauben, er wache.

> **Hinweis:** Ein Abgleichs-Job ist ein Consumer der Daten, die er prüft, und er kann sich irren. Behandeln Sie eine Abweichung zunächst als Anhaltspunkt für eine Untersuchung. Die ersten Wochen eines neuen Jobs gehen überwiegend dafür drauf, False Positives zu finden, die durch Timing, Rundung oder eine Vergleichsregel entstehen, die nicht zum Business passt.

Reparatur und Backfill

Eine Abweichung zu finden ist die halbe Arbeit. Die andere Hälfte besteht darin, den Zustand so zu ändern, dass er übereinstimmt, ohne eine zweite Abweichung zu erzeugen.

Die Regel, an die ich mich halten würde, lautet: **Reparaturen gehen denselben Weg wie normale Änderungen.** Fehlt im Ledger ein Eintrag für eine eingezogene Zahlung, fügen Sie die Zeile nicht mit einer Datenbankkonsole ein. Kapitel 3 hat Operator-Aktionen als Teil des Ausfallmodells behandelt, und eine von Hand bearbeitete Zeile sendet kein Ereignis, sodass jeder andere Consumer dieses Fakts falsch bleibt und der nächste Abgleich eine Inkonsistenz meldet, die Sie selbst herbeigeführt haben. Veröffentlichen Sie stattdessen den Fakt erneut, oder senden Sie das Command, das ihn auslöst, über die Outbox des Producers, damit die Consumer ihn so behandeln wie alles andere.

Eine Reparatur kann wiederholt, unterbrochen und von zwei Engineers zweimal ausgeführt werden, und sie muss in jedem Fall sicher bleiben. Die Idempotenz aus Kapitel 7 beantwortet das nur, wenn das Reparaturereignis die **ursprüngliche** `event_id` trägt. Ein Reparaturereignis mit einer neuen ID ist für jeden Consumer ein anderes Ereignis. War das erste Ereignis lediglich verzögert, kommt es nach der Reparatur an, und das Ledger bucht die Zahlung zweimal. Die Reparatur veröffentlicht also den Fakt erneut, der bereits existiert: derselbe Umschlag, mit derselben `event_id` und `causation_id`, und die Inbox behandelt die Kopie, die als zweite ankommt, als Duplikat.

Das setzt voraus, dass der Producer die ursprüngliche ID behält. Am einfachsten speichert man die `event_id` zusammen mit einem Hash des Payloads beim Schreiben der Outbox-Zeile am fachlichen Datensatz (hier der Einzug). Existiert die Outbox-Zeile noch, wird sie als unveröffentlicht markiert, und der Relay sendet sie erneut. Wurde sie bereinigt, wird der Umschlag aus dem Datensatz mit der gespeicherten ID neu aufgebaut.

Der Neuaufbau hat eine Grenze. Der Datensatz ist der aktuelle Zustand, und das Ereignis ist ein Fakt aus der Vergangenheit. Hat sich der Datensatz seither geändert, sagen wir, eine Teilerstattung hat den Betrag angepasst, weicht ein daraus aufgebauter Payload vom Original ab, obwohl er dieselbe `event_id` trägt. Deshalb wird der Hash gespeichert: Die Reparatur baut den Payload neu auf, vergleicht seinen Hash mit dem gespeicherten und hält für einen Menschen an, wenn beide voneinander abweichen. Ein Datensatz ohne gespeicherte ID oder Hash, meist einer aus der Zeit vor dieser Gewohnheit, muss auf eine aus seinem fachlichen Schlüssel abgeleitete ID und auf unveränderliche Felder ausweichen, und dann muss der Consu-

mer zusätzlich durch einen Unique Constraint auf diesem Schlüssel geschützt sein, wie Kapitel 7 beschreibt. Ich würde diesen Constraint dem Ledger ohnehin als zweite Verteidigungslinie hinzufügen.

Eine erneute Veröffentlichung **fächert** außerdem **auf** (Fan-out). Der Topic liefert den Umschlag an jeden Subscriber, nicht nur an das Ledger. Ein Subscriber, dessen Inbox die `event_id` noch enthält, überspringt ihn. Einer, dessen Inbox-Zeile bereinigt wurde, weil das Ereignis älter ist als sein Deduplizierungsfenster (Kapitel 7), wendet ihn ein zweites Mal an, und Notifications könnte dieselbe E-Mail noch einmal verschicken. Es gibt also zwei sichere Wege. Ist das Ereignis jünger als das kürzeste Deduplizierungsfenster unter allen Subscribern, veröffentlichen Sie es wie oben über die Outbox des Producers erneut. Ist es älter, liefern Sie die Reparatur nur an den Consumer, der es verpasst hat, über dessen eigene Reparatur-Queue oder den eigenen Reparatur-Topic oder indem Sie die Offsets dieser einen Group zurücksetzen, und verlassen sich als Schutz auf einen Constraint auf dem fachlichen Schlüssel. Kapitel 8 wendet dasselbe Prinzip auf den Redrive an.

```
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
```

Zwei Details tragen die Sicherheit. Eine unveröffentlichte Zeile erneut als unveröffentlicht zu markieren ist bei Wiederholung harmlos, und die Unique-Verletzung durch ein zweites gleichzeitiges Einfügen wird als „bereits erledigt“ behandelt statt als Fehler. Der erneut veröffentlichte Umschlag behält außerdem seine ursprüngliche `causation_id`, sodass die Kausalkette wahrheitsgemäß bleibt, und die Verbindung zur Abweichung liegt im Reparaturdatensatz. Der letzte Kommentar ist der wichtige. Die Reparaturfunktion markiert die Abweichung nicht als gelöst. Das tut der nächste Abgleichslauf, wenn beide Seiten übereinstimmen. So wird verhindert, dass eine Reparatur, die veröffentlicht und dann verloren ging, als behoben gemeldet wird.

Die Massenreparatur, meist **Backfill** genannt, bringt Größenordnung ins Spiel. Ein Backfill von 30.000 Ereignissen in einen laufenden Consumer konkurriert mit dem normalen Verkehr und kann die Latenz des Ablaufs über seine Deadline treiben. Führen Sie ihn in kleinen Batches aus, mit Ratenbegrenzung und einem Pausenschalter, wie Kapitel 8 es für den Redrive empfohlen hat, und beobachten Sie währenddessen Alter der Nachricht und Consumer-Lag. Bei einem logbasierten Broker ziehen Sie womöglich einen bewussten Replay in eine separate Consumer-Group vor, die eine abgeleitete View neu aufbaut und dann umschaltet. Das ist sauberer, wenn der Consumer eine Projektion ist, und falsch, wenn der Consumer Seiteneffekte wie den Versand von E-Mails hat.

Die Reparatur über den Producer passt für eine Handvoll Datensätze. Ist der Schaden größer, etwa wenn eine Consumer-Group eine ganze Aufbewahrungslücke übersprungen hat, bei einer wiederhergestellten Datenbank oder einem verlorenen Broker-Cluster, ist die erneute Veröffentlichung Datensatz für Datensatz nicht mehr das richtige Werkzeug. Kapitel 14 behandelt Wiederherstellung per Replay, Katastrophen und Broker-Migration.

Jede Reparatur braucht eine Aufzeichnung: wer sie ausgeführt hat, welche Abweichungen sie abdeckte, was sie veröffentlicht hat und wann. Der Prüfpfad ist der Weg, auf dem der nächste Engineer erfährt, dass die Spitze bei den Duplikaten um 14:10 ein Backfill war, und der Weg, auf dem eine Aufsichtsbehörde oder das Finanzteam später erfährt, dass ein Ledger-Eintrag eine Korrektur war.

Fachliche Invarianten

Der Abgleich vergleicht zwei Systeme. Eine **Invariantenprüfung** vergleicht ein System mit einer Regel, die das Business für wahr hält. Invarianten sind oft billiger auszuwerten und fangen Probleme ab, die kein zweites System finden könnte, etwa einen Bug, der dafür sorgt, dass zwei Systeme sich auf die falsche Antwort einigen.

Für Parcelly zeigen ein paar Beispiele die Form:

- Keine Bestellung ist `CONFIRMED` ohne eine eingezogene Zahlung.
- Keine Reservierung bleibt länger gehalten als ihre Deadline.

- Keine Saga-Instanz sitzt über ihr `deadline_at` hinaus in `AWAITING_RESULTS`.
- Jede eingezogene Zahlung hat genau einen Ledger-Eintrag, nicht null und nicht zwei.
- Die Summe der Ledger-Einträge einer Bestellung gleicht sich zu null aus.

Jede davon ist eine Abfrage, die keine Zeilen zurückgeben sollte. Führen Sie sie nach Zeitplan aus, und behandeln Sie jede zurückgegebene Zeile als Abweichung mit einem Verantwortlichen. Das Saga-Beispiel ist das, das Kapitel 10 offen gelassen hat: Der Timeout-Handler ist die erste Absicherung gegen eine eingefrorene Bestellung, und eine Invariantenabfrage auf hängende Instanzen ist die zweite, denn der Handler, der hätte auslösen sollen, kann selbst gescheitert sein.

```
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)
```

Die Schlüsselung nach Invariante und Element bedeutet, dass eine fortbestehende Verletzung aktualisiert statt dupliziert wird, und `first_seen_at` verrät, wie lange ein Problem offen ist. Dieses Alter ist selbst eine Metrik, auf die zu alarmieren sich lohnt.

Ich würde Invarianten mit den Produkt- und Finanzleuten schreiben statt für sie. Engineers wissen, was das System kann, und wissen nicht immer, was das Business als Defekt ansieht. Eine Regel wie „Eine Erstattung übersteigt nie den Einzug“ ist für das Finanzteam selbstverständlich und steht womöglich nirgends im Code.

Abwägungen und Kosten

Observability und Abgleich sind nicht umsonst, und sie laufen nicht von selbst.

Telemetrie hat einen Preis. Strukturierte Logs für jedes verarbeitete Ereignis vervielfachen das Logvolumen, und Labels mit hoher Kardinalität wie `order_id` an Metriken können ein Metrik-Backend überfordern. Halten Sie Identifikatoren in Logs und Traces, und beschränken Sie Metrik-Labels auf kleine Mengen wie Service, Topic und Ereignistyp. Sampeln Sie Traces, und nehmen Sie hin, dass der Trace, den Sie brauchen, womöglich nicht existiert.

Ein Ereignisjournal dupliziert Daten. Es lohnt sich für Abläufe, bei denen „Was ist passiert?“ rechtliches oder finanzielles Gewicht hat. Für einen geringwertigen Ablauf wie eine kosmetische Benachrichtigung genügen die Logs, und das Journal ist ein Kostenpunkt ohne Leser.

Der Abgleich liest Produktionsdaten. Ein Vergleichsjob kann die Quelldatenbank belasten, und einer, der beim Batching nachlässig ist, kann den Service verlangsamen, den er schützt. Führen Sie ihn gegen ein Replikat aus, wo Sie eines haben, und begrenzen Sie Batchgröße und Rate. Er fügt außerdem eine Abhängigkeit hinzu: Ist die Schnittstelle der anderen Seite unten, kann der Job nicht laufen.

Alarme kosten Aufmerksamkeit. Jede Alarmierung verbraucht die Glaubwürdigkeit des nächsten. Eine kleine Menge gut verantworteter Alarme leistet mehr als eine große, die niemand liest.

Reparatur hat ein eigenes Risiko. Ein Reparaturmechanismus ist ein Weg, den Produktionszustand außerhalb des normalen Anfrageflusses zu ändern. Ohne Audit, Freigaben für die gefährlichen Arten und idempotentes Design ersetzt er eine Klasse von Problemen durch eine schlimmere.

Wann man das alles nicht aufbauen sollte: Ein System mit einem Service und einem Consumer, geringem Einsatz und kurzen Abläufen braucht kein Ereignisjournal, keine Invariantentabellen und keinen nächtlichen Abgleich. Strukturierte Logs mit `correlation_id` und ein Alarm auf das Alter der Nachricht sind das Minimum, das ich in jeder Größenordnung beibehalten würde. Der Rest wächst mit den Kosten eines stillen Ausfalls. Kostet ein verlorenes Ereignis eine Ledger-Abweichung, bauen Sie zuerst den Abgleich. Kostet es ein veraltetes Suchergebnis, bauen Sie ihn zuletzt.

Häufige Fehler

- **Auf Gesundheit alarmieren statt auf den Ablauf.** Grüne CPU- und Speichergraphen sagen, dass die Prozesse leben, und nichts darüber, ob Bestellungen abgeschlossen werden. Es ist verlockend, weil Infrastrukturmetriken schon vorhanden sind. Fügen Sie die Symptomalarme hinzu: Alter der Nachricht, Outbox-Alter und Invariantenverletzungen.
- **Den Lag nur in Nachrichten beobachten.** Eine Nachrichtenzahl hat keine Deadline. Rechnen Sie in Alter um, und vergleichen Sie das Alter mit der fachlichen Deadline und mit der Aufbewahrung.
- **Trace-Kontext im Relay einfügen.** Der Relay hat keinen Umgebungs-Span, hängt also den falschen an oder keinen. Erfassen Sie den Kontext beim Schreiben der Outbox-Zeile und leiten Sie ihn unverändert weiter.
- **Uneinheitliche Identifikatoren.** Ein Service protokolliert `order-81724`, ein anderer `81724`. Die Suche nach Korrelation scheitert dann genau dann, wenn man sie braucht. Definieren Sie das Format einmal im Ereignisdesign und erzwingen Sie es mit einem gemeinsamen Logging-Helper. Behalten Sie dieses eine Format auch für synthetischen Verkehr bei: Kapitel 15 markiert Probes mit einem separaten Header und nie mit einer besonderen `correlation_id`.
- **Einen Zeitstempel für jedes Alter verwenden.** Erstellungszeit des Producers, Append-Zeit des Brokers und `occurred_at` messen verschiedene Verzögerungen. Verwenden Sie die Append-Zeit für das Consumer-Alter-Gauge und `occurred_at` für die Ende-zu-Ende-Latenz.
- **Eine Reparatur an jeden Subscriber erneut veröffentlichen.** Ein Consumer, dessen Inbox-Zeile bereinigt wurde, wendet das Ereignis erneut an. Prüfen Sie das Ereignis gegen das kürzeste Deduplizierungsfenster, oder liefern Sie die Reparatur nur an den einen Consumer, der es verpasst hat.
- **Ohne Schonfrist abgleichen.** Der Job meldet jedes Ereignis im Flug als Abweichung, das Team lernt, ihn zu ignorieren, und der eine echte Fund geht im Rauschen unter. Vergleichen Sie nur gesetzte Datensätze und prüfen Sie vor dem Melden erneut.

- **Durch Handbearbeitung von Zeilen reparieren.** Es korrigiert eine Tabelle, sendet kein Ereignis und lässt jeden nachgelagerten Consumer falsch zurück. Reparieren Sie über den Producer und die Outbox, damit die Änderung wie jede andere reist.
- **Eine Reparatur, die nicht idempotent ist.** Ein zweiter Lauf dupliziert die Korrektur. Veröffentlichen Sie mit der ursprünglichen `event_id` erneut, damit die Inbox einen zweiten Lauf und ein verspätetes Original gleichermaßen erkennt.
- **Den Abgleich verfallen lassen.** Der Job bricht nach einer Schemaänderung, niemand bemerkt es, und das Team glaubt sich abgesichert. Alarmieren Sie, wenn ein Lauf verspätet ist oder scheitert, und prüfen Sie den Rückstau an Abweichungen nach Zeitplan.

Zentrale Erkenntnisse

- Ein Ablauf, der sich über Services verteilt, braucht Identifikatoren, die jeden Hop überstehen: `correlation_id` in jeder Logzeile und jeder Zeile, und der in der Outbox-Zeile gespeicherte Trace-Kontext, damit der Relay den richtigen weiterleitet.
- Tracing hilft, erfasst aber nur Stichproben und kann verloren gehen, und Verknüpfungen gegenüber Eltern-Kind-Beziehung ist eine Abwägung je nach Fall, sodass die Identifikatoren im Umschlag für sich genügen müssen, um die Geschichte einer Bestellung zu rekonstruieren.
- Das Alter der Nachricht, gemessen an einer fachlichen Deadline und an der Aufbewahrung des Brokers, ist nützlicher als der Lag in Nachrichten, und es ist das Signal, das warnt, bevor ein zurückliegender Consumer still Ereignisse verliert; die Offset-Reset-Richtlinie entscheidet, ob ein solcher Verlust still oder laut ist.
- Alarmieren Sie auf Symptome, die an Deadlines gebunden sind, geben Sie jedem Alarm einen Verantwortlichen und eine genannte Begründung für seinen Schwellenwert, und nehmen Sie Alarme für Stille auf, denn die Ausfälle, die Daten verlieren, erzeugen oft gar kein Signal.
- Der Abgleich findet, was keine Metrik meldet: Er vergleicht gesetzte Datensätze über zwei Sichten auf die Wahrheit, ordnet Unterschiede als fehlend, veraltet oder überzählig ein und zeichnet sie auf, ohne etwas zu ändern.

- Reparaturen müssen den normalen Weg gehen und die ursprüngliche `event_id` erneut veröffentlichen, damit ein verspätetes Original und eine Reparatur nicht beide wirksam werden, mit einem Prüfpfad, und eine Abweichung wird vom nächsten erfolgreichen Vergleich geschlossen, nicht von der Reparatur selbst. Eine erneute Veröffentlichung erreicht jeden Subscriber, alte Ereignisse gehen also nur an den Consumer, der sie verpasst hat.
- Fachliche Invariantenprüfungen, etwa keine bestätigte Bestellung ohne eingezogene Zahlung, fangen Defekte ab, die zwei übereinstimmende Systeme verbergen würden.

Beobachtung und Abgleich sagen Ihnen, dass das System sich falsch verhält, und die Reparatur behebt eine Handvoll Datensätze. Kapitel 14 behandelt, was zu tun ist, wenn der Schaden weiter reicht, als eine Reparatur kommt: Wiederherstellung per Replay, eine Katastrophe überstehen und zu einem anderen Broker umziehen.

Kapitel 14 — Wiederherstellung, Katastrophen und Broker-Migration

Stellen Sie sich einen Dienstag bei Parcelly vor. Der Managed-Broker-Cluster, der jedes Ereignis auf dem Marktplatz transportiert, ist nicht mehr erreichbar. Die Statusseite des Anbieters meldet, dass die Zone, in der er läuft, ausgefallen ist und dass innerhalb einer Stunde ein Ersatz-Cluster zur Verfügung steht. Wenn es so weit ist, sind die Topics entweder vorhanden und leer, oder sie enthalten Daten bis zu einem Punkt irgendwo in den letzten Minuten. Wo genau dieser Punkt liegt, kann niemand sagen.

Den Services geht es gut. Orders, Payments, Inventory, Shipping und Ledger haben jeweils eine eigene Datenbank, und jede Datenbank ist intakt. Jeder Service hat einen Relay, der sein Veröffentlichen immer wieder versucht, und die Outboxes füllen sich. Die Consumer sind untätig und warten auf einen Broker, den es nicht gibt. In der Zwischenzeit muss das Team entscheiden, was zu tun ist, sobald er zurück ist, und alle Optionen klingen plausibel. Die Consumer auf den neuen Cluster zeigen lassen und laufen lassen. Alles aus den Outboxes erneut veröffentlichen. Das letzte Backup von irgendetwas wiederherstellen. Jede Option ist für einen Store richtig und für einen anderen falsch.

In diesem Kapitel geht es um den Tag danach, wenn ein Store fehlerhaft wird, verschwindet oder absichtlich ersetzt wird. Es stellt eine einzige Frage. Wenn der Broker, die Datenbank oder die Region nicht mehr das ist, was sie war: Welche der Garantien, die Sie gebaut haben, stehen noch, und wie nutzen Sie sie, um zu einem Zustand zurückzufinden, dem Sie vertrauen können?

Das Problem und warum es schwierig ist

Die früheren Kapitel haben Ausfälle abgewehrt, die die Stores intakt lassen: einen Absturz zwischen zwei Schritten, einen langsamen Consumer, ein Duplikat, eine Poison Message. Bei einer Wiederherstellung ist es anders, denn das, was schiefgegangen ist, ist das Gedächtnis selbst. Ein Store ist in der Zeit zurückgesprungen, hat einen Teil seiner Historie verloren oder wird gegen einen anderen ausgetauscht. Jede Abwehr in diesem Buch verlässt sich darauf, dass irgendein Store sich etwas merkt. Die Frage lautet also, welches Gedächtnis überlebt hat und welches nicht.

Das ist aus zwei Gründen schwierig. Erstens fallen die Stores unabhängig voneinander und in unterschiedliche Richtungen aus. Eine Datenbank, die auf den Stand von vor sechs Stunden zurückgesetzt wurde, und ein Broker, der seine letzten zwei Minuten verloren hat, widersprechen einander auf eine Weise, die keiner von beiden erkennen kann. Beide sind in sich konsistent. Kapitel 3 nannte das den Verlust eines ganzen Stores und stellte fest, dass zunächst nichts einen Fehler auslöst.

Zweitens kommt Wiederherstellungsarbeit selten vor, und selten vorkommende Arbeit wird unter Druck schlecht erledigt. Die Befehle, die eine Consumer-Group zurückversetzen, die Einstellungen, die verhindern, dass ein Replay eine zweite Runde E-Mails verschickt, die Reihenfolge, in der Services neu gestartet werden: Das alles sind Entscheidungen, die sich im Voraus günstig treffen und spontan teuer improvisieren lassen.

Die These dieses Kapitels ist die These des Buches in ihrer nacktesten Form. Der Broker ist nicht das, was überlebt. Was den Verlust eines Clusters, eine wiederhergestellte Datenbank oder den Wechsel eines Produkts überlebt, ist die Ausfallsemantik, die Sie um ihn herum entworfen haben: die Transactional Outbox (Kapitel 4), die sich merkt, was nie bestätigt wurde, die Inbox, die erkennt, was bereits erledigt ist, der stabile Bezeichner, der Vertrag und die Gewohnheit, zwei Sichten auf die Wahrheit miteinander zu vergleichen. Sind diese vorhanden, ist der Broker ersetzbar. Sind sie es nicht, hat der Broker die Korrektheit des Systems getragen, und niemand wusste es.

Wiederherstellung durch Replay

Der in Kapitel 2 definierte **Replay** ist das gezielte erneute Lesen der Historie aus einem Ereignis-Log. Es gilt nur für die Log-Familie, weil eine Warteschlange löscht, was sie zustellt. Kapitel 15 verwendet den Replay als Testtechnik. Hier ist er ein Werkzeug zur Wiederherstellung, und er hat ein Verfahren.

Die Situation, für die es passt, ist die vom Beginn von Kapitel 13: Eine Consumer-Group fiel zurück, die Aufbewahrung löschte Nachrichten, die sie nicht gelesen hatte, und die Gruppe zog an einer Lücke vorbei weiter. Sind die Daten noch aufbewahrt, ist die erste Wiederherstellung, die in Betracht kommt, nicht, Ereignisse einzeln vom Producer neu zu erzeugen. Es ist, die Position der Gruppe auf einen Punkt vor der Lücke zurückzusetzen und erneut zu lesen.

Eine Gruppe auf einen Zeitpunkt zurücksetzen

Die Position einer Consumer-Group wird als Offset pro Partition gespeichert. Sie zurückzusetzen ist eine administrative Aktion auf dem Broker, und die meisten logbasierten Broker können entweder einen expliziten Offset oder einen Zeitstempel ansteuern, den der Broker auf den ersten Offset zu oder nach diesem Zeitpunkt auflöst. Ich bevorzuge einen Zeitstempel. Ein Offset ist eine Zahl, die nur für eine Partition eines Topics auf einem Cluster etwas bedeutet, und der Zeitstempel ist das, was Sie wissen: Das Deployment begann um 09:40, also zurück auf 09:30. Prüfen Sie in der Dokumentation Ihres Brokers, wie er einen Zeitstempel auflöst, denn die Zeit des Producers und die Anfügezeit des Brokers sind nicht immer die, die er verwendet.

Das Verfahren ist kurz, und die Reihenfolge ist wichtig.

1. Die Gruppe anhalten. Die Offsets einer laufenden Gruppe zurückzusetzen schlägt je nach Broker entweder fehl oder wird vom nächsten Commit überschrieben.
2. Die aktuellen Offsets festhalten, bevor etwas geändert wird. Das ist der einzige Weg zurück, falls das Zurücksetzen falsch war.
3. Das Ziel mit einem Spielraum wählen. Gehen Sie weiter zurück, als Sie es für nötig halten, denn Duplikate sind billig und eine Lücke ist es nicht.

4. Auf das Ziel zurücksetzen, die Gruppe neu starten und das Alter der ältesten unverarbeiteten Nachricht (Kapitel 13) beobachten, während sie abgebaut wird.
5. Weiter beobachten, bis die Gruppe wieder am Head angekommen ist, und dann abgleichen, denn das Zurücksetzen repariert nur, was das Log noch enthielt.

Einen Spielraum zu wählen ist nur wegen der Inbox (Kapitel 7) sicher, die das erneute Lesen von Nachrichten, die die Gruppe verarbeitet hat, in Nulloperationen verwandelt. Eine Gruppe, deren Consumer nicht idempotent sind, sollte nicht zurückgesetzt werden, unabhängig davon, was das Verfahren vorsieht.

Replay auf eine Consumer-Group begrenzen

Ein Log lässt 10 Gruppen denselben Topic lesen, und jede hat ihre eigenen Offsets. Diese Eigenschaft macht das Zurücksetzen gezielt einsetzbar. Setzen Sie die Gruppe zurück, die die Ereignisse verpasst hat, und keine andere Gruppe bemerkt etwas. Setzen Sie die kleinste Menge an Gruppen zurück, die etwas verpasst haben, und nennen Sie sie im Plan, bevor Sie ihn ausführen. Der Consumer von `OrderPlaced` bei Payments behält seine Position, während die Gruppe von Inventory den letzten Tag noch einmal liest.

Eine zweite Form derselben Disziplin ist eine eigene Gruppe für den Replay. Muss eine Projektion neu aufgebaut werden, etwa die Suchsicht von Orders, starten Sie eine neue Gruppe mit anderem Namen, die vom Anfang in eine frische Tabelle liest, und schalten den Leser um, sobald sie aufgeholt hat. Die alte Gruppe bewegt sich nie, es gibt also nichts rückgängig zu machen, wenn der Neuaufbau schiefgeht. Das kostet für eine Weile Speicher und einen zweiten Consumer, was ein kleiner Preis für einen Rollback ist, der nichts kostet.

Seiteneffekte unterdrücken

Ein Replay wendet den Code von heute auf die Fakten von gestern an. Verschickt ein Handler E-Mails, ruft den Zahlungsanbieter auf oder benachrichtigt einen Paketdienst, tut er das beim Replay erneut. Kapitel 12 hat

das angesprochen, und die dort genannte Regel ist der Ausgangspunkt: Ein Consumer, der durch Replay neu aufgebaut wird, muss wissen, dass er einen Replay ausführt.

Die erste Idee, auf die man kommt, ist ein Flag in der Nachricht, ein Header, der sagt „dies ist ein Replay“. Das funktioniert nicht, weil eine wiedergegebene Nachricht die ursprüngliche Nachricht ist. Der Broker hält die Bytes, die der Producer geschrieben hat, und nichts markiert sie als zum zweiten Mal gelesen. Der Modus gehört zum Consumer, nicht zur Nachricht. Die Replay-Gruppe läuft als eigenes Deployment mit einem Konfigurationswert wie `mode: replay`, oder die Effektschicht prüft den Namen der Gruppe.

Ist der Modus bekannt, gibt es vier Optionen für die Effekte selbst, und jede hat ihren Preis.

- **Effekte am Rand mit einem Gate absichern.** Jeder externe Aufruf läuft über eine Funktion, die den Modus prüft und im Replay nichts tut. Das ist das klarste Design und am günstigsten zu prüfen, funktioniert aber nur, wenn kein Handler das Gate umgeht. Ein neuer Handler, der einen Anbieter direkt aufruft, hebelt es aus, die Regel braucht also einen Test (Kapitel 15).
- **Idempotenzschlüssel an externe Aufrufe hängen.** Trägt der Einzugsaufruf einen aus `event_id` abgeleiteten Schlüssel, erkennt der Anbieter den zweiten Versuch. Kapitel 7 hat das behandelt. Das schützt nur so lange, wie der Anbieter den Schlüssel behält. Es ist also eine zweite Verteidigungslinie und keine, auf die man sich bei einem Replay des letzten Monats verlassen sollte.
- **Die Outbox abgeleiteter Ereignisse anhalten.** Ein wiedergegebener Handler schreibt in derselben Transaktion neue Zeilen in seine eigene Outbox, und diese Zeilen gehen an den nächsten Service. Tragen die neuen Ereignisse frische Bezeichner, behandeln nachgelagerte Consumer sie als neue Fakten. Entweder leiten Sie den Bezeichner jedes ausgehenden Ereignisses deterministisch aus der eingehenden `event_id` und dem Namen des Handlers ab, sodass der zweite Lauf dieselben Bezeichner erzeugt und die nachgelagerten Inboxes sie auffangen, oder Sie halten den Relay des wiedergebenden Services an, bis Sie geprüft haben, was er erzeugt hat.

- **In eine andere Umgebung wiedergeben.** Für eine Projektion ohne Effekte ist eine neue Tabelle die sauberste Antwort, wie oben. Für einen Handler mit Effekten ist ein anderes Deployment, dessen Effektschicht an einen Stub angeschlossen ist, eine Möglichkeit, den Replay zu testen, bevor man ihn ernsthaft ausführt.

> **Warnung:** Eine Gruppe zurückzusetzen, die E-Mails verschickt, Karten belastet oder Paketdienste aufruft, ohne Gate und ohne Schlüssel, macht aus jedem historischen Ereignis eine neue Aktion. Die Inbox hilft nicht, wenn Sie sie ebenfalls zurücksetzen, und bei externen Effekten hätte sie nie helfen können, weil der Anbieter nicht mit Ihrer Datenbank zurückrollt. Legen Sie die Methode der Unterdrückung fest, bevor Sie die Offsets verschieben, nicht nach der ersten Beschwerde.

Das Deduplizierungsfenster und die Inbox

Die Inbox verändert, was ein Replay bewirkt, und das arbeitet für Sie oder gegen Sie, je nachdem, was Sie wollen.

Ist die Datenbank des Consumers intakt und liegt der Bereich, den Sie erneut lesen, innerhalb ihres Deduplizierungsfensters (Kapitel 7), erkennt die Inbox jede `event_id`, die sie bereits verarbeitet hat, und überspringt sie. Das wollen Sie nach einer Lücke: Die Gruppe verarbeitet nur, was fehlt, und der Rest ist billig. Deshalb kann ein Replay auch den Consumer reparieren, der ein Ereignis nie gesehen hat, ohne die zu stören, die es gesehen haben.

Ist der Bereich älter als das Fenster, sind die Inbox-Zeilen verschwunden. Der Consumer wendet alte Ereignisse ein zweites Mal an. Bei einem Consumer, der durch einen Constraint auf einen fachlichen Schlüssel geschützt ist, etwa einen Hauptbucheintrag pro Einzug, scheitert die zweite Anwendung folgenlos. Bei einem ohne diesen Schutz tritt der Effekt zweimal ein. Das Deduplizierungsfenster ist also Teil des Wiederherstellungsplans: Nennen Sie vor einem Replay das älteste Ereignis im Bereich, das kürzeste Fenster unter den Gruppen, die es sehen werden, und was den Consumer dort schützt, wo sich beide nicht überlappen. Kapitel 7 hat die Regel gegeben, dass das Fenster jeden Replay überdauern muss, das Sie planen. Dies hier ist der Plan.

Die Inbox kann auch das Hindernis sein. Ein Consumer mit einem Bug hat falschen Zustand geschrieben. Sie beheben den Bug und führen einen Replay aus, und jede Nachricht wird übersprungen, weil jede `event_id` in der Inbox steht. Dieser Fall kommt gleich.

Replay, Repair und Redrive

Drei Werkzeuge bewegen Nachrichten erneut, und sie unterscheiden sich darin, worauf sie wirken.

Werkzeug	Worauf es wirkt	Kapitel	Richtig, wenn
Repair	Eine Handvoll Datensätze, über die Outbox des Producers mit der ursprünglichen `event_id` erneut veröffentlicht	13	Wenige Einträge fehlen oder sind falsch, und der Producer sagen kann, welche
Redrive	Dead Letters, zurück an den Consumer verschoben, der sie abgelehnt hat	8	Nachrichten sind sichtbar fehlgeschlagen und die Ursache ist behoben
Replay	Ein Bereich des Logs, von einer Gruppe erneut gelesen	14	Viele Schlüssel sind betroffen, die Daten liegen noch in der Aufbewahrung und der Consumer ist idempotent

Die entscheidende Frage ist, wer weiß, was falsch ist. Beim Repair benennt der Producer oder ein Abgleichjob die konkreten Einträge. Beim Redrive hat das System die Nachrichten selbst beiseitegelegt. Beim Replay hat niemand eine Liste: Der Schaden ist ein Zeitbereich, und der günstigste Weg, einen Bereich abzudecken, ist, ihn komplett erneut zu lesen und die Inbox sortieren zu lassen, was erledigt wurde. Der Replay tauscht Genauigkeit gegen Abdeckung.

Aufbewahrung ist eine Wiederherstellungsgrenze

Ein Replay reicht nur so weit zurück, wie das Log Daten hält. Die Aufbewahrung, eine vor langer Zeit mit Blick auf Plattenkosten gewählte Einstellung, ist deshalb zugleich die Grenze der günstigsten Wiederherstellung, die Sie haben. Ein Ausfall, der länger dauert als die Aufbewahrung, oder ein Bug, der nach ihrem Ablauf entdeckt wird, bringt Sie vom günstigen Weg zum erneuten Veröffentlichen aus den Producern und zum Abgleichen.

Ich würde die Aufbewahrung an der Zeit ausrichten, die das Team braucht, um ein Problem zu bemerken und zu beheben, nicht allein an der Platte. Läuft ein Abgleichjob täglich und dauert es einen Tag, einen Fix auszuliefern, bleibt bei einer Aufbewahrung von einem Tag nichts übrig. Die Aufbewahrung wirkt auch auf die Kostenseite: Eine längere Aufbewahrung erhöht die Zahl alter Schemas, auf die ein Replay treffen kann (Kapitel 12), und verlängert, wie lange personenbezogene Daten im Log leben. Ich kenne die richtige Zahl für Ihr System nicht. Sie hängt davon ab, wie schnell Sie etwas bemerken, wie lange die Reparatur dauert und was das Log halten darf.

Ein Consumer, der falsch lag, ohne auszufallen

Kapitel 3 hat eine Zeile für einen Consumer-Bug ohne Fehler: Der Handler war erfolgreich und hat das falsche Ergebnis geschrieben. Das ist die Wiederherstellung, die ein Replay am dringendsten braucht, und die, bei der die Inbox im Weg steht.

Stellen Sie sich vor, der Consumer von `PaymentCaptured` bei Ledger hätte eine Woche lang Beträge für eine Währung mit dem falschen Exponenten umgerechnet. Jede Nachricht wurde bestätigt. Nichts ist fehlgeschlagen, und die Inbox enthält jede `event_id`. Der Abgleich (Kapitel 13) meldet schließlich, dass die Summen von Ledger für diese Währung nicht mit Payments übereinstimmen. Der Fix wird ausgeliefert, und nun existieren die falschen Zeilen und müssen korrigiert werden.

Den Topic in denselben Consumer wiederzugeben bewirkt nichts, weil die Inbox sagt, dass jedes Ereignis erledigt ist. Es gibt zwei ehrliche Wege, und die Wahl hängt davon ab, was Ledger ist.

Ist der Zustand des Consumers abgeleitet und kann weggeworfen werden, etwa bei einer Projektion, bauen Sie ihn neu auf. Starten Sie eine neue Gruppe in einen neuen Store mit leerer Inbox, geben Sie ab dem ersten betroffenen Ereignis wieder, verifizieren Sie das Ergebnis per Abgleich gegen die Quelle und schalten Sie um. Der falsche Zustand wird verworfen, nicht korrigiert.

Ist der Zustand des Consumers ein Datensatz, der nicht überschrieben werden darf, wie in einem Append-only-Hauptbuch, hilft der Replay trotzdem, aber nicht zum Überschreiben. Lesen Sie den betroffenen Bereich

mit dem korrigierten Code in einem Korrekturmodus. Berechnen Sie für jedes Ereignis, wie der Eintrag hätte aussehen müssen, vergleichen Sie ihn mit dem, was nach Abzug früherer Korrekturen gebucht ist, und buchen Sie für die Differenz einen Korrektureintrag, mit einem Bezeichner, der aus der ursprünglichen `event_id` und dem eigenen Namen der Korrektur abgeleitet ist, sodass ein zweimaliges Ausführen der Korrektur eine Nulloperation ist. Diese Arbeit leistet der Unique Constraint auf dem eigenen Bezeichner des Hauptbucheintrags, die Korrektur braucht also keine Inbox-Zeile, und die Schleife behandelt einen doppelten Schlüssel als „bereits korrigiert“ und geht zum nächsten Ereignis über.

Die falschen Einträge bleiben als Historie bestehen. Die Korrekturen sind neue Fakten, die sagen, warum. Das ist langsamer und expliziter als ein Neuaufbau, und es ist die einzige Form, die ein Prüfer akzeptiert.

```
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
```

Der gemeinsame Nenner ist, dass der Replay den betroffenen Bereich günstig findet und die Entscheidung darüber, was mit jedem Ereignis geschieht, der eigenen Semantik des Consumers gehört.

Katastrophenszenarien

Eine Katastrophe ist ein Ausfall, bei dem der Verlust keine Nachricht ist, sondern ein Store. Die brauchbare Art, die Möglichkeiten zu ordnen, ist danach, welcher Store in der Zeit zurückgesprungen ist. Schreiben Sie jedes Szenario in dem Muster auf, das Kapitel 3 verwendet hat: was kaputtging, woran man es erkennt und welche Garantie noch gilt. Das Letzte der drei ist das Nützliche, weil es benennt, worauf Sie sich verlassen, und ein Ablauf, der dort nichts eintragen kann, ist in diesem Szenario ungeschützt. Vor den Szenarien ein Wort dazu, was Sie anstreben.

Ziele für jeden Store

Zwei Zahlen beschreiben eine Wiederherstellung. Das **Recovery Point Objective (RPO)** gibt an, wie viele Daten Sie sich zu verlieren leisten können, gemessen als Zeit: Ein Backup alle sechs Stunden bedeutet ein RPO von bis zu sechs Stunden. Das **Recovery Time Objective (RTO)** gibt an, wie lange Sie ohne den Service auskommen können. Beides sind Entscheidungen, und der übliche Fehler ist, eine einzige für das ganze System zu treffen.

Sie unterscheiden sich, weil die Stores sich unterscheiden. Für die Stores in diesem Kapitel sieht eine Skizze so aus.

Store	Was ein Verlust kostet	Üblicher Weg zurück	Bestimmt
Eine Service-Datenbank	Zustand, der sonst nirgends existiert, es sei denn, ein Ereignis trägt ihn	Backup plus Log-Shipping	RPO pro Datenbank, festgelegt durch den geschäftlichen Wert der Daten
Die Outbox	Noch nicht veröffentlichte Ereignisse	Dieselbe Datenbank, also dasselbe RPO	Jede verlorene Zeile ist ein Ereignis, das nie existiert hat
Die Inbox	Das Gedächtnis darüber, was verarbeitet wurde	Dieselbe Datenbank wie die Effekte	Springt mit ihnen zurück (Kapitel 3)
Der Broker	unterwegs befindliche Ereignisse und aufbewahrte Historie	Replikation oder erneutes Veröffentlichen aus den Producern	Das RPO wird durch die Replikation bestimmt, das RTO dadurch, wie schnell Consumer abbauen

Store	Was ein Verlust kostet	Üblicher Weg zurück	Bestimmt
Eine abgeleitete Projektion	Nichts, wenn sie neu aufgebaut werden kann	Replay	Das RTO ist die Replay-Dauer, das RPO ist null, wenn das Log alles hält
Ein externer Anbieter	Effekte außerhalb Ihrer Kontrolle	Seine Aufzeichnungen und der Abgleich	Außerhalb beider Ihrer Ziele

Der Kern der Tabelle ist, dass ein RTO für die wiedergegebene Projektion eine Abflussrechnung ist. Hält das Log einen Monat und kann die Gruppe einen Tagesumfang pro Stunde verarbeiten, dauert der Neuaufbau etwa 30 Stunden, und das ist sein RTO (nehmen Sie diese Zahlen als Annahme; Ihre stammen aus Messungen, wie in Kapitel 13). Ein Team, das seine Abflussrate nie gemessen hat, kennt sein RTO nicht.

Ich würde die Ziele außerdem Ablauf für Ablauf an das Ausfallmodell aus Kapitel 3 binden. Der Ablauf für den Einzug von Zahlungen hat eine andere Toleranz als der Ablauf für Benachrichtigungen, und die Toleranz von Ledger gegenüber einer Lücke liegt näher bei null. Ein einheitliches Ziel ist am Ende für Benachrichtigungen zu streng und für das Hauptbuch zu lasch.

Eine Datenbank aus dem Backup wiederherstellen

Angenommen, die Datenbank von Orders wird auf einen Stand von sechs Stunden zuvor wiederhergestellt. Was bewirkt das bei der Maschinerie dieses Buches?

Die **Outbox** springt mit dem Zustand zurück. Zeilen, die in diesen sechs Stunden geschrieben wurden, sind weg, zusammen mit den Bestellungen, die sie angekündigt haben, obwohl Payments und Inventory auf die Ereignisse vielleicht schon reagiert haben. Zeilen, die zum Wiederherstellungszeitpunkt unveröffentlicht waren, aber in der alten Zeitlinie später veröffentlicht wurden, sind wieder unveröffentlicht, der Relay veröffentlicht sie also ein zweites Mal. Das ist ein harmloses Duplikat, weil die `event_id` dieselbe ist.

Es gibt eine schärfere Gefahr. Stammen Bestellbezeichner aus einer Sequenz, ist die wiederhergestellte Sequenz zurückgespult, und die nächste Bestellung erhält einen Bezeichner, der in der verlorenen Zeitlinie bereits

an eine andere Bestellung vergeben wurde. Payments hält nun einen Einzug für `order-81724`, der zu einem Käufer gehört, den Orders vergessen hat, und die neue `order-81724` gehört jemand anderem. Aus diesem Grund würde ich opake Zufallsbezeichner für Aggregate bevorzugen. Müssen Sie eine Sequenz behalten, setzen Sie sie vor der Annahme von Traffic über den höchsten je vergebenen Wert hinaus, eine Zahl, die Sie aus nachgelagerten Services auslesen können.

Die **Inbox** eines Consumers, dessen Datenbank wiederhergestellt wird, springt zusammen mit den Effekten zurück. Kapitel 3 hat darauf hingewiesen, dass das korrekt ist: Nachrichten, die vor dem Wiederherstellungszeitpunkt verarbeitet wurden, stehen noch in der Inbox, und ihre Effekte haben überlebt, und Nachrichten danach sind an beiden Stellen vergessen. Diese Eigenschaft gilt nur, wenn Inbox und Effekte in derselben Datenbank liegen.

Was nicht zurückspringt, ist der **Offset** des Consumers, der im Broker lebt. Die Gruppe steht am Head, jenseits von sechs Stunden an Ereignissen, die die wiederhergestellte Datenbank vergessen hat. Der Consumer sieht sie nie wieder, es sei denn, jemand versetzt die Gruppe zurück, daher enthält der Wiederherstellungsplan ein Zurücksetzen dieser Gruppe.

Das Ziel braucht Sorgfalt, denn der Wiederherstellungszeitpunkt ist eine Zeit auf der Uhr der Datenbank, während ein Zurücksetzen per Zeitstempel mit den Ereigniszeiten im Log arbeitet. Im Moment der Wiederherstellung stand die Gruppe nicht am Head. Sie stand an irgendeiner Position, und das Ereignis an dieser Position wurde einige Zeit zuvor erzeugt, und zwar um bis zu so viel wie das Alter der Nachricht der Gruppe (Kapitel 13). Das Ziel ist also die Ereigniszeit an der Position der Gruppe in diesem Moment, abzüglich eines Spielraums. Wird das Ziel allein aus dem Wiederherstellungszeitpunkt bestimmt, überspringt eine Gruppe, die eine Stunde im Rückstand war, die Stunde, die sie beim Zurückspringen der Datenbank noch nicht verarbeitet hatte.

Lesen Sie die Position aus der Historie des Alters der Nachricht oder aus den aufgezeichneten Offsets, und bewahren Sie diese Historie außerhalb der Datenbank auf, die Sie wiederherstellen wollen. Manche Teams speichern den Offset in derselben Datenbank wie die Effekte, in derselben

Transaktion, sodass beide gemeinsam zurückspulen. Das beseitigt den manuellen Schritt und übernimmt die Buchführung der Offsets vom Broker, was ein Kostenfaktor ist, den man abwägen muss.

Externe Effekte bleiben, wo sie waren. Der Einzug, den Payments in der verlorenen Zeitlinie an den Anbieter geschickt hat, existiert in den Aufzeichnungen des Anbieters und nicht in der Datenbank von Payments. Hier ist der Abgleich (Kapitel 13) erforderlich. Vergleichen Sie Ihre Aufzeichnungen mit denen des Anbieters und reparieren Sie die Unterschiede.

Die Reihenfolge, in der das System neu gestartet wird, ist wichtig, und es lohnt sich, sie als Verfahren aufzuschreiben.

```
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")
```

Die Schritte sind der Punkt. Beide Seiten zuerst anzuhalten verhindert, dass der wiederhergestellte Service seinen inkonsistenten Zustand verschlimmert, und das Abgleichen vor dem Fortsetzen folgt Kapitel 3. Das Zurücksetzen macht aus der Wiederherstellung ein Replay-Problem, mit einer Ergänzung.

Ein wiederhergestelltes Orders, das die verlorenen Stunden erneut liest, erzeugt neue Outbox-Zeilen, darunter ein Command `CapturePayment`, und eine frische `command_id` würde für die Inbox von Payments neu aussehen und könnte zu einem doppelten Einzug führen. Deshalb stammen die Bezeichner abgeleiteter Ereignisse aus der eingehenden `event_id`, wie im Abschnitt über Seiteneffekte. Ein Command, das von zwei Ereignissen ausgelöst wird, wie `CapturePayment`, nimmt seinen Bezeichner stattdessen aus dem fachlichen Schlüssel, etwa `order_id` plus Command-Typ (Kapitel 7), denn welches der beiden Ereignisse beim Replay zuletzt gelesen wird, kann sich ändern. Der Relay wird angehalten, bis jemand geprüft hat, was erzeugt wurde.

Broker-Daten verlieren

Ein Broker kann Daten verlieren, ohne den Cluster zu verlieren. In einer Log-Familie kann ein Replikat, das im Rückstand war, zum Leader befördert werden, und die Nachrichten, die es nicht hatte, sind weg. Die Ack-Stufe und die Mindestzahl synchroner Replikate entscheiden darüber, ob dem Producer gemeldet wurde, dass eine solche Nachricht sicher war (Kapitel 2). Manche Broker erlauben oder verbieten per Konfiguration, ein veraltetes Replikat zu befördern, und ich würde den Standardwert für Ihren Broker herausfinden und aufschreiben. Die Aufbewahrung kann dasselbe unbemerkt bewirken, wie Kapitel 13 gezeigt hat.

Was Sie sehen: Consumer-Positionen, die springen oder als außerhalb des gültigen Bereichs gemeldet werden, ein Topic, dessen End-Offset niedriger ist als zuvor, und im Nachhinein Befunde des Abgleichs der Art „fehlt“. Der Ack des Brokers ist ein Nachweis mit einer Grenze, und die Grenze ist die Zahl der Kopien, die in diesem Moment existierten.

Was überlebt: die Outbox, wenn Sie die Zeilen behalten haben. Ein Relay, der Zeilen unmittelbar nach dem Veröffentlichen löscht, hat kein Gedächtnis dafür, was er gesendet hat. Einer, der veröffentlichte Zeilen für eine Zeit behält, sagen wir als Veranschaulichung einige Tage, kann die Zeilen aus dem betroffenen Zeitfenster als unveröffentlicht markieren und erneut senden. Die erneute Veröffentlichung behält die ursprüngliche `event_id`, sodass Consumer, die ein Ereignis erhalten haben, die zweite Kopie als Duplikat behandeln. Das ist derselbe Schritt wie das Repair aus Kapitel 13, auf einen Bereich angewandt, und er trägt dieselbe Fan-out-Warnung: Er-

eignisse, die älter sind als das Deduplizierungsfenster eines Abonnenten, müssen an den einen Consumer zugestellt werden, der sie verpasst hat, nicht an alle.

Sind die Outbox-Zeilen weg, sind die eigenen Aufzeichnungen des Producers die nächste Quelle: den Umschlag aus dem fachlichen Datensatz mit seiner gespeicherten `event_id` und dem Payload-Hash neu aufbauen und für einen Menschen anhalten, wo der Hash abweicht. Und existiert keines von beidem, lässt sich das Ereignis nicht wiederherstellen, und der Abgleich wird es weiter melden. An diesem Punkt ist die richtige Frage eine geschäftliche: Wird der fehlende Fakt aus einem anderen System rekonstruiert oder als verloren akzeptiert und erklärt? So zu entwerfen, dass diese Frage selten gestellt wird, ist das, was man sich damit erkauft, die Outbox-Zeilen eine Weile zu behalten.

Eine Region verlieren

Der Verlust einer Region verbindet die beiden vorigen Fälle und bringt ein eigenes Problem hinzu. Nehmen Sie an, Parcelly betreibt in einer zweiten Region einen Standby-Cluster, der von einem Replikationswerkzeug gespeist wird, das Topics fortlaufend kopiert, und dass Datenbanken asynchron replizieren.

Die erste Eigenschaft, die festzuhalten ist: **Asynchrone Replikation hat ein RPO**, und es ist die Replikationsverzögerung im Moment des Ausfalls. Was in der Primärregion geschrieben und noch nicht kopiert wurde, ist aus Sicht des Standby verloren. Das ist das Maß dessen, was Sie verlieren, und es ist eine Zahl, die von Minute zu Minute schwankt. Es lohnt sich auch, bei Überschreiten eines festgelegten Limits zu alarmieren, so wie Kapitel 13 das Alter der Nachricht behandelt: Sie beobachten Ihr RPO.

Die zweite Eigenschaft: **Offsets reisen nicht mit.** Der Standby hält eine Kopie der Nachrichten, aber die Kopie hat eigene Offsets, die vergeben wurden, als das Replikat sie schrieb. Die festgeschriebene Position eines Consumers in der Primärregion sagt nichts über den Standby. Manche Replikationswerkzeuge bieten Offset-Übersetzung an; ich würde prüfen, was Ihres garantiert, und ich würde keine Wiederherstellung entwerfen, die davon abhängt, dass sie exakt ist.

Der portable Plan ist, jede Gruppe auf dem Standby ab der Ereigniszeit der Position zu starten, die sie beim Ausfall der Primärregion erreicht hatte, abzüglich eines Spielraums, und die Inbox die Überlappung auffangen zu lassen. Es ist die Regel aus der Datenbankwiederherstellung. Eine Gruppe, die 40 Minuten im Rückstand war, startet 40 Minuten früher. Ereignisse, die nie repliziert wurden, sind auf dem Standby gar nicht vorhanden, und das unten beschriebene Outbox-Fenster deckt sie ab.

Weil die eigenen Commit-Informationen der Primärregion mit der Region verloren gehen, muss die Historie des Alters der Nachricht außerhalb von ihr aufbewahrt werden. Ist sie weg, weichen Sie auf den ältesten Punkt innerhalb der Deduplizierungsfenster der Consumer aus und nehmen die zusätzlichen Duplikate in Kauf. Ein Alarmschwellenwert reicht nicht, denn der Alarm könnte zu dieser Zeit bereits ausgelöst haben.

Die dritte Eigenschaft: **Duplikate beim Failover sind sicher, nicht bloß möglich.** Ein Consumer, der früher als seine wahre Position neu startet, verarbeitet Ereignisse zweimal. Ein Relay, der beim Ausfall der Region mitten im Veröffentlichen war, veröffentlicht möglicherweise ein zweites Mal aus der überlebenden Datenbank.

Lücken sind die andere Seite: Ereignisse, die die Primärregion bestätigt hat und der Standby nie erhielt. Hat die Datenbank des Producers überlebt und ist neuer als der Broker des Standby, enthält sie Zeilen, die als veröffentlicht markiert sind und die der Standby nie gesehen hat. Erweitern Sie dafür das Fenster für das erneute Veröffentlichen der Outbox wie im Fall des Broker-Verlusts: Markieren Sie die Zeilen aus dem letzten Intervall der Replikationsverzögerung plus einem Spielraum als unveröffentlicht und senden Sie sie erneut. Die Replikationsverzögerung bestimmt die Größe dieses Fensters, und nur dieses Fensters. Duplikate kosten beim Consumer nichts. Lücken kosten das, was Sie zu schützen versuchen.

Ist die Datenbank des Producers selbst mit der Region verloren und wird aus einem älteren Replikat wiederhergestellt, reduziert sich der Fall auf die Wiederherstellung aus dem Backup, mit den zurückgespulten Sequenzen und den vergessenen Bestellungen, die dazugehören.



Lesen Sie das Diagramm von links. Die beiden Schritte, die mit dem Unbekannten umgehen, das Fenster für das erneute Veröffentlichen und der Spielraum beim Zeitstempel, irren beide in Richtung Duplikation. Jeder folgende Schritt dient dazu, Duplikation harmlos zu machen.

Broker-Migration

Eine Migration ist eine Katastrophe, die Sie terminieren. Ein Team wechselt wegen der Kosten, eines Support-Endes, eines Managed-Angebots oder eines geänderten Zugriffsmusters von einem Broker zu einem anderen, und der Wechsel hat dieselben Zutaten wie die Katastrophen oben: Daten auf der einen Seite, einen neuen leeren Ort auf der anderen, Positionen, die sich nicht übertragen lassen, und eine Zeit, in der zwei Systeme gleichzeitig leben. Der Unterschied ist, dass Sie Zeit haben, sorgfältig zu sein, und kein Notfall Sie zum Improvisieren zwingt.

Die Werkzeuge des Buches machen eine Migration zu einer kontrollierten Änderung. Der Grund ist, dass mehrere von ihnen absichtlich zwischen Anwendung und Broker gesetzt wurden.

Der Relay als einzige Umschaltstelle

Mit einer Transactional Outbox (Kapitel 4) spricht ein produzierender Service nie mit dem Broker. Er schreibt in seine eigene Datenbank, und der Relay (Kapitel 5) bringt die Zeilen zum Broker. Die Wahl des Brokers steckt also in einem einzigen Prozess. Zu ändern, wohin Ereignisse gehen, heißt zu ändern, wohin der Relay veröffentlicht, und der Code des Producers bleibt unverändert. Das ist eine kleinere Änderung als eine, die sich über jeden Service verteilt, ein Nebennutzen eines Patterns, das Kapitel 4 aus einem anderen Grund empfiehlt.

Der Relay besitzt außerdem den Umschlag. Er übernimmt `event_id` aus der Outbox-Zeile und setzt sie in die Nachricht, die er veröffentlicht. Auf dem neuen Broker hat dasselbe Ereignis dieselbe `event_id`, und diese Gleichheit macht den Rest der Migration sicher.

Dual-Publish

Der sicherste erste Schritt ist, dass der Relay jede Zeile an beide Broker veröffentlicht. Eine Weile existiert jedes Ereignis auf dem alten Broker, wo die heutigen Consumer es lesen, und auf dem neuen, wo es noch niemand liest.

Das Dual-Publish ist selbst ein Dual Write, und Kapitel 4 erklärt, warum diese gefährlich sind. Hier ist das Risiko kleiner, weil die Outbox-Zeile die Quelle der Wahrheit ist und ein Absturz die Zeile zum erneuten Senden zurücklässt.

Die Entwurfsfrage ist, worauf der Relay wartet. Markiert er eine Zeile erst als veröffentlicht, nachdem beide Veröffentlichungen gelungen sind, lässt ein nicht erreichbarer Shadow-Broker Zeilen unveröffentlicht und stoppt den Live-Ablauf, von dem die Consumer abhängen, um eines Brokers willen, den niemand liest. Verfolgen Sie die beiden Ziele deshalb getrennt. Die Zeile gilt im Sinne von Kapitel 5 als veröffentlicht, wenn der alte Broker sie hat, und eine zweite Markierung hält den neuen Broker fest. Ein Durchlauf in der Reihenfolge von `id` füttert den neuen Broker mit den Zeilen, denen die zweite Markierung fehlt, was die Reihenfolge pro Schlüssel erhält, und das Löschen wartet auf beide. Der Preis ist eine Spalte mehr und ein Durchlauf mehr, und im Gegenzug zeigt sich ein Ausfall des neuen Brokers als wachsende Lücke im Vergleich und nicht als Ausfall von Parcelly.

In dieser Phase bleibt der alte Broker der, dem die Consumer vertrauen. Der neue Broker ist ein Shadow, und seine einzige Aufgabe ist, mit dem Original verglichen zu werden.

Spiegelung und Bridges

Eine Alternative zum Dual-Publish im Relay ist, Nachrichten mit einem Werkzeug zwischen den Brokern zu kopieren: einem Spiegelungswerkzeug für Log zu Log oder einer Bridge oder einem Connector für Log zu Warteschlange und Warteschlange zu Log. Das erfordert keine Änderung am Relay und überträgt auch die Historie, die vor Beginn der Migration existierte, was das Dual-Publish nicht tut.

Zwei Warnungen. Eine Spiegelung oder Bridge ist ein Consumer und ein Producer, ist also At-least-once und erzeugt Duplikate, wie es jedes Consumer-Producer-Paar tut. Und sie muss `event_id` unverändert durchreichen, im Nachrichtenkörper oder in einem Header. Eine Bridge, die einen eigenen Bezeichner prägt, macht jedes kopierte Ereignis zu einem neuen Ereignis, und die Inbox kann keines davon erkennen. Testen Sie das an einer Stichprobe, bevor Sie sich darauf verlassen. Ob eine Spiegelung Partitionierung, Schlüssel und Header erhält und in welcher Reihenfolge über Partitionen hinweg, unterscheidet sich je nach Werkzeug, und die Antwort steht in dessen Dokumentation, nicht in diesem Buch.

Offsets lassen sich nicht übersetzen

Der Offset einer Consumer-Group auf dem alten Broker ist auf dem neuen bedeutungslos. Selbst zwischen zwei Brokern derselben Art hat die Kopie ihre eigene Nummerierung. Zwischen einer Warteschlange und einem Log gibt es auf einer Seite überhaupt keinen Offset.

Die Antwort ist die, zu der der Abschnitt über den Verlust einer Region gelangt ist: einen sicheren Startpunkt wählen und sich auf Idempotenz verlassen. Wählen Sie für jede Gruppe einen Zeitstempel, nach dem jedes Ereignis veröffentlicht wurde, dessen Verarbeitung die Gruppe auf dem alten Broker noch nicht abgeschlossen hatte. Nehmen Sie den Lag der Gruppe im Moment der Umschaltung, rechnen Sie ihn in Zeit um und addieren Sie einen Spielraum. Starten Sie die neue Gruppe auf dem neuen Broker bei diesem Zeitstempel. Sie liest Ereignisse erneut, die die alte Gruppe schon behandelt hat, und die Inbox überspringt sie.

Das Deduplizierungsfenster gilt hier wie bei einem Replay: Das älteste Ereignis, das die neue Gruppe erneut lesen wird, muss innerhalb des Fensters jedes Consumers liegen.

Reihenfolge und Partitionen

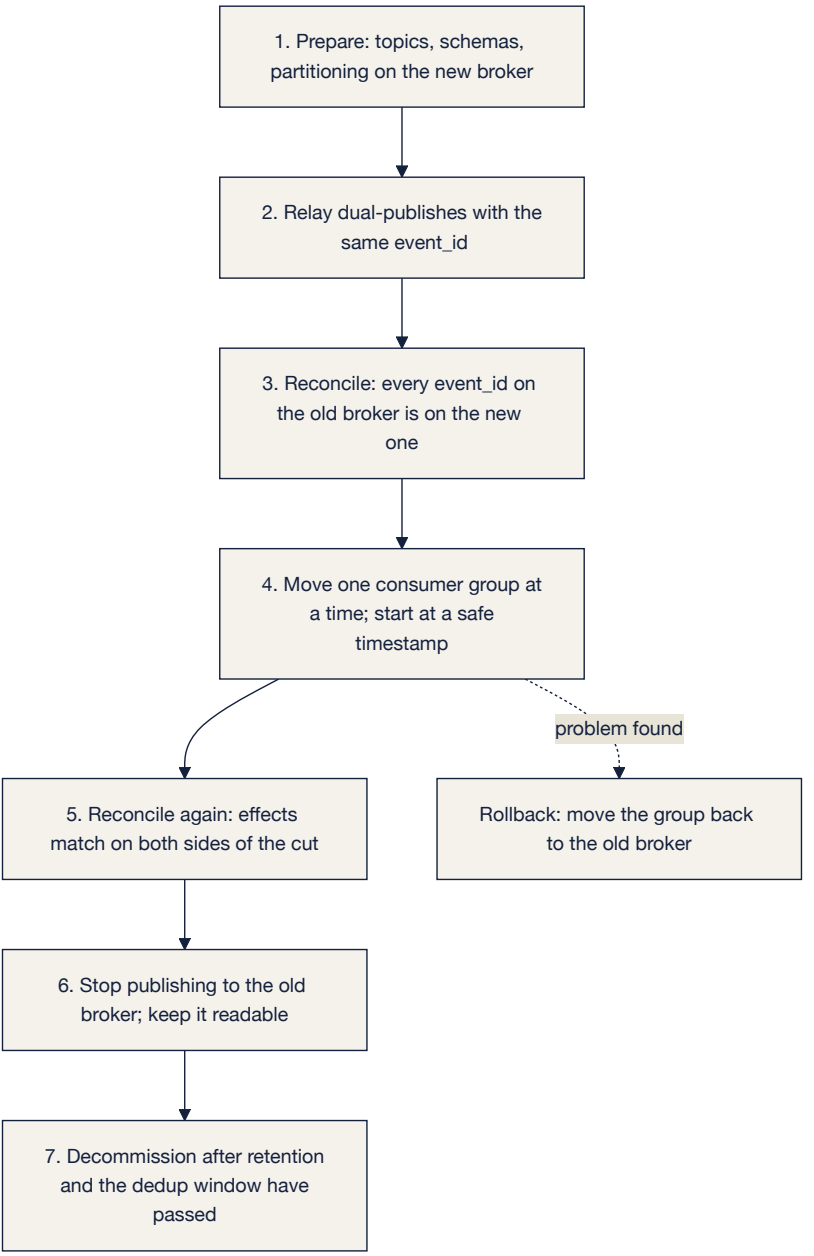
Kapitel 9 hat die Bedingung für die Reihenfolge pro Schlüssel genannt: derselbe Schlüssel, derselbe Partitionierer, dieselbe Partitionszahl. Eine Migration ändert mindestens eines der drei. Der neue Broker verwendet vielleicht einen anderen Partitionierer, sodass eine `order_id` auf einer Partiti-

on landet, auf der sie nie war, und der neue Topic wird womöglich mit einer anderen Zahl von Partitionen angelegt, weil man die Zahl in diesem Moment überdenkt.

Daraus folgen zwei Dinge. Berechnen Sie die Partition im Relay aus dem gespeicherten `partition_key`, wie Kapitel 9 es vorgeschlagen hat, sodass ein Stück Code die Zuordnung besitzt, und testen Sie, dass Ereignisse für einen Schlüssel im neuen Broker auf einer Partition landen. Und planen Sie die Umschaltung so, dass kein Schlüssel Ereignisse in Bearbeitung an beiden Orten hat. Beginnt ein Consumer, den neuen Topic zu lesen, während der alte Topic noch unverarbeitete Ereignisse für dieselbe `order_id` hält, können die neuen Ereignisse vor den alten behandelt werden: wieder `OrderCancelled` vor `OrderConfirmed`, der Fehler aus Kapitel 9. Es gibt zwei Wege, das zu vermeiden. Das alte Backlog auf null abbauen, bevor die neue Gruppe gestartet wird, was die Verarbeitung für die Dauer des Abbaus pausiert. Oder die Consumer von vornherein tolerant machen, mit zustandstragenden Ereignissen und einer Aggregatversion, die alles Ältere zurückweist, wie Kapitel 9 es beschrieben hat. Ich würde für jeden Ablauf, bei dem die Reihenfolge zählt und die Pause tragbar ist, zuerst abbauen.

Die Umschaltung

Die Reihenfolge der Schritte ist das, wonach die Leute fragen: zuerst die Consumer oder zuerst die Producer? Mit Dual-Publish löst sich die Frage größtenteils auf, weil die Producer bereits an beide schreiben, bevor ein einziger Consumer umzieht. Die Reihenfolge wird zu dieser.



Der Abgleich klammert den Wechsel ein. Davor vergleichen Sie die Menge der Werte von `event_id` auf den beiden Brokern über ein Zeitfenster: Alles, was auf dem alten Broker ist und auf dem neuen fehlt, ist ein Defekt des Dual-Publish, gefunden, solange niemand vom neuen Broker abhängt. Danach vergleichen Sie die Effekte: Eine um 14:00 umgezogene Gruppe sollte denselben Zustand erzeugt haben, den sie auf dem alten Broker er-

zeugt hätte, und der Abgleich über Services hinweg aus Kapitel 13 verifiziert das. Eine Migration, die für beendet erklärt wird, weil die Consumer gesund sind, hat nur verifiziert, dass nichts abgestürzt ist.

Ziehen Sie Consumer-Groups einzeln um, beginnend mit der am wenigsten kritischen, sagen wir Notifications, und endend mit Ledger. Jede Gruppe ist ein kleiner, einzeln umkehrbarer Schritt.

Rollback

Eine Migration ohne Rollback-Plan ist eine Tür in nur eine Richtung. Mit diesem Design ist der Plan fast kostenlos, solange eine Bedingung gilt: Der alte Broker erhält weiter jedes Ereignis. Solange der Relay doppelt veröffentlicht, ist der alte Broker vollständig, ein Rollback bedeutet also, die Gruppe mit einem Spielraum auf einen Zeitstempel zurück auf den alten Broker zu richten. Die Inbox fängt die Ereignisse auf, die die Gruppe am neuen Ort behandelt hat.

Diese Bedingung ist der Grund, warum Schritt sechs spät kommt. Sobald der Relay nicht mehr an den alten Broker schreibt, erfordert ein Rollback, dass der Fluss umgekehrt wird: Der Relay schreibt wieder an den alten, und die Consumer des alten Brokers starten ab dem Punkt, an dem er aufgehört hat. Das ist weiterhin möglich, weil die Outbox die Zeilen hält, und es ist langsamer. Ein Rollback ist außerdem durch die Aufbewahrung auf dem alten Broker und durch das Deduplizierungsfenster begrenzt, deshalb würde ich ihn erst außer Betrieb nehmen, wenn beide seit dem letzten Ereignis, das er erhalten hat, abgelaufen sind. Den alten Broker am Tag des Umzugs des letzten Consumers stillzulegen entfernt den Rollback genau in dem Moment, in dem die Probleme des neuen Brokers am wahrscheinlichsten auftreten.

Zwischen Warteschlange und Log

Wenn die Migration die Familien wechselt, ändert sich die Semantik, und ein Plan, der sie als Adressänderung behandelt, ist falsch.

Aspekt	Warteschlange zu Log	Log zu Warteschlange
Historie	Gewonnen: Replay wird möglich (und die Aufbewahrung wird zur Entscheidung)	Verloren: kein Replay, die Wiederherstellung stützt sich also auf die Outbox und die Producer

Aspekt	Warteschlange zu Log	Log zu Warteschlange
Bestätigung	Der Ack pro Nachricht wird zu einem Offset-Commit; eine langsame Nachricht kann nun ihre Partition blockieren (Kapitel 9)	Ein Offset-Commit wird zum Ack pro Nachricht, mit Visibility-Timeouts und erneuter Zustellung
Reihenfolge	Die Gruppenreihenfolge, wo die Warteschlange sie bietet, wird zur Reihenfolge pro Partition; wählen Sie den Schlüssel	Die Reihenfolge pro Partition muss neu geschaffen werden, zum Beispiel mit Message-Groups, wo die Warteschlange sie unterstützt
Dead Letters	Der Redrive ist etwas, das Sie bauen oder konfigurieren; das Log hat keine eingebaute Dead-Letter-Queue (DLQ), nur ein Dead-Letter-Topic, den Sie anlegen (Kapitel 8)	Die DLQ-Funktionen der Warteschlange kommen hinzu, und ihre Grenzen und die Aufbewahrung unterscheiden sich
Verzögerung und Ablauf	Verzögerte Zustellung und Ablauf pro Nachricht gibt es womöglich nicht, und sie brauchen einen Ersatz	Verzögerung und Ablauf sind womöglich nativ; die Wiederherstellung über die Aufbewahrung entfällt

Jede Zeile ist eine Stelle, an der sich ein Verhalten, auf das sich Ihre Consumer verlassen haben, unbemerkt ändern kann. Ich würde vor der Wahl des Ziels die Liste der Verhaltensweisen aufschreiben, von denen jeder Consumer abhängt, und die wichtigen davon in einer Übung testen (Kapitel 15), etwa eine Poison Message, die ihre Partition nicht blockieren darf.

Wiederherstellung üben

Nichts davon ist viel wert, wenn Sie es zum ersten Mal unter Druck ausführen. Eine Wiederherstellung, die nie durchgeführt wurde, ist ein Glaube. Ein Replay-Verfahren, das nie gegen realistisches Volumen gelaufen ist, hat keine Dauer.

Zwei Übungen zahlen sich aus. Ein **Wiederherstellungstest** nimmt ein Backup, stellt es in einer Scratch-Umgebung wieder her, prüft, was die Wiederherstellung mit der Outbox, der Inbox und den Offsets macht, und führt dann das Verfahren zum Fortsetzen aus und misst seine Dauer. Ein **Game Day** entfernt absichtlich an einem sicheren Ort etwas, einen Broker, eine Zone, eine Consumer-Datenbank, und lässt das Team sich allein an-

hand des Runbooks erholen. Kapitel 15 behandelt, wie man diese Übungen entwirft. Der Grund gehört hierher: Nur das Durchführen einer Wiederherstellung zeigt, welche Schritte Sie vergessen haben.

Ein Runbook für die Szenarien dieses Kapitels muss eine kurze Liste von Fragen beantworten, aufgeschrieben und nicht im Kopf von irgendjemandem getragen. Kapitel 16 nimmt sie in seine Hinweise zum Runbook auf.

- Welcher Store ist betroffen, und welcher ist in der Zeit zurückgesprungen?
- Was ist der Wiederherstellungszeitpunkt, und wie wurde er bestimmt?
- Welche Consumer-Groups müssen zurückgesetzt werden, auf welchen Zeitstempel und mit welchem Spielraum?
- Welche Effekte müssen während des Replays unterdrückt werden, und mit welchem Mechanismus?
- Liegt jeder erneut zu lesende Bereich innerhalb des Deduplizierungsfensters jedes betroffenen Consumers?
- Was wird in welcher Reihenfolge angehalten, und wer setzt es fort?
- Welche Abgleiche bestätigen, dass die Wiederherstellung funktioniert hat, und wer liest sie?
- Wie wird das Ganze rückgängig gemacht, wenn es schiefgeht?

Abwägungen und Kosten

Die Aufbewahrung ist ein Wiederherstellungsbudget. Eine längere Aufbewahrung hält die günstige Wiederherstellung länger verfügbar, zu den weiter oben aufgeführten Kosten. Eine kürzere Aufbewahrung drängt Sie dazu, aus den Producern erneut zu veröffentlichen.

Outbox-Zeilen zu behalten kostet Speicher und einen Bereinigungsjob. Dieselben Zeilen, die ein Fenster für das erneute Veröffentlichen möglich machen, sind die, die ein schlankeres Design sofort löscht. Mit Change Data Capture (CDC) und sofortigem Löschen gibt es das Fenster nicht (Kapitel 5). Wenn Sie die Sicherheit wollen, behalten Sie veröffentlichte Zeilen für einen festgelegten Zeitraum und überwachen Sie die Tabelle.

Der Replay ist ein stumpfes Instrument. Er liest alles im Bereich erneut, auch das, was in Ordnung war, und er verlässt sich darauf, dass jeder Consumer idempotent und jeder Effekt mit einem Gate abgesichert ist. Für ein paar bekannte Datensätze ist das Repair günstiger und sicherer. Der Replay verdient seinen Platz, wenn der Schaden breit ist und niemand eine Liste hat.

Die Unterdrückung von Seiteneffekten ist Code, der getestet werden muss. Ein Gate, das existiert, aber von einem neuen Handler umgangen wird, ist schlimmer als keines, weil es Vertrauen erzeugt. Der Preis ist ein dauerhafter Test und eine Regel im Code-Review.

Eine Migration mit Dual-Publish kostet doppelt. Zwei Broker laufen gleichzeitig, der Relay leistet mehr Arbeit und verfolgt zwei Ziele, und die Duplikate auf der alten Seite nehmen zu. Im Gegenzug erhalten Sie eine Verifikation, bevor jemand vom neuen Broker abhängt, und einen Rollback. Toleriert das System ein Wartungsfenster, ist eine direkte Umschaltung günstiger: den Relay anhalten, die alten Consumer abbauen, den Relay auf den neuen Broker richten und dort die Consumer starten. Mit einer Outbox braucht das einen Ausfall und keinen Verlust, weil die Zeilen warten. Es verzichtet auf die Verifikation und den Rollback, und ich würde es für das Hauptbuch nicht wählen.

Wann man nicht alles davon bauen sollte. Ein Ablauf, bei dem eine Lücke billig ist, etwa ein interner Metrik-Feed, braucht kein Fenster für das erneute Veröffentlichen, kein Runbook und keinen Game Day. Passen Sie die Wiederherstellung an die Kosten des Ausfalls an und schreiben Sie auf, welche Abläufe Sie ungedeckt gelassen haben. Ihre Abwesenheit im Plan sollte eine Entscheidung sein.

Häufige Fehler

- **Eine Gruppe zurücksetzen, ohne ihre Seiteneffekte zu prüfen.** Es ist verlockend, weil das Zurücksetzen ein einzelner Befehl ist und die Daten da sind. Das Ergebnis ist eine zweite Runde E-Mails und ein zweiter Belastungsversuch. Entscheiden Sie, wie Effekte unterdrückt werden, bevor Sie einen Offset verschieben.
- **Zu viele Gruppen zurücksetzen.** Alle neu zu starten fühlt sich gründlich an. Jede Gruppe, die Sie zurücksetzen, liest Historie erneut und

trägt das Risiko ihrer eigenen Effekte. Setzen Sie nur die Gruppen zurück, die etwas verpasst haben.

- **Sich auf einen Replay-Header verlassen.** Es ist verlockend, weil ein Flag an der Nachricht wie das günstigste Signal aussieht. Eine wiedergegebene Nachricht ist die ursprüngliche Nachricht und trägt kein Zeichen, dass sie erneut gelesen wird. Der Modus gehört zum Deployment des Consumers oder zur Effektschicht.
- **Ein Replay in einen Consumer mit der Inbox des Bugs.** Nach einem Consumer-Bug überspringt die Inbox jede Nachricht als erledigt. Bauen Sie abgeleiteten Zustand in einem frischen Store neu auf oder buchen Sie bei einem Append-only-Store explizite Korrekturen.
- **Ein Wiederherstellungsziel für das ganze System.** Es ist verlockend, weil es eine einzige Zahl ist, auf die man sich einigen muss. Es ist für Benachrichtigungen zu streng und für Ledger zu lasch. Legen Sie Ziele pro Ablauf und pro Store fest.
- **Eine Datenbank wiederherstellen und sofort fortsetzen.** Der Service ist gesund und die Dashboards sind grün, also wird der Traffic fortgesetzt. Der wiederhergestellte Service veröffentlicht dann zurückgespulten Zustand, vergibt Bezeichner erneut und verpasst Ereignisse, die seine Offsets übersprungen haben. Anhalten, abgleichen, Offsets zurücksetzen, dann fortsetzen.
- **Sequenzbasierte Bezeichner nach einer Wiederherstellung weiterverwenden.** Sequenzen sind der Standard, und nichts sieht falsch aus, bis zwei Bestellungen einen Bezeichner teilen. Eine zurückgespulte Sequenz vergibt einen Bezeichner, der bereits einer anderen Bestellung gehört. Verwenden Sie opake Bezeichner oder setzen Sie die Sequenz über den höchsten je vergebenen Wert hinaus.
- **Der Offset-Übersetzung vertrauen.** Es ist verlockend, weil das Werkzeug damit wirbt und sie die Rechnerei mit Zeitstempeln ersparen würde. Ein Replikationswerkzeug, das Nachrichten kopiert, bildet Offsets womöglich exakt ab, womöglich nicht. Planen Sie eine Wiederherstellung, die ohne sie funktioniert, mit Zeitstempeln und der Inbox.
- **Eine Bridge, die neue Bezeichner prägt.** Eine Bridge ist oft so konfiguriert, dass sie jede Nachricht in einen eigenen Umschlag packt, und das besteht einen Test mit einem Consumer. Die kopierten Ereignisse sehen neu aus, die Inbox erkennt also nichts, und jedes Duplikat wird

zum Effekt. Reichen Sie `event_id` unverändert durch und testen Sie das.

- **Den alten Broker mit dem Umzug des letzten Consumers außer Betrieb nehmen.** Es fühlt sich an, als wäre die Migration fertig. Es entfernt den Rollback genau dann, wenn er am wahrscheinlichsten gebraucht wird. Warten Sie, bis die Aufbewahrung und das Deduplizierungsfenster abgelaufen sind.
- **Nie ein Backup wiederherstellen.** Der Backup-Job ist grün. Die erste Wiederherstellung ist dann die echte. Stellen Sie regelmäßig in einer Scratch-Umgebung wieder her und messen Sie die Dauer.

Zentrale Erkenntnisse

- Wenn ein Store verloren geht, wiederhergestellt oder ersetzt wird, überlebt die Ausfallsemantik um ihn herum, also die Outbox, die Inbox, die stabile `event_id`, der Vertrag und der Abgleich, und der Broker selbst ist ersetzbar.
- Der Replay ist ein Wiederherstellungswerkzeug für die Log-Familie: eine Consumer-Group anhalten, ihre Offsets festhalten, sie mit einem Spielraum auf einen Zeitstempel zurücksetzen, der von der tatsächlichen Position der Gruppe aus gemessen wird, und die Inbox die Duplikate auffangen lassen, solange die Daten noch innerhalb der Aufbewahrung liegen.
- Eine wiedergegebene Nachricht sieht aus wie das Original, also muss der wiedergebende Consumer seinen Modus kennen und seine Effekte mit einem Gate absichern, nachgelagerte Bezeichner deterministisch ableiten oder seine Outbox anhalten und sich nie auf einen Header verlassen.
- Das Deduplizierungsfenster ist Teil jedes Wiederherstellungsplans, weil ein Bereich, der älter ist als das Fenster, Duplikate wieder zulässt, und ein Consumer-Bug die Inbox von einem Schutz in ein Hindernis verwandelt, das einen Neuaufbau oder explizite Korrekturen erfordert.
- Recovery Point Objective und Recovery Time Objective gehören zu jedem Store und jedem Ablauf, und eine wiederhergestellte Datenbank spult ihre Outbox, Inbox und Sequenzen zurück, während die Offsets bleiben, wo der Broker sie gelassen hat, daher hält das Verfahren zum Fortsetzen an, gleicht ab, setzt Offsets zurück und setzt erst dann fort.

- Asynchrone Replikation über Regionen hat ein RPO in Höhe ihrer Verzögerung und überträgt Daten, aber keine Consumer-Offsets, daher startet ein Failover jede Gruppe ab der Ereigniszeit ihrer eigenen Position beim Ausfall, bemisst ein Fenster für das erneute Veröffentlichen der Outbox nach der Replikationsverzögerung und behandelt Duplikate als sicher und Lücken als das eigentliche Risiko.
- Eine Broker-Migration nutzt den Relay als einzige Umschaltstelle, veröffentlicht per Dual-Publish mit derselben `event_id`, zieht jeweils eine Consumer-Group ab einem sicheren Zeitstempel um, gleicht davor und danach ab und hält den alten Broker lesbar, bis ein Rollback nicht mehr gebraucht wird.

Ein Wiederherstellungsverfahren, das nie ausgeführt wurde, ist eine Hypothese, und das gilt auch für den Code, der ein System Ausfälle überstehen lassen soll. Kapitel 15 behandelt, wie man ereignisgesteuerte Systeme testet, einschließlich der Frage, wie man die Wiederherstellungen, Failover und Replays übt, die dieses Kapitel beschrieben hat.

Kapitel 15 — Ereignisgesteuerte Systeme testen

Stellen Sie sich das Inventory-Team bei Parcelly am Freitag vor einem langen Wochenende vor. Die Suite ist grün: 400 Tests, wenige Minuten Laufzeit, eine Abdeckung, über die niemand streitet. Für den Handler von `OrderPlaced` gibt es einen Test, der ein Ereignis baut, den Handler aufruft und prüft, dass der Bestand um zwei gesunken ist. Er besteht jedes Mal. Die Tests verwenden einen In-Memory-Ersatz für den Broker, der jede Nachricht genau einmal, in Reihenfolge und sofort zustellt.

Am Samstag startet ein Deployment die Consumer neu. Die Handler laufen während des Aufwärmens langsamer als sonst, und einer überschreitet seinen Visibility-Timeout, sodass der Broker dieselbe Nachricht `OrderPlaced` an einen zweiten Consumer übergibt, während der erste sie noch verarbeitet. Beide lesen die Inbox, beide finden nichts, und beide reservieren den Bestand. Der Unique Constraint, der einen von beiden hätte scheitern lassen sollen, fehlte im echten Schema, und kein Test bemerkte es, weil die Tests auf einer handgeschriebenen Fake-Datenbank liefen, die nie Constraints durchsetzte und nie zwei Handler gleichzeitig ausführte. Die Suite hatte sich bei nichts, was sie prüfte, je geirrt. Sie prüfte nur nie das Eine, auf das es ankam.

Währenddessen schlägt ein Test in der Orders-Suite bei einem von 20 Läufen fehl. Das Team schaut nicht mehr hin. Er wartet zwei Sekunden auf ein eintreffendes Ereignis, und manchmal braucht das Ereignis drei.

Beide Geschichten haben dieselbe Wurzel: Die Tests beschreiben ein System, das es nicht gibt. Dieses Kapitel beantwortet eine praktische Frage. Wie testen Sie Code, dessen schlimmste Fehler nur bei Duplizierung, Verzögerung und Umsortierung auftreten, ohne eine Suite zu bauen, die langsam, instabil oder ignoriert ist?

Das Problem und warum es schwierig ist

Kapitel 1 argumentierte, dass der Happy Path derjenige ist, den man ohne Mühe testen kann und dem man am ehesten glaubt. Die Kapitel 2 bis 14 bauten dann Abwehrmaßnahmen gegen Ausfälle, die der Happy Path nie zeigt. Abwehrmaßnahmen sind Code, und Code, der nie unter dem Ausfall ausgeführt wird, den er behandelt, ist eine Hoffnung. Kapitel 3 führte ein Ausfallmodell mit einem Feld „verifiziert durch“ für jeden Eintrag ein. Dieses Kapitel ist der Ort, an dem dieses Feld ausgefüllt wird.

Drei Eigenschaften machen dieses Testen schwierig.

Die Ausfälle sind Interleavings, keine Eingaben. Ein Funktionstest liefert eine Eingabe und prüft eine Ausgabe. Der Fehler der doppelten Reservierung braucht ein bestimmtes Timing: zwei Zustellungen, die sich zeitlich überlappen, oder einen Commit, gefolgt von einem Absturz und einer erneuten Zustellung. Kein Eingabewert löst ihn aus. Der Test muss steuern, wann etwas geschieht, das heißt, das Test-Harness muss das Verhalten des Brokers, die Uhr und die Absturzpunkte besitzen.

Asynchronität verbirgt den Moment der Wahrheit. Nachdem ein Handler gelaufen ist, kann der Effekt Sekunden später in einem anderen Service eintreten. Ein Test kann nicht sofort prüfen, und er kann nicht wissen, wie lange er warten muss. Eine feste Zeit zu schlafen ist in beide Richtungen falsch: zu kurz, und der Test ist flaky, zu lang, und die Suite ist langsam.

Fakes lügen genau an den Stellen, auf die es ankommt. Idempotenz beruht auf einem Unique Constraint. Atomarität beruht auf einer Transaktion. Reihenfolge beruht auf der Partitionszuweisung. Ein Fake, der diese überspringt, liefert eine grüne Suite und keine Information. Realismus kostet Geschwindigkeit, die Kunst besteht also darin zu entscheiden, wo man ihn bezahlt.

Die Testpyramide anpassen

Die klassische Testpyramide gilt weiterhin: viele schnelle Tests unten, wenige langsame oben. Was sich ändert, ist, wofür jede Schicht da ist und welchen Fehler jede Schicht abfangen soll.

Schicht	Was läuft	Echt oder Fake	Fängt ab
Handler-Tests	Ein Handler gegen eine echte Datenbank	Echte Datenbank, kein Broker	Falscher Effekt, fehlende Inbox, gebrochene Atomarität, schlechter Umgang mit Duplikaten und Reihenfolge
Vertragstests	Ereignisformen zwischen Producer und Consumer	Überhaupt keine Services	Schemaänderungen, die einen Consumer brechen
Komponententests	Ein Service mit echtem Broker, Outbox und Relay	Echter Broker, andere Services als Stubs	Verdrahtung: Outbox zu Relay zu Topic, Acks, Retries, Dead-Lettering
Ablauftests	Einige Services zusammen, ein Ausfall eingeschleust	Alles echt, kleiner Maßstab	Servicesübergreifendes Verhalten wie Kompensationen und Timeouts
Produktions-Probes	Synthetische Ereignisse im Live-System	Echt	Deployment- und Konfigurationsprobleme, die keine Pre-Production-Umgebung zeigt

Zwei Entscheidungen in der Tabelle verdienen je einen Satz. Die Handler-Schicht verwendet eine echte Datenbank-Engine, weil die Inbox, das Schreiben in die Outbox und die Versionsprüfung alle von Constraints und Transaktionen abhängen, die ein Fake nachbauen müsste, und genau das Nachbauen hat die Inventory-Suite in die Irre geführt. Der Broker hingegen bleibt auf dieser Schicht außen vor. Ein Handler ist eine Funktion von einer Nachricht und einem Zustand zu einem neuen Zustand und einigen ausgehenden Nachrichten. Wenn Sie ihn direkt aufrufen können, können Sie ihn schnell testen.

Die Form der Pyramide folgt auch daraus, wo die Ausfälle liegen. Die meisten Abwehrmaßnahmen dieses Buches sind Logik auf Consumer- oder Producer-Seite, die die Handler-Schicht günstig erreicht. Nur die Verdrahtung braucht einen Broker, und dort ist die Zahl der Tests klein, weil jeder eine Eigenschaft der Konfiguration beweist und nicht der Geschäftslogik.

Handler testen

Ein Handler-Test sollte den Vertrag in der Sprache des Buches formulieren: Gegeben diese Nachricht und dieser Zustand, tritt der Effekt einmal ein, und das ausgehende Ereignis wird in die Outbox geschrieben. Dann be-

kommt derselbe Test einen zweiten Akt, in dem die Nachricht erneut eintrifft.

```
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
```

Die letzte Assertion ist die, die Teams vergessen. Ein Duplikat, das nicht doppelt reserviert, aber `StockReserved` zweimal ausgibt, richtet nachgelagert trotzdem Schaden an, und die Prüfung der Outbox fängt es ab.

Dieser Test deckt sequenzielle Duplikate ab, die die Prüfung auf dem schnellen Pfad in der Inbox behandelt. Der gefährliche Fall ist das nebenläufige Duplikat, bei dem zwei Worker beide die Prüfung bestehen. Um ihn zu testen, führt das Harness zwei Handler gleichzeitig aus und hält beide mit einer Barriere unmittelbar vor der Transaktion an, sodass das Interleaving erzwungen statt erhofft wird.

```
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
```

Dieser Test schlägt fehl, wenn jemand den Unique Constraint entfernt und sich auf die Prüfung „erst lesen, dann einfügen“ verlässt, genau die Regression, die im Review am schwersten zu erkennen ist. Kapitel 7 hat erklärt, warum der Constraint das Sicherheitsnetz ist. Ein solcher Test macht die Erklärung ausführbar.

Absturzpunkte funktionieren genauso. Ruft der Handler dort, wo der Ack stünde, `crashPoint("before-ack")` auf, kann ein Test diesen Punkt scharf schalten und dort einen simulierten Absturz auslösen.

```
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
```

Lesen Sie die Assertion im Block `on`. Sie prüft, dass der Commit den Absturz überlebt hat, was die halbe Geschichte ist. Die andere Hälfte ist die erneute Zustellung, die darauf folgt. Ein Absturztest, der nur die zweite Hälfte prüft, würde auch für einen Handler bestehen, der nie etwas festgeschrieben hat.

Duplikate, Umsortierung und Verzögerung einschleusen

Einzelne Tests kodieren die Fälle, an die Sie gedacht haben. Das Ausfallmodell enthält auch die, an die Sie nicht gedacht haben, deshalb würde ich eine Schicht von Tests ergänzen, die einen ganzen Ereignisstrom stören und prüfen, dass sich das Ergebnis nicht ändert.

Die Idee ist eine Konvergenzeigenschaft. Nehmen Sie einen aufgezeichneten, korrekt geordneten Strom für ein Aggregat. Berechnen Sie den erwarteten Endzustand, indem Sie die Ereignisse direkt falten, ohne Broker. Stellen Sie dann dieselben Ereignisse in einem gestörten Ablauf über den Handler zu, einige innerhalb eines Fensters durchgemischt, einige dupliziert, und prüfen Sie, dass der Endzustand dem erwarteten entspricht.

```
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
```

Führen Sie ihn in der CI für einige hundert Seeds aus. Zwei Details halten ihn ehrlich. Der Seed wird ins Log geschrieben, sodass ein Fehlschlag bei Seed 4172 einer ist, den Sie auf einem Laptop erneut ausführen können, statt einer Erzählung. Und das Referenzergebnis stammt aus einem anderen Codepfad als der Handler, sodass ein Fehler im Handler nicht zugleich in der Erwartung auftauchen kann.

Verzögerung ist die dritte Störung. Eine verzögerte Nachricht ist eine, die nach ihren Nachfolgern eintrifft oder nachdem ein Timeout ausgelöst wurde. Hier gilt es, die Zeit zu kontrollieren: Handler und Sagas, die die Sys-

temuhr direkt lesen, lassen sich nicht auf Timeouts testen, ohne zu warten. Ich würde die Uhr als Abhängigkeit hineingeben und den Test sie vorstellen lassen. Kapitel 10 hat die Zustände aufgelistet, in denen sich eine Saga befinden kann. Ein Saga-Test wird dann zu einer Tabelle: Für jeden Zustand jedes Ereignis zustellen, das eintreffen könnte, auch die verspäteten, und den resultierenden Zustand und die ausgegebene Kompensation prüfen.

Saga-Zustand	Eintreffendes Ereignis	Erwartetes Ergebnis
`AWAITING_RESULTS`	`PaymentAuthorized`	Bleibt in `AWAITING_RESULTS` und wartet auf den Bestand
`AWAITING_RESULTS`, Zahlung gehalten	Timeout löst aus	`COMPENSATING`; die Commands `ReleaseStock` und `VoidAuthorization` werden in die Outbox geschrieben
`COMPENSATING` oder `CANCELLED`	Verspätetes `StockReserved`	Kein Saga-Übergang. Ein verspätetes `StockReserved` bedeutet, dass Inventory reserviert hat, bevor `ReleaseStock` es erreichte, sodass dieses Command den Bestand freigibt. Wäre die Freigabe zuerst eingetroffen, hätte ihr Tombstone die Reservierung abgelehnt (Kapitel 10)
`COMPLETED`	Doppeltes `PaymentCaptured`	Ignoriert

Die dritte Zeile ist die, auf die es ankommt. Verspätetes Eintreffen nach einem Timeout ist der Fall, den Sagas falsch machen, weil der Happy Path und der Timeout-Pfad jeweils einzeln getestet wurden und die Kollision nie. Die Tabelle muss von jemandem geschrieben werden, der fragt „was könnte in diesem Zustand sonst noch eintreffen?“, und nicht von jemandem, der dem Flussdiagramm folgt.

> **Hinweis:** Wenn Ihre Tests auf einem randomisierten Ablauf beruhen, bewahren Sie einige fehlschlagende Seeds als feste Regressionsfälle auf, sobald sie gefunden sind. Ein Fehler, der einmal reproduziert wurde, sollte reproduziert bleiben.

Vertragstests und Kompatibilitätsprüfungen

Handler-Tests bauen ihre eigenen Ereignisse, das heißt, sie testen den Consumer gegen die Vorstellung des Consumers vom Producer. Ändert der Producer ein Feld, bemerkt das in der Suite des Consumers nichts. Kapitel 12 hat Kompatibilitätsregeln und eine Schema-Registry beschrieben, die sie durchsetzt. Das deckt die Struktur ab. Vertragstests decken den Teil ab, den die Registry nicht sehen kann: wovon jeder Consumer tatsächlich abhängt.

In einem **Consumer-Driven Contract** schreibt jeder Consumer die Felder und Typen auf, die er aus einem Ereignis liest, und die Pipeline des Producers prüft, dass sein Schema sie weiterhin bereitstellt. Die Richtung ist wichtig. Der Producer kann nicht wissen, welche Felder tragend sind, der Consumer schon. Ein Vertrag ist deshalb eine kleine Datei, die dem Consumer gehört und dort abgelegt wird, wo die CI des Producers sie lesen kann.

```
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)
```

Es braucht zwei Hälften. Die Pipeline des Producers führt `verifyContracts` gegen jeden hinterlegten Vertrag aus. Die Pipeline des Consumers dekodiert jedes Beispielereignis seiner eigenen Verträge mit seinem echten Deserializer und führt den Handler darauf aus. Zusammen fangen sie einen Producer ab, der ein Feld nicht mehr liefert, und einen Consumer, der beginnt, ein Feld zu lesen, das er nie deklariert hat. Der

zweite Fang gilt nur, wenn das Beispiel aus dem Vertrag gebaut wird und nur die deklarierten Felder enthält; ein vom vollständigen Ereignis des Producers kopiertes Beispiel lässt den Consumer ein nicht deklariertes Feld lesen und trotzdem bestehen.

Die Beispiele sollten jede Version abdecken, die noch in der Aufbewahrung liegt, wie Kapitel 12 es empfohlen hat. Ein Replay liest, was da ist, und ein Consumer, der nur die neueste Form behandelt, scheitert am ersten alten Ereignis. Hierhin gehört auch die Kompatibilitätsprüfung der Registry in der CI: Eine Schemaänderung des Producers sollte den Build fehlschlagen lassen, bevor sie registriert werden kann, nicht erst, nachdem sie den Topic erreicht hat.

Seien Sie ehrlich über die Grenze. Ein Vertrag beweist, dass `amount_minor` existiert und eine Ganzzahl ist. Er beweist nicht, dass es noch „den Betrag in kleinsten Währungseinheiten“ bedeutet. Eine Änderung der Bedeutung ohne Änderung der Struktur besteht jeden Vertragstest, und nur ein menschlicher Reviewer oder ein Verhaltenstest im Consumer bemerkt sie. Ich würde grüne Verträge nicht als Aussage über die Bedeutung behandeln.

Aussagen bei Eventual Consistency

Ein End-to-End-Test veröffentlicht etwas und prüft dann einen Effekt an anderer Stelle. Der Effekt tritt irgendwann ein, die Assertion braucht also eine Form, die das toleriert. Die Standardform ist ein Polling mit 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 AssertionTimeout(observation.description)
        sleep(interval)
```

Die Struktur ist wichtiger, als sie aussieht. Die Prüfung läuft nach Erreichen der Deadline noch einmal, sodass eine Bedingung, die während des letzten Schlafens erfüllt wurde, trotzdem besteht; das Intervall ist kurz im Verhältnis zum Timeout, und die Fehlermeldung enthält die letzte Beob-

achtung. Eine Meldung „Timeout“ sagt Ihnen nichts. Eine, die sagt „order-1 hängt bei AWAITING_RESULTS, 1 gehaltenes Ereignis“, sagt Ihnen, wo Sie suchen müssen.

Instabile asynchrone Tests haben eine kurze Liste von Ursachen, und die meisten lassen sich an der Ursache beheben.

- **Feste Sleeps.** Durch `eventually` ersetzen. Der Timeout wird zur Obergrenze, und der Test endet, sobald die Bedingung gilt.
- **Auf einen schnellen Laptop abgestimmte Timeouts.** Setzen Sie die Obergrenze großzügig, denn sie kostet nichts, wenn die Bedingung schnell erfüllt ist, und halten Sie sie getrennt von Performance-Erwartungen. Ein Latenzbudget ist ein anderer Test mit einer anderen Umgebung.
- **Geteilter Zustand zwischen Tests.** Eine übrig gebliebene Nachricht in einem gemeinsamen Topic lässt den nächsten Test ein Ereignis sehen, das er nicht veröffentlicht hat. Geben Sie jedem Test eine eigene Correlation-ID und filtern Sie danach, oder einen eigenen Topic oder eine eigene Warteschlange.
- **Reihenfolge angenommen, wo keine versprochen ist.** Prüft ein Test die Reihenfolge zweier Ereignisse aus verschiedenen Partitionen, prüft er etwas, das der Broker nie versprochen hat (Kapitel 9).
- **Prüfen, dass nichts geschehen ist.** „Es wurde keine zweite E-Mail gesendet“ lässt sich nicht per Polling zum Erfolg bringen, denn längeres Warten könnte immer noch eine zeigen. Bevorzugen Sie es, das System in die Ruhe zu treiben und dann zu prüfen. Ruhe (Quiescence) ist eine Eigenschaft des gesamten Pfades, nicht eines Consumers: Die Outbox ist leer, der Relay ist untätig, der Lag jedes beteiligten Consumers ist null, kein Retry und kein Timer ist geplant, und die Retry- und Verzögerungswarteschlangen sind leer. Erst dann bleibt nichts mehr übrig, was den Effekt hervorbringen könnte, auch nicht ein nachgelagerter Hop, dessen Nachricht noch unterwegs ist.

Für den letzten Fall braucht das Harness eine Möglichkeit zu sagen „laufe, bis nichts mehr aussteht“. In einem Handler- oder Komponententest kann das Harness den Consumer so lange antreiben, bis die Warteschlange abgebaut ist und die eingeschleuste Uhr keine Timer mehr auszulösen hat. Das ist schneller und deterministischer als das Warten auf Wall-Clock-Zeit,

und es ist der Hauptgrund, warum ich die meisten Verhaltenstests auf der Handler-Schicht halte und echte Zeit nur dort verwende, wo die Verdrahtung der Testgegenstand ist.

Fehlerinjektion und Chaos

Alles bisher testet Code in Isolation. Die nächste Frage ist, ob sich das zusammengesetzte System so verhält, wie das Ausfallmodell behauptet. Fehlerinjektion bringt absichtlich einen Ausfall in eine laufende Umgebung und prüft die Reaktion des Systems.

Beginnen Sie mit dem Ausfallmodell aus Kapitel 3 und wählen Sie Einträge, nicht Werkzeuge. Schreiben Sie für jeden die Hypothese in den Begriffen des Eintrags auf: „Wird ein Consumer zwischen Commit und Ack beendet, ändert die erneute Zustellung nichts.“ Injizieren Sie ihn dann: den Prozess in diesem Moment beenden, den Relay anhalten, die Antworten des Brokers verzögern, die Verbindung zwischen einem Service und seiner Datenbank kappen, einen Broker-Knoten mitten im Veröffentlichen neu starten. Prüfen Sie danach die Invarianten, nicht die Logs: Der Bestand stimmt mit den Reservierungen überein, das Hauptbuch hat einen Eintrag pro Einzug, keine Outbox-Zeile hängt fest.

Nehmen Sie Wiederherstellungsübungen in diese Liste auf, denn eine Abwehr, die nie ausgeführt wurde, ist eine Hoffnung. Ein Backup, das nie wiederhergestellt wurde, ist das klassische Beispiel. Stellen Sie eines in einer Scratch-Umgebung wieder her und prüfen Sie, was eine Wiederherstellung mit der Outbox, der Inbox und den Consumer-Offsets macht, was Kapitel 14 behandelt: wiederauferstandene Outbox-Zeilen, zurückgespulte Inbox-Zeilen, sodass Duplikate nicht mehr erkannt werden, Offsets, die nicht mehr passen. Führen Sie ein Failover auf einen Standby-Cluster durch oder proben Sie es, und vergleichen Sie die Wiederherstellungszeit und die verlorenen Daten mit den Recovery-Zielen, die für diesen Ablauf in Kapitel 14 gewählt wurden. Und testen Sie das Aufholen: einen Consumer lange genug anhalten, um ein Backlog aufzubauen, ihn fortsetzen, während weiter Traffic eintrifft, und prüfen, dass das Backlog vor der Deadline des Ablaufs und vor dem Ende der Aufbewahrung abgebaut ist. Das prüft die Abflussrate aus Kapitel 13 und die Grenzen für laufende Arbeit aus Kapitel 6.

Ich würde das in Schichten mit wachsendem Auswirkungsbereich ausführen. Ablauftests in einer dedizierten Umgebung injizieren Ausfälle nach Plan in der CI, klein und wiederholbar. Ein regelmäßiger Game Day in einer größeren Umgebung injiziert sie als Teamübung, teils um Fehler zu finden, teils um herauszufinden, ob jemand es bemerkt. Die Alarme aus Kapitel 13 stehen genauso unter Test wie der Code. Wenn Sie einen Consumer beendet haben und kein Alarm ausgelöst wurde, haben Sie einen Fehler im Monitoring gefunden.

Chaos in Produktion ist ein möglicher nächster Schritt, kein Ausgangspunkt. Es ergibt nur Sinn, wenn das System gezeigt hat, dass es denselben Ausfall in der Pre-Production übersteht, wenn sich der Auswirkungsbereich begrenzen lässt und wenn jemand das Experiment sofort stoppen kann. Ein Team ohne diese Voraussetzungen sollte dort nicht beginnen, und ich würde mich nicht schämen, lange bei Game Days zu bleiben. Der Wert einer Praxis der Fehlerinjektion liegt im Zyklus aus Hypothese, Injektion und Prüfung, nicht darin, wie dramatisch die Injektion ist.

> **Warnung:** Ein Lauf mit Fehlerinjektion, der sich einen Broker, eine Datenbank oder einen Zahlungsanbieter-Account mit echtem Traffic teilt, ist ein Ausfall mit Plan. Isolieren Sie die Umgebung oder beschränken Sie die Injektion auf synthetischen Traffic, bevor Sie irgendetwas abschalten.

Replay als Regressionstest

Ein Ereignis-Log ist eine Aufzeichnung dessen, was tatsächlich geschehen ist, und der Replay, das gezielte erneute Lesen der Historie, macht daraus eine Regressionssuite. Nehmen Sie einen Ausschnitt echter Ereignisse, lassen Sie ihn in einer Scratch-Umgebung durch die neue Version eines Consumers laufen und vergleichen Sie den abgeleiteten Zustand mit dem, den die alte Version erzeugt hat. Unterschiede sind entweder beabsichtigt, wie bei einem Bugfix, oder Regressionen. Der Replay hat auch einen operativen Nutzen, nämlich die Wiederherstellung nach einer Lücke oder einem Restore durch erneutes Lesen der Historie, und Kapitel 14 behandelt das. Dieser Abschnitt handelt nur davon, es zum Testen zu nutzen.

Das fängt Dinge ab, die handgeschriebene Fixtures nie finden werden: Ereignisformen von vor drei Schemaversionen, Sequenzen, die sich niemand vorgestellt hat, ein Feld, das in den Tests immer vorhanden war und in einem kleinen Teil der Historie null ist. Es ist auch das Nächstliegende an einem Test der Behauptung aus Kapitel 12, dass alte Ereignisse weiterhin dekodiert werden.

Ein praktikables Verfahren hat fünf Schritte.

1. **Einen Ausschnitt wählen.** Ein Zeitfenster oder die vollständige Historie einer Stichprobe von Aggregaten. Eine Stichprobe, die jede Schemaversion plus bekannte Vorfälle enthält, schlägt eine zufällige.
2. **Maskieren.** Wiedergegebene Ereignisse enthalten Käuferdaten. Anonymisieren Sie, bevor sie ihre ursprüngliche Grenze verlassen, oder halten Sie die Scratch-Umgebung unter denselben Kontrollen wie die Produktion.
3. **Effekte und ausgegebene Ereignisse deaktivieren.** Der wiedergebende Consumer darf keine Karten belasten, keine E-Mails senden und keine Paketdienste aufrufen. Effekte gehen an Stubs, die aufzeichnen, was von ihnen verlangt wurde, und die Aufzeichnungen der Stubs sind Teil der Ausgabe. Seine Outbox und sein Relay müssen an einen Scratch-Broker veröffentlichen oder abgefangen werden und dürfen nie auf Produktions-Topics zeigen, sonst veröffentlicht der Replay die Historie erneut an Live-Consumer.
4. **Die alte und die neue Version auf demselben Ausschnitt ausführen.** Vergleichen Sie abgeleiteten Zustand und ausgegebene Ereignisse, Aggregat für Aggregat.
5. **Jeden Unterschied klassifizieren.** Beabsichtigt, harmlos oder Regression. Ein Unterschied, den niemand erklären kann, ist eine Regression, bis das Gegenteil gezeigt ist.

Der Replay gilt für die Log-Familie, in der die Historie noch vorhanden ist. In einer klassischen Warteschlange sind konsumierte Nachrichten weg. Das Äquivalent ist eine aufgezeichnete Stichprobe: ein Tap, der einen Bruchteil der Nachrichten anonymisiert in einen Store kopiert, den Sie kontrollieren. Es ist ein schwächeres Werkzeug, weil die Aufzeichnung nur so gut ist wie die Stichprobe, aber es dient demselben Zweck.

Der Replay hat auch eine Kostengrenze. Große Historien brauchen echte Zeit und Speicher, und das Vergleichswerkzeug muss Ihr Team bauen und am Laufen halten. Ich würde den Replay der vollständigen Historie Änderungen vorbehalten, die berühren, wie Ereignisse interpretiert werden, und für Routine-Releases einen kleineren Ausschnitt verwenden.

In Produktion verifizieren

Keine Pre-Production-Umgebung bildet Traffic-Form, Daten, Konfiguration und Eigenheiten der Produktion nach. Manche Ausfälle, etwa eine fehlende Berechtigung auf einem Topic nach einem Deployment, ein veraltetes Secret oder eine fehlgeleitete Warteschlange, zeigen sich nur im echten System. Die Antwort ist nicht, die Pre-Production-Tests aufzugeben, sondern eine letzte Schicht hinzuzufügen, die dort läuft, wo die Ausfälle sind.

Ein **synthetisches Ereignis** ist ein absichtlich eingeschleuster, ausdrücklich gekennzeichneter Traffic, der durch die echte Pipeline läuft und am Ende geprüft wird. Bei Parcelly ist es eine Bestellung, die eine Probe alle paar Minuten aufgibt und die von `OrderPlaced` bis `ShipmentScheduled` fließt.

```
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)
```

Die Kennzeichnung ist der Sicherheitsmechanismus, und sie ist ein eigener Header `synthetic`, kein Präfix auf `correlation_id`. Ein Bezeichner, der zugleich Verhalten steuert, ist eine Falle, und Kapitel 13 verlangt ein einheitliches Bezeichnerformat, `order-81724`, damit die Suche nach Korrela-

tion überall funktioniert. Der Preis ist, dass ein Hop, der vergisst, den Header zu kopieren, eine Probe ohne jeden Fehler in echten Traffic verwandelt. Jeder Service kopiert ihn mit demselben Helfer, der `correlation_id` kopiert, die Probe verwendet außerdem ein dediziertes synthetisches Käuferkonto, das Payments und Shipping unabhängig erkennen, und ein Test lässt den Header absichtlich weg und prüft, dass keine echte Belastung und keine echte Sendung entsteht.

Jeder Service erkennt das Flag und ändert seine Ränder, nicht seine Kernlogik: Payments spricht mit der Sandbox des Anbieters, Shipping überspringt die Übergabe an den Paketdienst, Notifications unterdrückt die Zustellung, und Ledger bucht auf ein Testkonto, das die Finanzabteilung aus Berichten ausschließt. Diese Zweige sind Code, der nur für Proben läuft, ein Fehler darin ist also ein Fehler, den nur die Produktion aufdeckt. Inventory braucht dieselbe Behandlung, sonst reserviert jede Probe echten Bestand: gegen eine dedizierte synthetische SKU reservieren oder die Reservierung für gekennzeichnete Bestellungen überspringen. Ich würde die Zweige selbst auf der Handler-Schicht testen und die Probe testen, indem ich absichtlich etwas kaputt mache und prüfe, dass sie fehlschlägt.

Das Aufräumen birgt eine weitere Falle. Eine Probe, die in ein Timeout läuft, hat nicht zwingend nicht fertig gestellt; ihre Ereignisse können noch unterwegs sein, und ein verspätetes `StockReserved` oder `PaymentAuthorized` kann nach der Stornierung eintreffen. Deshalb wird die Probe vorab aufgezeichnet und jeder Service richtet sich nach dem Flag `synthetic`: Ein periodischer Sweep kann Proben finden, die älter sind als das Budget, und stornieren, was sie hinterlassen haben, und die Stornierung ist selbst idempotent, sodass ein zweimaliges Ausführen harmlos ist.

Eine Probe braucht außerdem ein Budget, und das Budget braucht einen Verantwortlichen. Eine fehlschlagende Probe alarmiert jemanden, das Budget muss also großzügig genug sein, um Fehlalarme zu vermeiden, und knapp genug, um aussagekräftig zu sein. Kapitel 13 behandelt die End-to-End-Latenz und wie man darauf alarmiert. Die Probe ist die günstigste Quelle dieser Messung, weil sie den gesamten Pfad durchläuft, auch die Teile, die in der letzten Stunde keine echte Bestellung berührt hat.

Die Verifikation in Produktion umfasst die Abgleichjobs aus Kapitel 13. Ein Abgleichlauf ist ein Invariantentest über echte Daten, der fortlaufend ausgeführt wird, und er fängt die Ausfälle ab, die niemand zu simulieren gedacht hat. Ich zähle ihn zur letzten Schicht der Pyramide.

Abwägungen und Kosten

Realismus kostet Zeit. Echte Datenbanken, echte Broker und eingeschleuste Ausfälle machen Tests langsamer. Die Suite, die schnell war, war schnell, weil sie übersprang, worauf es ankam. Ich würde die langsamen Schichten ausdrücklich budgetieren: Handler-Tests in Sekunden, Komponententests in einigen Minuten, Ablauftests und Fehlerinjektion nach Plan statt bei jedem Commit.

Vertragstests erhöhen den Koordinationsaufwand. Jeder Consumer besitzt Dateien, die in der Pipeline des Producers liegen, und jemand muss sie aktuell halten. Für ein System mit zwei Services und einem Team kann ein gemeinsamer Test des Schemas genügen. Vertragstests beginnen sich zu lohnen, wenn Producer- und Consumer-Teams unabhängig ausrollen, und davor bringen sie wenig.

Randomisierte Abläufe erhöhen den Untersuchungsaufwand. Ein fehlschlagender Seed ist ein Rätsel und ein echter Defekt zugleich. Teams, die das nicht mögen, senken am Ende die Zahl der Seeds, bis der Test nichts mehr findet. Das Gegenmittel ist, Seeds ins Log zu schreiben und feste Regressionen zu behalten, nicht, die Technik aufzugeben.

Chaos und Proben haben einen Auswirkungsbereich. Beide berühren Live- oder gemeinsam genutzte Systeme. Wann man sie nicht einsetzt: wenn das System den Ausfall in Isolation noch nicht übersteht, wenn niemand den Lauf sofort stoppen kann oder wenn sich der gekennzeichnete Traffic nicht von echtem Geld und echten Käufern fernhalten lässt.

Replay ist aufwendig einzurichten. Die Infrastruktur für Maskierung, Effekt-Stubs und Vergleich ist ein eigenes Projekt. Ein kleines System mit kurzer Historie hat mehr von Handler-Tests und Absturzpunkten. Ich weiß nicht, wo Ihr Break-even-Punkt liegt, denn er hängt davon ab, wie oft sich Consumer ändern und wie teuer eine falsche Interpretation der Historie wäre, aber ein Team, das bereits einen Vorfall durch ein unerwartetes altes Ereignis hatte, kann ihn meist erraten.

Häufige Fehler

- **Gegen einen Broker-Fake testen, der sich nie danebenbenimmt.** Es ist verlockend, weil er schnell ist und die ersten Tests bestehen. Lassen Sie den Fake duplizieren, verzögern und umsortieren, oder bringen Sie den Ausfall mit expliziten Aufrufen in die Handler-Tests, wie in diesem Kapitel.
- **Die Constraints der Datenbank faken.** Ein handgeschriebener Fake setzt weder Eindeutigkeit noch Isolation noch Atomarität durch, die Tests bestehen also, ob die Abwehr existiert oder nicht. Verwenden Sie die echte Engine, in einem Container oder einer Scratch-Instanz, für jeden Test, der von einem Constraint abhängt.
- **Eine feste Zeit schlafen.** Es sieht wie der einfachste Weg zu warten aus. Fragen Sie die Bedingung mit einer Obergrenze per Polling ab und melden Sie bei einem Fehlschlag die letzte Beobachtung.
- **Den Absturz zwischen Commit und Ack nie testen.** Er wirkt exotisch, passiert aber bei jedem Deployment. Fügen Sie jedem Handler einen Absturzpunkt hinzu und einen Test, der ihn scharf schaltet.
- **Nur den Endzustand testen, nicht die Outbox.** Ein Duplikat, das den Zustand nicht ändert, kann trotzdem ein zweites Ereignis ausgeben. Prüfen Sie, was in die Outbox geschrieben wurde.
- **Vertragstests, die Vorhandensein prüfen, aber nicht Pflichtigkeit.** Ein Feld, das im Schema des Producers optional ist, kann fehlen. Lassen Sie den Vertrag „required“ sagen und prüfen Sie es.
- **Randomisierte Tests ohne Seeds.** Ein Fehlschlag, der sich nicht reproduzieren lässt, wird gelöscht. Schreiben Sie den Seed ins Log und behalten Sie fehlschlagende Seeds als feste Fälle.
- **Einem Backup vertrauen, das nie wiederhergestellt wurde.** Der Backup-Job meldet Erfolg, die Abwehr fühlt sich also getestet an. Stellen Sie es nach Plan in einer Scratch-Umgebung wieder her und prüfen Sie danach Outbox, Inbox und Offsets.
- **Proben gegen echtes Geld laufen lassen.** Synthetischer Traffic, der den Live-Zahlungsanbieter oder einen echten Paketdienst erreicht, ist keine Probe. Leiten Sie gekennzeichneten Traffic am Rand in Sandboxes und testen Sie diese Weiterleitung.

Zentrale Erkenntnisse

- Tests, die einen Broker annehmen, der jede Nachricht genau einmal und in Reihenfolge zustellt, testen ein System, das es nicht gibt, daher braucht jede Abwehr in diesem Buch einen Test, der den Ausfall anwendet, gegen den sie schützt.
- Handler-Tests gehören auf eine echte Datenbank-Engine, weil die Inbox, die Outbox und die Versionsprüfungen von Constraints und Transaktionen abhängen, die Fakes nicht nachbilden.
- Duplikate, Umsortierung und Verzögerung testet man am besten als Konvergenzeigenschaft über einen gestörten Strom, mit ins Log geschriebenem Seed, damit sich jeder Fehlschlag reproduzieren lässt.
- Consumer-Driven Contracts fangen strukturelle Brüche vor dem Deployment ab, aber sie sehen keine Änderung der Bedeutung, daher bleibt ein menschliches Review dessen, was ein Feld aussagen soll, Teil des Prozesses.
- Ersetzen Sie feste Sleeps durch Polling gegen eine Bedingung und eine Obergrenze, prüfen Sie auf Ruhe statt auf Stille und geben Sie die Uhr hinein, damit sich Timeouts testen lassen, ohne zu warten.
- Fehlerinjektion sollte von den Einträgen des Ausfallmodells ausgehen und danach Invarianten prüfen, den Auswirkungsbereich nur erweitern, wenn das System zeigt, dass es jeden Schritt übersteht, und Wiederherstellungsübungen für Restores, Failovers und das Aufholen einschließen.
- Der Replay echter Historie und synthetische Proben in Produktion fangen ab, was handgeschriebene Fixtures und Pre-Production-Umgebungen nicht können, und sie brauchen Maskierung, Effekt-Stubs und gekennzeichneten Traffic, um sicher zu sein.

Einem System, das so getestet wurde, kann man ein wenig mehr vertrauen, wenn auch nie vollständig, und damit bleibt die Frage, was ein Team prüfen sollte, bevor es einem Entwurf überhaupt vertraut. Kapitel 16 fasst das Buch in einer Zuverlässigkeits-Checkliste zusammen und schließt damit, was die Checkliste nicht leisten kann.

Kapitel 16 — Eine Checkliste für Zuverlässigkeit und Schlussgedanken

Stellen Sie sich ein Design-Review bei Parcelly an einem ruhigen Donnerstag vor. Das Team will einen neuen Ablauf hinzufügen: Verkäufer können Teilerstattungen anbieten, und Payments sendet dafür ein neues Ereignis. Das Diagramm ist sauber. Jemand hat die Pfeile gezeichnet, das Ereignis benannt und die Consumer aufgelistet. Das Meeting ist auf 30 Minuten angesetzt, und es sieht so aus, als wäre es nach 20 vorbei.

Dann stellt die technische Leitung eine schlichte Frage: „Was passiert, wenn die Erstattung ausgelöst und das Ereignis nie veröffentlicht wird?“ Im Raum wird es still, nicht weil niemand das Pattern kennt, sondern weil niemand danach gefragt hatte. Die Outbox gibt es an anderer Stelle in der Codebasis. Den idempotenten Consumer gibt es dort ebenfalls. Ob dieser neue Ablauf sie verwendet, ist eine andere Frage, und das Diagramm gibt darauf keine Antwort.

So erodiert Zuverlässigkeit in der Regel. Das Team kennt die Patterns. Was verblasst, ist die Gewohnheit, bei jedem neuen Ablauf zu prüfen, ob jedes Pattern tatsächlich vorhanden ist. Die Frage dieses letzten Kapitels ist deshalb praktisch: Was würden Sie Ablauf für Ablauf fragen, um herauszufinden, ob das System seine Versprechen an einem schlechten Tag halten kann, und wer muss die Antworten verantworten?

Das Problem und warum es schwierig ist

Zuverlässigkeit ist eine Eigenschaft von Nahtstellen, und Nahtstellen sind in keiner einzelnen Datei sichtbar. Ein Producer kann korrekt sein, ein Consumer kann korrekt sein, und der Ablauf dazwischen kann trotzdem

Ereignisse verlieren, weil die Übergabe nie untersucht wurde. Kapitel 1 hat dieses Argument am Anfang des Buches vorgebracht, und die Kapitel seither haben die Nahtstellen eine nach der anderen durchgearbeitet.

Die Schwierigkeit besteht nun darin, dass sich das Wissen über 15 Kapitel verteilt. Niemand hat so viel während eines Reviews im Kopf. Unter Termindruck werden genau die Fragen übersprungen, deren Antwort lautet: „Es funktioniert, bis zu dem Tag, an dem es nicht mehr funktioniert“: das Deduplizierungsfenster, die verantwortliche Person für die Dead Letters, die Kompatibilitätsprüfung, der Alarm, der beim Alter auslöst und nicht bei der Anzahl.

Eine Checkliste ist die übliche Antwort, und sie hat bekannte Grenzen. Sie kann nicht denken. Wer einen Punkt nicht versteht, kann ihn trotzdem abhaken, und eine vollständig abgehakte Liste vermittelt dann falsche Sicherheit. Ich würde die folgende Liste als Anstoß für ein Gespräch verwenden, nicht als Kontrollpunkt, den ein Formular passieren kann. Jeder Punkt verweist auf das Kapitel, das erklärt, warum er da ist, sodass, wer anderer Meinung ist, die Begründung findet und die Prämisse anfechten kann.

Behandeln Sie jeden Punkt als Frage zu einem konkreten Ablauf, etwa dem Bestellablauf von Parcelly, und nicht zum System im Allgemeinen. „Wir verwenden eine Outbox“ ist keine Antwort. „Orders schreibt `OrderPlaced` in derselben Transaktion wie die Bestellzeile in seine Outbox“ ist eine.

Die Methode hinter der Liste

Die Liste ist keine neue Idee. Sie ist die These des Buches, auf ein Review angewandt: Zuverlässige ereignisgesteuerte Systeme werden um die **Ausfallsemantik** herum gebaut, nicht um Message-Broker: Was kann an einer gegebenen Nahtstelle ausfallen, wie würden Sie es beobachten, und welche Garantie überlebt den Ausfall?

Das ergibt drei Fragen, und jeder Punkt unten ist eine davon, auf eine Nahtstelle gestellt. Ein Punkt, der einen Ausfall benennt, fragt, was ausfallen kann. Ein Punkt, der eine Metrik, einen Alarm oder einen Test benennt, fragt, wie Sie es beobachten würden. Ein Punkt, der eine Einstellung, einen Constraint oder ein Pattern benennt, fragt, welche Garantie

den Ausfall überlebt. Eine Nahtstelle, deren Punkte nur eine der drei Fragen beantworten, ist halb fertig, und eine Nahtstelle ohne Antwort auf die zweite Frage fällt aus, ohne es jemandem zu sagen.

Die Checkliste

Die Liste ist danach gruppiert, wo der Ausfall liegt. Die rechte Spalte nennt das Kapitel, das Sie erneut lesen sollten, wenn eine Antwort „nein“ oder „ich bin nicht sicher“ lautet.

Producer

Prüfpunkt	Kapitel
Die Zustandsänderung und das Ereignis werden in einer lokalen Transaktion geschrieben, ohne direktes Veröffentlichen im Geschäftspfad.	4
Die Outbox-Zeile enthält den vollständigen Nachrichtenumschlag, und die `event_id` wird beim Schreiben erzeugt und nie neu generiert.	4
Das Team hat festgehalten, dass die Veröffentlichung At-least-once erfolgt und dass Duplikate der Preis dafür sind.	4
Der Relay veröffentlicht zuerst und markiert danach, und er wählt nach Zustand aus, nicht über einen Cursor auf eine aufsteigende ID.	5
Mehrere Relays, falls vorhanden, beanspruchen Zeilen sicher und halten die Zeilen eines Aggregats auf einem Worker.	5
Das Alter der ältesten unveröffentlichten Zeile wird gemessen, und nur veröffentlichte Zeilen werden gelöscht.	5

Wenn ein Team in einem Review nur eine Frage beantworten kann, wünschte ich mir, dass es die erste ist, denn jede spätere Garantie setzt voraus, dass das Ereignis überhaupt existiert. Wo ein Ablauf keine Outbox braucht, weil seine Schritte eine lokale Transaktion teilen können, sollte das Team das sagen und begründen (Kapitel 1).

Broker

Prüfpunkt	Kapitel
Für jedes wichtige Ereignis hat das Team festgehalten, was „veröffentlicht“ bedeutet: die Replikations- und Flush-Einstellungen hinter einem Ack an den Producer.	2, 3

Prüfpunkt	Kapitel
Ack oder Offset-Commit erfolgt nach dem Effekt, sodass die Semantik At-least-once ist und kein versehentliches At-most-once.	2
Visibility-Timeouts, Session-Timeouts und die Retry-Zeit im Handler sind aufeinander abgestimmt, sodass langsame Arbeit nicht erneut zugestellt wird, solange sie noch läuft.	2, 6
Der Partitionsschlüssel identifiziert das Aggregat, und das Team weiß, welche Nachrichtenreihenfolge seine Broker-Familie bietet.	9
Jeder Producer verwendet in jeder Sprache denselben Partitioner und dieselbe Schlüsselkodierung, oder der Relay berechnet die Partition selbst.	9
Die Aufbewahrungsdauer übersteigt den längsten realistischen Ausfall plus die Zeit, ihn zu bemerken.	3, 13
Consumer werden beim Deployment ordnungsgemäß gestoppt, sodass Rebalance keine routinemäßige Quelle von Duplikaten ist.	3

Die meisten dieser Punkte sind Einstellungen, kein Code. Sie bleiben auf ihren Standardwerten, bis ein Incident zeigt, was die Standardwerte waren, und dann sind sie schwer nachzuverfolgen.

Consumer

Prüfpunkt	Kapitel
Die Inbox-Zeile und die fachliche Änderung werden in einer lokalen Transaktion festgeschrieben, mit einem Unique Constraint auf dem Schlüssel.	7
Das Deduplizierungsfenster überdauert Aufbewahrungsdauer, Redrive und geplanten Replay, und ein Constraint auf dem fachlichen Schlüssel deckt die Domänentabelle darüber hinaus ab.	7
Externe Seiteneffekte tragen einen deterministischen Idempotenzschlüssel, der aus der Arbeit abgeleitet wird, nicht aus dem Versuch.	7
Ausfälle werden als transient, dauerhaft oder unbekannt klassifiziert, und Retries verwenden Exponential Backoff mit Jitter, eine Deadline und ein gemeinsames Budget.	6
Die laufende Arbeit ist begrenzt, die Nebenläufigkeit sinkt, wenn die Abhängigkeit langsamer wird, und der Consumer pausiert, statt fehlzuschlagen, sodass sich Überlast als Consumer-Lag zeigt; ein Limit am Ingress weist neue Arbeit mit einem klaren Signal an den Aufrufer ab oder stellt sie zurück.	3, 6

Prüfpunkt	Kapitel
Eine Poison Message landet in einer DLQ, mit Body, Headern, Quellposition, Fehler, Anzahl der Versuche und Consumer-Version, veröffentlicht, bevor das Original per Ack bestätigt wird.	8
Ein Consumer, der an einem Datensatz hängt, wird erkannt, über Lag ohne Durchsatz und über einen Absturzzähler pro Position, und läuft nicht in einer Schleife, bis es jemand bemerkt.	8, 13
Der Redrive behält die ursprüngliche `event_id` bei, läuft in Batches mit Ratenbegrenzung und hat einen Pausenschalter.	8
Die Reihenfolge pro Aggregat wird durch Lanes mit Schlüssel, Versionsprüfungen und Hold-back-Buffer mit Timeouts geschützt.	9
Jeder Workflow-Schritt vor dem Pivot hat eine idempotente Kompensation, und jedes Warten hat eine Deadline.	10

Das ist die längste Gruppe, weil die meisten Patterns des Buches hier landen. Wenn sie sich schwer anfühlt, stimmt das: At-least-once-Zustellung bürdet dem Consumer die Last auf.

Vertrag

Prüfpunkt	Kapitel
Jedes Ereignis beschreibt einen Fakt, ist in der Vergangenheitsform benannt und hat genau einen zuständigen Service.	11
Das Integrationsereignis ist ein bewusster öffentlicher Vertrag, aus dem Domänenereignis abgebildet, ohne personenbezogene Daten oder Secrets im Payload.	11
Ein Command ist im Imperativ benannt, hat genau einen vorgesehenen Empfänger, trägt eine `command_id`, hat einen Antwortpfad, damit eine Ablehnung den Absender erreicht, und wird nie als Ereignis bezeichnet; der Absender wartet mit einer eigenen Deadline.	10, 11
`correlation_id` wird durch den Ablauf kopiert, und `causation_id` zeigt auf den direkten Auslöser.	11
Die Wire-Typen des Nachrichtenumschlags, etwa Zeitstempel, Identifikatoren und Geldbeträge, sind für jede Sprache festgelegt, die ihn liest oder schreibt.	11
Jede Änderung wird als kompatibel oder als Breaking Change eingestuft, und die Kompatibilität wird in der Deployment-Pipeline geprüft.	12
Eine Bedeutungsänderung wird von einem Menschen geprüft, weil kein Tool sie erkennt.	12

Prüfpunkt	Kapitel
Consumer ignorieren Felder, die sie nicht verwenden, und tolerieren Werte, die sie noch nicht gesehen haben, ein Hop, der eine Nachricht weiterleitet, bewahrt unbekannte Felder, und ein Ereignis mit einer ihnen unbekannten `schema_version` wird abgelehnt oder geparkt.	12
Breaking Changes werden als Folge von Schritten ohne Breaking Change migriert, mit im selben Dokument geplanten Rollbacks, und die Consumer, die noch die alte Version lesen, sind bekannt.	11, 12

Betrieb

Prüfpunkt	Kapitel
Trace-Kontext und `correlation_id` überstehen jeden asynchronen Hop, sodass die Geschichte einer Bestellung anhand ihres Identifikators rekonstruiert werden kann.	13
Consumer-Lag, Alter der Nachricht, Outbox-Alter, Retry-Rate und Zugänge in der DLQ werden gemessen, und Alarme lösen bei Alter und Rate aus.	13
Abgleichs-Jobs vergleichen die maßgebliche Quelle regelmäßig mit dem abgeleiteten Zustand, und Reparaturen werden vor der Ausführung geprüft.	13
Fachliche Invarianten, etwa „jede erfasste Zahlung hat einen Eintrag im Hauptbuch“, werden direkt geprüft.	13
Handler werden mit Duplikaten, Umsortierung und Verzögerung getestet, und Vertragstests laufen in der CI.	15
Fehlerinjektion und Replay-basierte Regressionstests gibt es für die wichtigsten Abläufe.	15
Jeder Store hat ein festgelegtes Recovery Point Objective (RPO) und Recovery Time Objective (RTO), und das Team weiß, welcher Store bei einer Wiederherstellung in der Zeit zurückspringt.	14
Eine Replay-Prozedur benennt die Consumer-Groups, den Zeitstempel samt Sicherheitsspanne, die Seiteneffekte, die sie unterdrückt, und das Deduplizierungsfenster, in das jeder wiedergegebene Bereich passen muss.	7, 14
Ein Plan für die Broker-Migration verwendet den Relay als einzigen Umschaltpunkt, veröffentlicht doppelt mit Deduplizierung im Consumer und hat Schritte für Umschaltung und Rollback.	14
Wiederherstellungen und Game Days wurden geübt, und die Runbooks unten wurden nach dem geschrieben, was die Übung gezeigt hat.	14, 15
Ein schriftliches Ausfallmodell benennt Verantwortliche, Tests und Review-Termine.	3

Fragen für ein Design-Review

Die Checkliste verifiziert ein Design, das existiert. Diese Fragen helfen dabei, eines zu formen, das noch nicht existiert. Ich würde sie in dieser Reihenfolge stellen, weil jede Antwort die nächste einschränkt.

1. **Was ist der Fakt, und wer verantwortet ihn?** Wenn Sie das Ereignis nicht in der Vergangenheitsform benennen und keinen zuständigen Service nennen können, ist das Design nicht reif für einen Broker (Kapitel 11).
2. **Braucht das überhaupt ein Ereignis?** Wenn die Schritte eine lokale Transaktion teilen können oder der Aufrufer sofort eine Antwort braucht, kann ein synchroner Aufruf das bessere Werkzeug sein (Kapitel 1).
3. **Was passiert zwischen dem Commit und der Veröffentlichung?** Fragen Sie nach dem Absturzfenster, nicht nach dem Pattern. „Wir veröffentlichen direkt nach dem Commit“ ist ein Verlust, der auf ein Deployment wartet (Kapitel 4).
4. **Was passiert, wenn diese Nachricht zweimal eintrifft, und was, wenn sie verspätet eintrifft?** Das sind verschiedene Ausfälle mit verschiedenen Lösungen (Kapitel 7 und 9).
5. **Was ist die Obergrenze für „schließlich“?** Nennen Sie eine Dauer, einen Alarm und einen für Nutzer sichtbaren Zustand für das Warten (Kapitel 1).
6. **Was passiert, wenn diese Nachricht nie erfolgreich verarbeitet werden kann?** Fragen Sie, wo sie landet, wer sie sieht und wie sie zurückkommt (Kapitel 8).
7. **Was passiert, wenn der Consumer oder die Abhängigkeit dahinter langsamer ist als der Producer?** Fragen Sie, was begrenzt ist, was pausiert und welche Arbeit wartet, und wer diese Entscheidung verantwortet (Kapitel 3 und 6).
8. **Wenn ein späterer Schritt fehlschlägt, was macht die früheren rückgängig?** Benennen Sie die Kompensation und den Pivot, nach dem nichts mehr rückgängig gemacht wird (Kapitel 10).
9. **Wie werden wir diesen Vertrag in einem Jahr ändern?** Wenn niemand darüber nachgedacht hat, wird die Antwort in einem Incident improvisiert (Kapitel 12).

10. **Woran merken wir, dass es kaputt ist, und woran merken wir, dass es repariert ist?** Erkennung und Verifizierung sind getrennte Fragen (Kapitel 13 und 15).
11. **Wenn ein Store, der Broker oder eine Region verloren geht, wie viel verlieren wir, und wie lange dauert es, bis wir wieder da sind?** Fragen Sie nach einem RPO und einem RTO pro Store und nach dem Replay oder der Wiederherstellung, die darauf folgt (Kapitel 14).
12. **Wer wird alarmiert, und was tut diese Person zuerst?** Wenn die Antwort ein Name ist, haben Sie eine verantwortliche Person. Wenn sie „das Team“ lautet, haben Sie vielleicht keine.

Für manche dieser Fragen gibt es an einem bestimmten Tag keine saubere Antwort, und das ist in Ordnung. „Wir wissen es noch nicht, und so werden wir es herausfinden“ ist ein besseres Ergebnis als eine selbstsichere Antwort, die niemand geprüft hat.

Die organisatorische Seite

Jedes Pattern in diesem Buch hat eine menschliche Hälfte. Eine Outbox braucht jemanden, der das Alter ihrer ältesten Zeile beobachtet. Eine DLQ braucht jemanden, der sie liest. Eine Schema-Registry braucht jemanden, der entscheidet, was „kompatibel“ bedeutet. Fehlt die menschliche Hälfte, existiert das Pattern weiterhin im Diagramm und erfüllt seine Aufgabe nicht mehr.

Verantwortung

Ich würde drei Arten von Verantwortung festhalten.

- **Verantwortung für Ereignisse.** Jedes Ereignis hat einen zuständigen Service und ein verantwortliches Team, die dokumentieren, was das Ereignis bedeutet und wer es konsumiert (Kapitel 11).
- **Betriebliche Verantwortung.** Jede DLQ, jeder Parking Lot, jeder Abgleichsbericht und jeder Alarm hat eine namentlich benannte verantwortliche Person. Ein Alarm, den niemand verantwortet, ist eine Benachrichtigung, und Benachrichtigungen werden ignoriert.
- **Abläufe über mehrere Services hinweg.** Der Bestellablauf von Parcelly durchquert sechs Services. Irgendein Team muss den Ablauf als

Ganzes verantworten, einschließlich seiner Timeouts und hängenden Workflows, auch wenn das kein einzelner Service tut (Kapitel 10).

Runbooks

Ein Runbook schreibt man in der Annahme, dass die lesende Person müde ist und das System seit sechs Monaten nicht gesehen hat. Sagen Sie für jeden Alarm, was er bedeutet, worauf zuerst zu schauen ist, welche Aktionen sicher sind, welche nicht, und wen man anrufen soll, wenn die ersten Schritte nicht helfen.

Manche Runbooks gehören zu einem Alarm und manche zu einem Ereignis, das so selten eintritt, dass im Team zuletzt niemand es durchgeführt hat. Kapitel 14 ergänzt die zweite Art. Sein Abschnitt über das Üben der Wiederherstellung listet acht Fragen auf, die ein Runbook für die Wiederherstellung beantworten muss, schriftlich festgehalten und nicht im Kopf von jemandem getragen:

1. Welcher Store ist betroffen, und welcher ist in der Zeit zurückgesprungen?
2. Was ist der Wiederherstellungspunkt, und wie wurde er bestimmt?
3. Welche Consumer-Groups müssen zurückgesetzt werden, auf welchen Zeitstempel und mit welcher Sicherheitsspanne?
4. Welche Effekte müssen während des Replays unterdrückt werden, und mit welchem Mechanismus?
5. Liegt jeder erneut zu lesende Bereich innerhalb des Deduplizierungsfensters jedes betroffenen Consumers?
6. Was wird pausiert, in welcher Reihenfolge, und wer setzt es fort?
7. Welche Abgleiche bestätigen, dass die Wiederherstellung funktioniert hat, und wer liest sie?
8. Wie wird das Ganze rückgängig gemacht, wenn es schiefgeht?

Die Tabelle listet die Ereignisse auf, für die ich ein Runbook schreiben würde. Die letzte Spalte markiert, welche der acht Fragen ich jeweils beantwortet sehen möchte. Sie ist ein Ausgangspunkt, und das eigene System eines Teams wird eigene Fragen hinzufügen.

Ereignis	Die gefährliche Aktion	Fragen
Ein Consumer, der an einer Poison Message hängt (Kapitel 8)	Die Position überspringen, ohne dass es dazu eine Aufzeichnung gibt, oder das Retry-Limit anheben, bis die Nachricht durchgeht	6, 7, 8
Ein Consumer, dessen Lag sich der Aufbewahrungsdauer nähert (Kapitel 13 und 14)	Die Nachrichten ungelesen ablaufen lassen oder vorwärts zurücksetzen, ohne dass jemand das entschieden hat	2, 3, 5, 7
Offset-Reset und Replay (Kapitel 14)	Einen Replay mit aktiven Seiteneffekten durchführen oder über das Deduplizierungsfenster hinaus	3, 4, 5, 6, 8
Redrive (Kapitel 8)	Eine ganze DLQ auf einmal in eine Abhängigkeit zurückschicken, die noch Mühe hat	5, 6, 8
Reparatur und Backfill (Kapitel 13)	Ein Skript ohne Review gegen den Zustand der Produktion ausführen	4, 7, 8
Eine Datenbank aus dem Backup wiederherstellen (Kapitel 14)	Consumer fortsetzen, ohne ihre Offsets zurückzusetzen, oder Identifikatoren wiederverwenden, die die zurückgespulte Sequenz erneut vergibt	1, 2, 3, 4, 5, 6, 7, 8
Datenverlust im Broker oder Failover (Kapitel 14)	Groups an Positionen fortsetzen, die nicht mehr bedeuten, was sie früher bedeuteten	1, 2, 3, 5, 6, 7
Evakuierung einer Region (Kapitel 14)	Ein Standby hochstufen, ohne zu wissen, wie weit es zurücklag	1, 2, 3, 6, 7, 8
Umschaltung bei der Broker-Migration (Kapitel 14)	Consumer umziehen, bevor die doppelt veröffentlichten Kopien dedupliziert sind, oder ohne Weg zurück	3, 5, 6, 8

Zwei davon verdienen besondere Sorgfalt. Der Redrive kommt zuerst, weil ein nachlässiger Redrive einen eingedämmten Ausfall in eine Flut verwandelt (Kapitel 8). Reparatur und Backfill kommen an zweiter Stelle, weil ein Skript, das abgeleiteten Zustand korrigiert, eine Änderung in der Produktion ist und ein Review, einen Dry Run und eine Aufzeichnung dessen verdient, was es getan hat (Kapitel 13).

Rufbereitschaft

Die Rufbereitschaft für ein ereignisgesteuertes System unterscheidet sich von der für einen Request-Response-Service in einem entscheidenden Punkt: Der Ausfall ist oft still. Ein synchroner Service liefert Fehler zurück, und ein Graph färbt sich rot. Ein ereignisgesteuertes System liefert wo-

möglich gar nichts zurück, weil die Nachricht in einer Warteschlange liegt und wartet. Alarme für Alter, Lag und unverarbeitete Arbeit sind das primäre Signal, und die diensthabende Person braucht genug Zugriff, um in die Warteschlange, die DLQ und die Outbox zu schauen, ohne auf eine Erlaubnis zu warten.

> **Hinweis:** Üben Sie die Reparatur, bevor Sie sie brauchen. Ein Team, das einmal, tagsüber und unter den Augen eines Kollegen, einen Redrive für eine Test-DLQ durchgeführt hat, wird die echte anders handhaben als ein Team, das nur das Runbook gelesen hat. Kapitel 14 macht denselben Punkt für Wiederherstellungen und Game Days, und Kapitel 15 behandelt, wie man die Übungen gestaltet.

Abwägungen und Kosten

Eine Checkliste hat Kosten, und das erste ist die Aufmerksamkeit. Eine Liste mit 43 Punkten, auf jeden Ablauf angewandt, ist zu lang, und ein Team, das alle auf einen Benachrichtigungsablauf mit geringem Einsatz anwendet, wird zu dem Schluss kommen, dass die Checkliste das Problem ist. Ich würde die Liste nach den Kosten des Ausfalls gewichten. Ein Ablauf, der Geld bewegt oder Bestände ändert, verdient die ganze Liste. Eine Erinnerungs-E-Mail verdient Idempotenz und eine verantwortliche Person für die DLQ, und vielleicht wenig mehr.

Die zweiten Kosten sind falsche Sicherheit. Eine vollständig abgearbeitete Liste sagt, dass jemand jeden Punkt bedacht hat, nicht dass das System korrekt ist. Die Kapitel 13, 14 und 15 gibt es, weil auch bedachte Designs auf Weisen ausfallen, die ihre Urheber sich nicht vorgestellt haben. Betrachten Sie die Checkliste als günstigste Verteidigungsschicht, nicht als letzte.

Die dritten Kosten sind Drift. Eine Liste, die einmal geschrieben wurde, wird zum Protokoll dessen, was das Team in einem Moment geglaubt hat. Gehen Sie sie nach jedem Incident erneut durch, und streichen Sie Punkte, die ihren Platz nicht mehr verdienen. Eine kurze Liste, die Menschen benutzen, ist mehr wert als eine lange, die Menschen abzeichnen.

Wann man sie nicht einsetzt: Verwenden Sie die Checkliste nicht als Ersatz für Urteilsvermögen, und verlangen Sie sie nicht für Wegwerf-Prototypen. Wenn ein Ablauf nächsten Monat gelöscht wird, übersteigen die Kosten der vollständigen Liste die Kosten des Ausfalls. Sagen Sie das ausdrücklich, und schreiben Sie das Datum auf, bis zu dem der Prototyp entweder stirbt oder ein ordentliches Review bekommt.

Häufige Fehler

- **Die Checkliste als Kontrollpunkt behandeln.** Ein Kontrollpunkt lässt sich prüfen, indem man Häkchen zählt, und das ist die Versuchung. Das Ergebnis ist ein ausgefülltes Formular und ein nicht besprochenes Design. Verwenden Sie die Punkte als Fragen, und halten Sie die Antworten fest, die jemanden überrascht haben.
- **Patterns prüfen statt Abläufe.** „Wir haben eine Outbox“ fühlt sich nach Fortschritt an. Fragen Sie, ob dieser Ablauf sie verwendet, und nennen Sie die Tabelle und die Transaktion.
- **Alles überall anwenden.** Es fühlt sich gründlich an, führt dann aber zu Ermüdung und Abbruch. Gewichten Sie die Liste nach den Kosten des Ausfalls, und sagen Sie, welche Punkte Sie überspringen und warum.
- **Die menschlichen Punkte auf später verschieben.** Verantwortliche, Runbooks und Review-Termine wirken weniger dringend als Code, und sie fehlen als Erstes, wenn nachts ein Alarm auslöst. Weisen Sie sie im Design-Review selbst zu.
- **Die Antworten nie erneut ansehen.** Ein Ablauf, der vor einem Jahr das Review bestanden hat, hat seither Consumer, Versionen und Menschen dazugewonnen. Versehen Sie die Antworten mit einem Review-Termin, wie es Kapitel 3 für das Ausfallmodell vorschlägt.
- **Glauben, die Liste sei vollständig.** Ihr System hat Ausfallarten, die dieses Buch nicht benennt. Nehmen Sie sie in Ihre eigene Liste auf, wenn Sie ihnen begegnen.

Zentrale Erkenntnisse

- Zuverlässigkeit erodiert häufiger durch übersprungene Fragen als durch unbekannte Patterns, daher schützt ein fester Satz von Fragen,

der bei jedem neuen Ablauf gestellt wird, ein Design besser als Expertise im Raum.

- Jeder Punkt der Checkliste ist eine der drei Fragen des Buches, was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall überlebt, auf eine Nahtstelle gestellt, daher sollte jeder Punkt für einen konkreten Ablauf beantwortet werden und nicht für das System im Allgemeinen.
- Gewichten Sie die Liste nach den Kosten des Ausfalls: Abläufe, die Geld oder Bestände bewegen, verdienen jeden Punkt, und Abläufe mit geringem Einsatz verdienen die wenigen, die doppelte Effekte und stillen Verlust verhindern.
- Jedes Pattern in diesem Buch hat eine menschliche Hälfte, daher gehören Verantwortliche, Runbooks, Empfänger von Alarmen und Review-Termine in dasselbe Design-Review wie der Code.
- Ausfälle in ereignisgesteuerten Systemen sind still, daher hängt die Rufbereitschaft von Alarmen für Alter, Lag und unverarbeitete Arbeit ab, und die seltenen Ereignisse, eine Wiederherstellung, ein Replay, ein Regionsverlust und eine Broker-Migration, brauchen Runbooks, die tatsächlich ausgeführt wurden.
- Eine vollständig abgearbeitete Checkliste zeigt, dass jemand über jeden Punkt nachgedacht hat, nicht dass das System korrekt ist, und deshalb gibt es Abgleich und Tests als weitere Schichten.
- Zuverlässigkeit ist kein Feature, das man einmal ausliefert, sondern eine Disziplin, die man weiter übt, in kleinen Fragen, schriftlichen Antworten und Reparaturen, die geprobt werden, bevor man sie braucht.

Parcelly wird nie zuverlässig in der Weise, wie ein Feature fertig wird. Es wird jedes Mal zuverlässiger, wenn jemand fragt, was passiert, wenn die Nachricht zweimal eintrifft, die Antwort aufschreibt und ihr eine verantwortliche Person gibt. Das bietet kein Broker an, und keine Bibliothek liefert es mit. Es ist eine Gewohnheit der Aufmerksamkeit, gepflegt von Menschen, die wissen, dass jedes System, das sie bauen, eines Tages von einem Netzwerk, einem Deployment oder einem müden Kollegen zweckentfremdet wird, und die beschlossen haben, ohne Drama für diesen Tag zu ent-

werfen. Das möchte ich Ihnen aus diesem Buch mitgeben: nicht eine Sammlung von Patterns, sondern die Gewohnheit, die nächste Frage zu stellen.

Anhang: Glossar und Kurzreferenz der Patterns

Dieser Anhang hat zwei Teile. Teil A ist ein Glossar der Begriffe, die das Buch verwendet, in alphabetischer Reihenfolge, mit dem Kapitel, in dem jeder Begriff am ausführlichsten definiert oder verwendet wird. Teil B ist eine Kurzreferenz der Patterns, geschrieben als Tabelle, die Sie während eines Design-Reviews überfliegen können.

Keiner der beiden Teile ersetzt die Kapitel. Eine Definition hier ist eine Erinnerung, und eine Tabellenzeile ist ein Verweis. Die Begründung, die Kosten und die Fälle, in denen ein Pattern nicht eingesetzt werden sollte, stehen in dem Kapitel, das in der letzten Spalte genannt wird.

A. Glossar

Abgleich. Die Sichtweisen zweier Systeme auf die Wahrheit vergleichen und die Unterschiede beheben. Kapitel 13.

Abwärtskompatibilität. Ein Consumer mit dem neuen Schema kann Ereignisse lesen, die mit dem alten Schema geschrieben wurden: neue Leser, alte Daten. Kapitel 12.

Ack (Acknowledgment). Das Signal des Consumers an den Broker, dass eine Nachricht verarbeitet wurde und aus der Warteschlange entfernt werden kann oder dass die Position des Consumers vorrücken darf. Die negative Form ist ein **Nack**. Kapitel 2.

Aggregat. Die Konsistenzgrenze, die einen Partitionsschlüssel besitzt. Ereignisse für ein Aggregat sind untereinander geordnet; Ereignisse für verschiedene Aggregate sind es nicht. Kapitel 9.

Aggregatversion. Ein Zähler für die Änderungen an einem Aggregat, gespeichert in der Outbox-Zeile und im Payload mitgeführt, etwa `order_version` für Orders. Er ordnet die Ereignisse eines Aggregats und

sagt nichts über die Form des Payloads. Er ist eine andere Zahl als die **Schemaversion** und die **Registry-Revision**. Kapitel 4 definiert sie; Kapitel 9 verwendet sie.

Alter der Nachricht. Die Zeit seit der Erzeugung der ältesten Nachricht, die noch wartet, verglichen mit einer fachlichen Deadline und mit der Aufbewahrungsdauer. Kapitel 13.

At-least-once. Eine Zustellsemantik, bei der eine Nachricht ein- oder mehrfach zugestellt wird, sodass Duplikate möglich sind. Sie gilt, solange die Nachricht noch aufbewahrt wird und der Consumer erst nach dem Effekt per Ack bestätigt. Ein Consumer, der über die Aufbewahrungsdauer hinaus zurückliegt, verliert dennoch unbemerkt Nachrichten. Sie ist der Standard des Buches. Kapitel 2 definiert sie; Kapitel 3 behandelt den Verlust.

At-most-once. Eine Zustellsemantik, bei der eine Nachricht null- oder einmal zugestellt wird, sodass eine Nachricht verloren gehen, aber nie zweimal zugestellt werden kann. Kapitel 2.

Aufbewahrungsdauer. Wie lange ein Broker eine Nachricht aufbewahrt, ob sie gelesen wurde oder nicht: ein Zeit- oder Größenlimit bei einem Log, ein Ablauf bei einer Warteschlange. Sie begrenzt Replay, Redrive und die Zeit, die ein zurückliegender Consumer ausfallen darf. Kapitel 2 definiert sie; Kapitel 3, 13 und 14 verwenden sie.

Ausfallmodell. Die schriftliche Liste der Nahtstellen eines Systems und, für jede, was ausfallen kann, wie es beobachtet würde und welche Garantie den Ausfall überlebt, mit Verantwortlichen, Tests und Review-Terminen. Kapitel 3.

Ausfallsemantik. Für eine gegebene Nahtstelle: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall überlebt. Sie ist die These des Buches. Das Vorwort nennt sie, Kapitel 3 wendet sie als Ausfallmodell an, und Kapitel 16 macht daraus die Checkliste für das Review.

Backfill. Eine Massenreparatur oder ein Neuaufbau, bei dem eine große Menge von Ereignissen in einen laufenden Consumer eingespeist wird, ausgeführt in kleinen Batches mit Ratenbegrenzung. Kapitel 13.

Backpressure. Die Arbeit in Bearbeitung an jeder Stufe begrenzen, sodass die Last endlich bleibt. In einem Broker-vermittelten System zeigt sie sich als Limits für die Aufnahme eines Consumers, etwa Prefetch oder Datensätze pro Poll, und als Lag, den der Broker abfängt. Druck erreicht einen Producer nur durch die Flusskontrolle des Brokers oder durch ausdrückliches Drosseln. Kapitel 3 ordnet sie im Ausfallmodell ein; Kapitel 6 liefert die Mechanismen.

causation_id. Das Feld im Nachrichtenumschlag, das auf das Ereignis oder Command zeigt, das dieses direkt verursacht hat, während `correlation_id` den ganzen Ablauf benennt. Kapitel 11.

Change Data Capture (CDC). Das Transaktionslog einer Datenbank lesen, um festgeschriebene Änderungen in Nachrichten zu verwandeln. Im Outbox-Kontext liest ein Relay, der auf CDC aufbaut, Outbox-Zeilen aus dem Log, statt die Tabelle per Polling abzufragen. Kapitel 5.

Circuit Breaker. Ein Schutz um eine Abhängigkeit, der Aufrufe stoppt, wenn Ausfälle einen Schwellenwert überschreiten, und nach einer Wartezeit Probeaufrufe durchlässt. Kapitel 6.

Command. Eine Aufforderung, etwas zu tun, im Imperativ benannt, etwa `CapturePayment`. Ein Command ist an einen Service adressiert und kann abgelehnt werden. Es ist kein Ereignis. Kapitel 11.

Consumer. Ein Prozess, der Nachrichten aus einem Topic oder einer Warteschlange liest und darauf reagiert. Kapitel 2.

Consumer-Driven Contract. Eine Datei im Besitz eines Consumers, die die Felder und Typen aufführt, die er aus einem Ereignis liest, und in der Pipeline des Producers geprüft wird. Kapitel 15.

Consumer-Group. Eine Menge von Consumern, die sich die Partitionen eines Topics teilen, sodass jede Partition jeweils von einem Mitglied gelesen wird. Kapitel 2.

Consumer-Lag. Wie weit ein Consumer hinter dem Head einer Partition oder Warteschlange zurückliegt, gemessen in Nachrichten oder in Zeit. Kapitel 2 definiert ihn; Kapitel 3 und 13 verwenden ihn.

Consumer-Outbox. Eine Outbox, die ein Consumer schreibt, der eine Folgenachricht veröffentlicht, in derselben Transaktion wie seine Inbox-Zeile und seine Zustandsänderung. Ein Absturz vor dem Commit hinterlässt nichts, und ein Absturz danach wird bei erneuter Zustellung von der Inbox aufgefangen. Kapitel 7 führt die Regel ein; Kapitel 10 wendet sie auf Saga-Schritte an.

correlation_id. Das Feld im Nachrichtenumschlag, das alle Ereignisse eines Geschäftsablaufs gemeinsam haben. Sein Gegenstück `causation_id` zeigt auf das Ereignis oder Command, das dieses direkt verursacht hat. Kapitel 11.

Dead Letter. Eine Nachricht, die beiseitegelegt wurde, weil ihre Verarbeitung fehlgeschlagen ist. Als Bestandteil eines Kompositums wird der Begriff mit Bindestrichen geschrieben: Dead-Letter-Queue. Kapitel 8.

Dead-Letter-Queue (DLQ). Ein separates Ziel, in dem ein Consumer oder Broker Nachrichten ablegt, die er aufgegeben hat. Bei einem logbasierten Broker heißt dieselbe Idee **Dead-Letter-Topic**. Kapitel 8.

Deadline. Der Zeitpunkt, nach dem ein Ergebnis nicht mehr nützlich ist. Retries müssen vorher enden. Kapitel 6.

Deduplizierungsfenster. Der Zeitraum, in dem ein Consumer verarbeitete IDs behält. Eine Nachricht, deren Inbox-Zeile gelöscht wurde, kann erneut verarbeitet werden. Kapitel 7.

Dual-Publish. Dieselbe Tatsache in zwei Formen oder an zwei Ziele veröffentlichen, während einer Migration: beide Felder oder beide Topics während einer Schemaänderung (Kapitel 12), beide Broker während einer Broker-Migration (Kapitel 14). Bei einer Broker-Migration führt der Relay eine zweite Markierung für das neue Ziel, sodass der alte Broker vollständig bleibt und der neue gegen ihn abgeglichen werden kann.

Dual Write. Zwei Systeme aktualisieren, etwa eine Datenbank und einen Broker, ohne gemeinsame Transaktion. Kapitel 4.

Duplikat. Eine Nachricht, die mehr als einmal verarbeitet oder zugestellt wurde. Kapitel 2.

Ereignis. Eine Nachricht, deren Inhalt eine Tatsache ist, die bereits eingetreten ist, benannt in der Vergangenheitsform. Kapitel 11.

Ereignisjournal. Eine Append-only-Aufzeichnung von Nachrichtenumschlägen von Ereignissen, ohne Payloads, die ein Service neben seiner Inbox führt, damit Fragen auch dann beantwortet werden können, wenn der Broker die Ereignisse vergessen hat. Kapitel 13.

Ereignis-Log. Die dauerhafte, Append-only geführte Folge von Ereignissen. Kapitel 2.

erneute Zustellung. Der Broker stellt eine Nachricht erneut zu, zum Beispiel nach einem ausgebliebenen Ack oder einem abgelaufenen Visibility-Timeout. Ein Replay ist etwas anderes. Kapitel 2.

event_id. Das Feld im Nachrichtenumschlag, das ein Ereignis identifiziert. Es wird erzeugt, wenn die Outbox-Zeile geschrieben wird, und nie neu generiert, und es ist der übliche Idempotenzschlüssel für den Konsum. Kapitel 4 und 7.

Eventual Consistency. Die Eigenschaft, dass Replikate oder Services zum selben Zustand konvergieren, wenn die Aktualisierungen aufhören, ohne Schranke dafür, wie lange das dauert. Kapitel 1.

Exactly-once. Im Buch nur als qualifizierter Effekt verwendet: At-least-once-Zustellung plus ein idempotenter Consumer, was das Buch auch Effectively-once nennt. Es ist keine Zustellgarantie. Die Exactly-once-Verarbeitung eines Brokers gibt es tatsächlich innerhalb der Grenze dieses Brokers, und sie endet an ihr. Kapitel 2.

Expand and Contract. Ein Migrations-Pattern, das eine inkompatible Änderung in mehrere kompatible Schritte verwandelt: beide Formen unterstützen, alle Leser migrieren, dann die alte Form entfernen. Kapitel 12.

fachlicher Schlüssel. Ein Identifikator, der aus der Bedeutung der Arbeit abgeleitet wird, etwa eine Zahlungs-ID plus eine Operation, statt aus dem Nachrichtenumschlag. Er schützt davor, dass dieselbe Arbeit in zwei verschiedenen Nachrichten eintrifft. Kapitel 7.

Failover. Ein vorgesehener Wechsel auf eine andere Kopie einer Broker-Partition, einer Datenbank oder eines Clusters, wenn die aktive ausfällt. Dabei kann verloren gehen, was noch nicht repliziert war. Kapitel 3 und 14.

Fat Event. Ein Ereignis, das genug Zustand mitführt, damit Consumer handeln können, ohne beim Producer nachzufragen. Das Gegenstück ist ein **Thin Event**. Kapitel 11.

Head-of-Line-Blocking. Ein Datensatz, den ein Consumer nicht überspringen will, blockiert jeden Datensatz dahinter in derselben Partition oder demselben Stream. Kapitel 8.

Hold-back-Buffer. Eine dauerhafte Tabelle, in der ein Consumer ein Ereignis zurückhält, dessen Vorgänger noch nicht eingetroffen ist, und es freigibt, wenn die Lücke sich schließt oder ein Timeout abläuft. Kapitel 9.

idempotenter Consumer. Ein Consumer, bei dem die wiederholte Verarbeitung derselben Nachricht denselben Effekt hat wie die einmalige. Kapitel 7.

Idempotenzschlüssel. Der Schlüssel, mit dem ein Consumer oder eine API wiederholte Arbeit erkennt. Beim Konsum von Ereignissen ist es normalerweise die `event_id`. Kapitel 7.

Inbox. Die Tabelle der verarbeiteten Nachrichten-IDs auf der Consumer-Seite, geschrieben in derselben Transaktion wie die Effekte des Handlers. Kapitel 7.

Integrationsereignis. Ein Ereignis, das als bewusster öffentlicher Vertrag veröffentlicht wird, aus einem Domänenereignis abgebildet und frei von personenbezogenen Daten und Secrets. Kapitel 11.

Invariantenprüfung. Eine geplante Abfrage für eine Regel, die laut Geschäft gilt, etwa dass jede erfasste Zahlung genau einen Ledger-Eintrag hat. Jede zurückgegebene Zeile ist eine Abweichung. Kapitel 13.

Jitter. Zufälligkeit, die einer Retry-Verzögerung hinzugefügt wird, damit viele Clients nicht im Gleichschritt wiederholen. „Exponential Backoff mit Jitter“ ist die Standardformulierung. Kapitel 6.

Kompensation. Der Schritt, der einen früheren Saga-Schritt semantisch rückgängig macht, etwa einen Bestand freizugeben oder eine Autorisierung zu stornieren. Das Buch nennt sie nicht Rollback. Kapitel 10.

konkurrierende Consumer. Mehrere Consumer, die sich eine Warteschlange teilen, sodass jede Zustellung an einen von ihnen geht. Eine erneut zugestellte Nachricht kann einen anderen Consumer erreichen. Sie

sind das Gegenstück der Warteschlangen-Familie zu einer Consumer-Group. Kapitel 2.

Message-Broker. Die Infrastruktur, die Nachrichten zwischen Services transportiert. Im Buch kommen zwei Familien vor: logbasierte und warteschlangenbasierte. Kapitel 2.

Nachricht. Eine Dateneinheit, die über einen Broker gesendet wird: ein Ereignis, ein Command oder ein Dokument. Kapitel 11.

Nachrichtenumschlag. Die Standardmenge von Feldern, die mit jedem Ereignis reist, etwa `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` und `causation_id`, getrennt vom Payload. Kapitel 11.

Offset. Die Position eines Consumers in einer Partition eines Ereignis-Logs. Kapitel 2.

Offset-Commit. Die Aufzeichnung des Consumers bei einem logbasierten Broker, dass er alles vor einer Position verarbeitet hat. Sie ersetzt den Ack pro Nachricht. Kapitel 2.

Offset-Reset. Die festgeschriebene Position einer Consumer-Group verschieben, meist rückwärts auf einen Zeitstempel mit Sicherheitsspanne, damit die Historie erneut gelesen wird. Offsets gehören zu einem Cluster und lassen sich nicht auf einen anderen übertragen. Kapitel 14.

Outbox. Die Tabelle, die in derselben lokalen Transaktion wie die Zustandsänderung geschrieben wird und aus der ein Relay Ereignisse veröffentlicht. Der vollständige Begriff lautet **Transactional Outbox**. Kapitel 4.

Parking Lot. Ein Ablageort für Dead Letters, die eine menschliche Entscheidung brauchen, getrennt von der DLQ selbst. Kapitel 8.

Partition. Eine Einheit eines logbasierten Topics, die geordnet ist und jeweils von einem Consumer einer Group gelesen wird. Kapitel 2 und 9.

Partitionsschlüssel. Der Aggregat-Identifikator, mit dem Ereignisse zu einer Partition geleitet werden und so die Reihenfolge pro Aggregat erhalten bleibt. Kapitel 9.

Pivot-Schritt. Der Punkt in einer Saga, nach dem der Workflow nicht mehr kompensiert und nur noch vorwärts geht. Kapitel 10.

Poison Message. Eine Nachricht, die bei jeder Verarbeitung fehlschlägt. Bei einem logbasierten Broker lässt eine solche Nachricht, die den Prozess zum Absturz bringt, den Consumer an einer Position hängen, sodass sie über Lag ohne Durchsatz und über einen Zähler pro Position erkannt wird. Kapitel 8.

Producer. Ein Prozess oder Service, der Nachrichten veröffentlicht, und der Verantwortliche für die Ereignisse, die er aussendet. Kapitel 11.

Rebalance. Die Neuzuweisung von Partitionen, wenn Mitglieder einer Consumer-Group beitreten oder gehen oder wenn ein Mitglied als nicht reagierend gilt. Zwei Consumer können kurzzeitig an derselben Partition arbeiten. Kapitel 3.

Recovery Point Objective (RPO). Wie viele Daten Sie sich leisten können zu verlieren, gemessen in Zeit. Ein Backup alle sechs Stunden bedeutet ein RPO von bis zu sechs Stunden, und jeder Store hat sein eigenes. Kapitel 14.

Recovery Time Objective (RTO). Wie lange Sie ohne den Service auskommen können. Für eine wiedergegebene Projektion ist es eine Abflussberechnung. Kapitel 14.

Redrive. Dead Letters an den Consumer zurückgeben, an dem sie gescheitert sind, an dessen eigenen Eingang und nie an den gemeinsamen Topic, aus dem sie stammen. Der Schlüssel und die `event_id` bleiben erhalten, die Anzahl der Versuche beginnt von vorn, und `redrive_count` hält die Runde fest. Kapitel 8.

Registry-Revision. Der Identifikator, den eine Schema-Registry jedem registrierten Schema zuweist. Sie rückt bei jeder registrierten Änderung vor, auch bei additiven, und ist eine andere Zahl als die **Schemaversion** und die **Aggregatversion**. Kapitel 12.

Relay. Der Prozess, der Outbox-Zeilen zum Broker bringt. Kapitel 5.

Replay. Die Historie eines Ereignis-Logs absichtlich erneut lesen. Kapitel 2 definiert ihn; Kapitel 7, 12, 13, 14 und 15 verwenden ihn.

Retry-Budget. Eine Begrenzung der Retries als Anteil am gesamten Verkehr statt pro Aufruf. Kapitel 6.

Retry-Storm. Eine Welle von Retries, die einer angeschlagenen Abhängigkeit zusätzliche Last aufbürdet und sie am Boden hält. Kapitel 6.

Saga. Ein langlaufender Workflow aus lokalen Transaktionen mit Kompensationen, entweder von einem Koordinator orchestriert oder über Ereignisse choreografiert. Kapitel 10.

Schemaevolution. Die Form von Ereignissen im Lauf der Zeit ändern, ohne die vorhandenen Leser und Schreiber zu brechen. Kapitel 12.

Schema-Registry. Ein Service, der Schemas speichert, Identifikatoren vergibt und ein vorgeschlagenes Schema gegen frühere prüft. Kapitel 12.

Schemaversion. Die ganzzahlige `schema_version` im Nachrichtenumschlag, die Kapitel 12 auch die inkompatible Version nennt. Sie rückt nur bei einer Änderung vor, die ein alter Leser nicht verarbeiten kann, und ist eine andere Zahl als die **Registry-Revision** und die **Aggregatversion**. Kapitel 12.

Schonfrist. Das Alter, das ein Datensatz erreichen muss, bevor ihn ein Abgleichs-Job vergleicht, damit Ereignisse, die noch unterwegs sind, nicht als Abweichungen gemeldet werden. Kapitel 13.

semantische Sperre. Ein Statusflag in den Daten, das besagt, dass eine Saga an einem Datensatz arbeitet, und auf das anderer Code Rücksicht nehmen soll. Es ist keine Datenbanksperre. Kapitel 10.

Spiegelung. Nachrichten mit einem Tool zwischen Brokern kopieren: ein Mirroring-Tool zwischen zwei Logs oder eine Bridge oder ein Connector zwischen einem Log und einer Warteschlange. Sie überträgt auch die Historie aus der Zeit vor einer Migration und kann einen Standby-Cluster versorgen, und die Offsets auf der Kopie sind ihre eigenen. Kapitel 14.

synthetisches Ereignis. Gekennzeichneter Verkehr, den eine Probe in die laufende Pipeline einspeist und am Ende des Ablaufs prüft. Kapitel 15.

Tolerant Reader. Ein Consumer, der aus einem Ereignis nur die Felder nimmt, die er braucht, und den Rest ignoriert. Kapitel 12.

Tombstone. Der Schlüssel, die letzte Version und eine Löschmarkierung, die ein Consumer nach dem Löschen der Zeile eines Aggregats behält, damit ein verspätetes Ereignis sie nicht neu anlegen kann. Kapitel 9.

Topic. Ein benannter Strom von Nachrichten in einer der beiden Broker-Familien. Kapitel 2.

Trace-Kontext. Die Trace-ID und Span-ID, die eine Arbeitseinheit in einem verteilten Trace verorten, in die Outbox-Zeile übernommen und in Nachrichten-Headern gesendet, damit der Span des Consumers auf den des Producers verweisen kann. Kapitel 13.

Umschaltung. Der Schritt einer Broker-Migration, in dem eine Consumer-Group beginnt, vom neuen Broker zu lesen, jeweils für eine Group, nachdem die beiden Broker abgeglichen wurden und mit geplantem Rollback. Kapitel 14.

Upcaster. Code, der ein Ereignis, das unter einem alten Schema geschrieben wurde, in das aktuelle umwandelt, bevor die Geschäftslogik es sieht. Kapitel 12.

Verknüpfung (Trace). Ein Verweis von einem Consumer-Span auf den Trace-Kontext des Producers, der statt einer Eltern-Kind-Beziehung verwendet wird, wenn die Zustellung verspätet, wiederholt oder gebündelt erfolgt. Kapitel 13.

Versionsprüfung. Ein Consumer vergleicht die Version in einem Ereignis mit der letzten Version, die er für dieses Aggregat angewendet hat, um veraltete Ereignisse zu ignorieren und Lücken zu erkennen. Kapitel 9.

Visibility-Timeout. Bei einem warteschlangenbasierten Broker die Zeit, die eine zugestellte Nachricht verborgen bleibt, bevor sie erneut zugestellt wird, falls sie nicht per Ack bestätigt wurde. Kapitel 2.

Vorwärtskompatibilität. Ein Consumer mit dem alten Schema kann Ereignisse lesen, die mit dem neuen Schema geschrieben wurden: alte Leser, neue Daten. Kapitel 12.

Warteschlange. Eine Broker-Struktur, in der jede Nachricht einzeln konsumiert und bestätigt wird. Kapitel 2.

zurückgestellt. Die Retry-Klasse für einen Ausfall, der nichts über die Nachricht aussagt, weil bekannt ist, dass die Abhängigkeit ausgefallen ist, etwa wenn ein Circuit Breaker offen ist. Eine zurückgestellte Nachricht verbraucht keinen Versuch und wird deswegen nie in die Dead Letters verschoben. Der Consumer pausiert, und die Nachricht wartet. Kapitel 6.

B. Kurzreferenz der Patterns

Jede Zeile nennt den Ausfall, den das Pattern behandelt, in den Begriffen des Buches, und was es Sie kostet. Ein Pattern verlagert einen Ausfall an eine Stelle, an der Sie ihn behandeln können; es beseitigt den Ausfall nicht. Kapitel 16 macht aus diesen Zeilen eine Checkliste für das Review.

Pattern	Behandelter Ausfall	Kosten	Kapitel
Transactional Outbox	Dual Write zwischen einer Datenbank und einem Broker	Zusätzliche Tabelle, ein Relay, der betrieben werden muss, an Consumer weitergereichte Duplikate	4
Polling-Relay	Outbox-Zeilen mit wenig Infrastruktur zum Broker bringen	Polling-Verzögerung, Last auf der Datenbank, sorgfältiges Cursor-Design	5
Change Data Capture	Outbox-Zeilen mit geringer Verzögerung aus dem Datenbank-Log veröffentlichen	Betriebsaufwand, Connector-Failover, Aufbewahrungsdauer des Logs, die verwaltet werden muss	5
Exponential Backoff mit Jitter	Retries, die eine sich erholende Abhängigkeit überlasten	Zusätzliche Latenz, Abstimmung von Verzögerungen und Versuchslimits	6
Retry-Budget	Retries, die einen Ausfall verstärken	Manche Aufrufe schlagen schnell fehl, die hätten gelingen können	6
Backpressure und Flusskontrolle	Eine Abhängigkeit oder ein Consumer, der mehr Arbeit bekommt, als er aufnehmen kann	Lag, pausierte Consumer und abgewiesene Arbeit, eine fachliche Entscheidung darüber, was wartet	6
Circuit Breaker	Wiederholte Aufrufe an eine ausfallende Abhängigkeit	Zustand, der abgestimmt werden muss; Aufrufe werden abgewiesen, solange der Breaker offen ist	6
Retry-Topic oder Warteschlange mit Verzögerung	Retry-Verzögerungen, die zu lang sind, um sie im Handler abzuwarten	Mehr Topics, mehr bewegliche Teile, schwierigere Reihenfolge	6
Idempotenter Consumer mit Inbox	Duplikate durch At-least-once-Zustellung	Tabellenschreibvorgänge bei jeder Nachricht, Aufbewahrungsdauer, die verwaltet werden muss	7

Pattern	Behandelter Ausfall	Kosten	Kapitel
Fachlicher Schlüssel	Dieselbe Arbeit trifft in verschiedenen Nachrichten ein	Schlüsseldesign pro Operation und Unique Constraints	7
An nachgelagerte Systeme weitergereichter Idempotenzschlüssel	Doppelte externe Seiteneffekte	Erfordert Unterstützung durch den nachgelagerten Service	7
Dead-Letter-Queue	Poison Messages, die in einer Schleife erneuter Zustellung hängen	Ein Haufen, der eine verantwortliche Person, Alarme und einen Reparaturpfad braucht	8
Parking Lot	Dead Letters, die eine menschliche Entscheidung brauchen	Handarbeit und eine klare Verantwortungsregel	8
Redrive	Dead Letters wiederherstellen, nachdem die Ursache behoben ist	Duplikate, Umsortierung und Last auf dem Hauptfluss	8
Consumer-Outbox	Ein Consumer, der Zustand ändert und eine Folgenachricht veröffentlicht, verliert eines von beidem	Eine Outbox auf der Consumer-Seite und ein Relay, der betrieben werden muss	7, 10
Partitionsschlüssel pro Aggregat	Ereignisse in falscher Reihenfolge innerhalb eines Aggregats	Hot Partitions, Grenzen der Parallelität	9
Versionsprüfung im Consumer	Veraltete oder falsch geordnete Ereignisse überschreiben neueren Zustand	Ein Versionsfeld und eine Entscheidung für jede Lücke	9
Hold-back-Buffer	Lücken in einer Folge, die der Reihe nach angewendet werden muss	Speicher, Timeouts und die Lücke, die sich nie schließt	9
Saga (orchestriert oder choreografiert)	Ein fachlicher Workflow, der mehrere Services ohne eine Transaktion umspannt	Zustände, Timeouts und Kompensationen, die entworfen und betrieben werden müssen	10
Kompensation	Abgeschlossene Saga-Schritte nach einem späteren Ausfall rückgängig machen	Manche Schritte lassen sich nicht rückgängig machen; der Pivot-Schritt markiert die Grenze	10
Semantische Sperre	Gleichzeitige Änderungen an einem Datensatz, an dem eine Saga arbeitet	Blockierte Operationen, solange die Saga läuft	10

Pattern	Behandelter Ausfall	Kosten	Kapitel
Wire-Typen pro Feld sprachübergreifend festgelegt	Dasselbe Feld des Nachrichtenumschlags wird von zwei Runtimes unterschiedlich gelesen	Disziplin in jeder Sprache und Tests, die Sprachen überschreiten	11
Standardisierter Nachrichtenumschlag	Ereignisse, die Tooling und Consumer nicht nachverfolgen oder weiterleiten können	Disziplin bei jedem Producer	11
Thin oder Fat Event	Die Wahl zwischen Rückfragen und mitgeführtem Zustand	Thin: Kopplung zur Lesezeit. Fat: größerer Vertrag und kopierte Daten	11
Schema-Registry mit Kompatibilitätsprüfungen	Producer, die Consumer unbemerkt brechen	Ein Service, der betrieben werden muss, und ein Prozess, dem man folgen muss	12
Tolerant Reader und Upcaster	Leser, die an neuen Feldern oder alten Versionen scheitern	Konvertierungscode, der für jede historische Version vorgehalten wird	12
Expand and Contract	Eine inkompatible Änderung, die sich nicht vermeiden lässt	Mehrere Deployments, Dual-Publish, eine längere Migration	12
Trace-Kontext und Correlation-ID	Die Geschichte eines Ablaufs über asynchrone Hops hinweg verlieren	Weitergabe in jedem Producer und Consumer	13
Abgleich und Reparatur	Unbemerktes Auseinanderlaufen von Quellsystem und abgeleitetem Zustand	Vergleichs-Jobs, Reparatur-Tooling und Review der Ergebnisse	13
Ereignisjournal	Fragen zu Ereignissen, die der Broker vergessen hat	Eine Tabelle, die bei jeder Nachricht geschrieben wird, und Aufbewahrungsdauer, die verwaltet werden muss	13
Backfill	Abgeleiteter Zustand, der aus vorhandenen Daten neu aufgebaut oder aufgefüllt werden muss	Batches mit Ratenbegrenzung und ein Pausenschalter, und die Last auf einem laufenden Consumer	13
Invariantenprüfungen	Schäden auf fachlicher Ebene, die keine technische Metrik zeigt	Die Invarianten definieren und die Alarme verantworten	13

Pattern	Behandelter Ausfall	Kosten	Kapitel
Replay per Offset-Reset	Ein Consumer, der falsch war, oder ein Store, der in der Zeit zurückgesprungen ist, und der die Historie erneut braucht	Ein Deduplizierungsfenster, das den Bereich abdeckt, unterdrückte Seiteneffekte und die Aufbewahrungsdauer als Grenze	14
Recovery Point und Recovery Time Objectives pro Store	Katastrophen, die durch Improvisation bewältigt werden, ohne vereinbarten Verlust oder Ausfallzeit	Eine Zahl, die für jeden Store gemessen und geübt werden muss	14
Clusterübergreifende Replikation zur Wiederherstellung	Verlust eines Broker-Clusters oder einer Region	Ein Standby, der betrieben werden muss, eine Replikationsverzögerung, die das RPO ist, und Offsets, die sich nicht übertragen lassen	14
Dual-Publish und Umschaltung	Zu einem anderen Broker wechseln, ohne Ereignisse zu verlieren oder Effekte zu duplizieren	Eine zweite Markierung in der Outbox, Abgleich und eine langsamere Migration	14
Consumer-Driven-Contract-Tests	Vertragsdrift zwischen Teams	Pflege der Tests und Einigung darüber, wer sie ausführt	15
Handler- und Störungstests	Handler, die nie unter Duplikaten, Verzögerung oder Umsortierung getestet wurden	Testinfrastruktur mit echter Datenbank und langsamere Test-Suites	15
Fehlerinjektion	Einträge des Ausfallmodells, die nie in einem laufenden System geprüft wurden	Eine eigene Umgebung, ein Auswirkungsbereich, der begrenzt werden muss	15
Replay-basierter Regressionstest	Änderungen, die das Verhalten bei echter Historie brechen	Kuratierte, anonymisierte Beispieldaten, die gepflegt werden müssen	15
Synthetische Ereignisse in der Produktion	Unbemerkte Ausfälle im laufenden Pfad	Testdaten, die von echten Geschäftsdaten getrennt werden müssen	15

So lesen Sie die Tabelle

Viele Zeilen treten paarweise auf: ein Pattern, das einen Ausfall behebt, und ein zweites Pattern, das den Ausfall behebt, den das erste eingeführt hat. Die Outbox erzeugt Duplikate, also braucht der Consumer eine Inbox.

Retries erzeugen Last, also brauchen sie ein Budget. Eine Dead-Letter-Queue erzeugt einen Haufen, also braucht sie eine verantwortliche Person und eine Redrive-Prozedur.

Diese Kette ist die Form des Buches. Wenn Sie eine Zeile übernehmen und ihr Gegenstück überspringen, ist der Ausfall verlagert, aber nicht verschwunden. Meiner Ansicht nach ist die Tabelle am nützlichsten, wenn man sie von rechts nach links liest: Beginnen Sie mit dem Ausfall, den Ihr System heute hat, finden Sie die Zeile, die ihn benennt, und lesen Sie die Spalte mit den Kosten, bevor Sie das Kapitel lesen.

ÜBER DAS BUCH

Zuverlässige ereignisgesteuerte Systeme werden um die Ausfallsemantik herum gebaut, nicht um Message-Broker.

An jeder Nahtstelle, an der eine Komponente Arbeit an eine andere übergibt, stellt das Buch dieselben drei Fragen: was ausfallen kann, wie Sie es beobachten würden und welche Garantie den Ausfall übersteht. At-least-once-Zustellung wird als Tatsache behandelt, nicht als Mangel; deshalb sind Handler idempotent, und Duplikate, verlorene Nachrichten, Retries und Ankunft außer der Reihe werden eingeplant, statt sie wegzuhoffen. Anhand von Parcelly geht das Buch die Transactional Outbox, Relays und Change Data Capture, Retries, Idempotenz, Dead Letters, Nachrichtenreihenfolge, Sagas, Schemaevolution, Observability und Wiederherstellung durch, jeweils mit den Kosten, die sie mit sich bringen.

ÜBER DEN AUTOR

Muhammet Şafak entwickelt seit 2008 Software. Über viele Jahre war Muhammet in den Bereichen Gesundheitswesen, Finanzen und Telekommunikation aktiv tätig und vertritt die Überzeugung, dass Entwicklung beim Problem ansetzen sollte, nicht bei der Technologie. Seit 2024 entwirft und entwickelt Muhammet hochfrequentierte Systeme, die durchgängig skalieren.

Muhammet bietet kleinen und mittelgroßen Softwareentwicklungsteams Mentoring an, leitet Open-Source-Projekte und hält Entscheidungen schriftlich fest – mit ihren Gründen und ihrem Preis.

Zuverlässige Systeme werden nicht aus zuverlässigen Teilen gebaut. Sie werden um die Teile herum gebaut, die ausfallen.